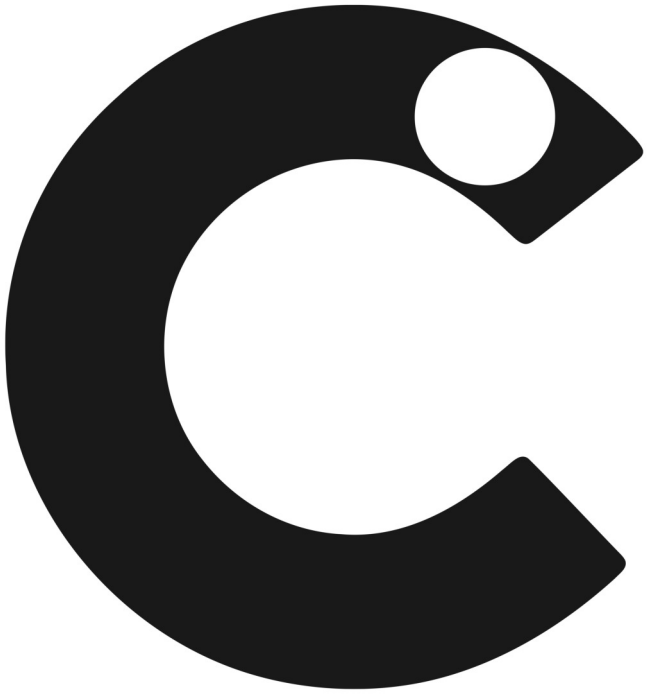

CCalc Technical Reference

By Fortunato Caragliano

Version 53.15 (2026)

Appendix A - General Input Extraction Guide

Overview

Many tools in CCalc do not require the user to enter a complete mathematical expression exactly as it appears in a textbook, worksheet, engineering note, or problem statement. Instead, they ask the user to identify the important parts of that expression and place those parts into separate fields.

This appendix explains that process.

In CCalc, this process is called input extraction. It means translating a written formula, function, equation, relation, or word problem into the structured values expected by the selected tool.

This guide is intentionally broad. It is not limited to one docklet. It is most directly useful in expression-based and template-based tools, but the same method also applies, in a broader sense, to parameter-based workflows such as Financials and Combinatorics, and to dimensional workflows such as the Converter.

Some CCalc areas, such as the Grapher, do accept direct expression entry. This appendix is mainly about the tools that use structured prompts, parameter fields, modes, bounds, units, or ordered lists instead of one continuous expression.

In practical terms, the workflow is:

1. Identify what kind of mathematical object or problem you are looking at.
2. Determine which values define it.
3. Rewrite it into the standard form expected by the tool.
4. Enter those values into the matching prompt fields.
5. Check signs, order, bounds, modes, and units before computing.

This makes structured tools in CCalc easier to use, more predictable, and less error-prone.

General Principle

The central rule is simple:

Do not try to enter the problem exactly as it is written. First rewrite it into the structure the selected CCalc tool expects.

This is the key to using structured prompt fields correctly.

In practice, that means identifying the parts of the problem that matter computationally, such as:

- variables
- constants
- coefficients
- exponents
- bounds
- evaluation points
- modes
- rates
- units
- lists of values

The user is therefore not merely copying mathematics into the app. The user is translating it into calculator-ready form.

General Structure

The input-extraction process in CCalc can be thought of as having two levels.

1. Source Form

This is the form in which the problem is originally presented. It may appear as:

- a formula
- a function
- an equation
- a word problem
- a rate expression

- a dimensional statement
- a list of values

Examples:

- f(x) = 3x² − 5x + 2
- ∫ from 1 to 4 of (x² + 1) dx
- “A loan has present value 10,000 at 1% per month for 36 periods.”
- “Convert 25 foot-pound force to newton meter.”

This is the user-facing or source form.

2. Structured Input Form

This is the form that CCalc expects internally.

Instead of entering the original statement exactly as written, the user often enters:

- coefficients
- constants
- rates
- period counts
- interval endpoints
- evaluation points
- unit selections
- ordered value lists

Examples:

- $a_0 = 2$, $a_1 = -5$, $a_2 = 3$
- lower bound = 1, upper bound = 4
- PV = 10000, rate = 1, NPER = 36
- category = Torque, from = Foot Pound Force, to = Newton Meter

This is the structured form used by the selected tool.

How Input Extraction Works

A reliable extraction workflow usually follows the same sequence.

1. Identify the kind of object

First determine what kind of input you are dealing with.

For example, it may be:

- a polynomial
- a power function
- an exponential function
- a logarithmic function
- a trigonometric function
- a damped sinusoid
- a rational function
- a multivariable expression
- a financial parameter set
- a combinatorial structure
- a unit-conversion task

This first step matters because each type has its own expected input pattern.

2. Identify the role of each symbol

Next determine which parts of the expression represent:

- variables
- coefficients
- constants
- exponents
- bounds
- rates

- directions
- units
- named parameters

The variable is what changes. The entered values are what define the problem.

3. Rewrite into standard form

Before entering values, rewrite the problem mentally or on paper into the form expected by the prompt fields.

4. Enter the values into the matching fields

Once the structure is clear, enter each extracted value into the field that corresponds to its role.

5. Check sign, order, and meaning

Before computing, confirm that:

- signs have been preserved
- the terms are in the right order
- missing terms have been treated correctly
- units are correct
- rates are interpreted correctly
- bounds and points are in the intended positions

Variables, Constants, and Coefficients

A common source of confusion is the difference between a variable and a coefficient.

For example, in:

$f(x) = 7x^2 - 4x + 9$

- x is the variable
- 7 is the coefficient of x^2
- -4 is the coefficient of x
- 9 is the constant term

In:

$f(x, y) = 2 + 3x - 5y + 4x^2 + 6xy - y^2$

- x and y are variables
- 2, 3, -5, 4, 6, -1 are entered values
- each entered value belongs to a specific term type

The variable is not the main input. The defining values are.

Standard Expression Forms

Many CCalc prompt systems expect expressions in recognizable standard forms. The user's task is to map the written expression into the correct one.

Constant Form

Standard form:

$f(x) = K$

Extract:

- K: the constant value

Example:

$f(x) = 12$

Enter:

- $K = 12$

Linear Form

Standard form:

$f(x) = a_0 + a_1x$

Extract:

- a_0: constant term
- a_1: coefficient of x

Example:

$f(x) = 7 - 3x$

Enter:

- $a_0 = 7$
- $a_1 = -3$

Polynomial Form

Standard form:

$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$

Each coefficient is entered according to the power of the variable.

Example:

$f(x) = 6x^3 - 4x + 1$

Rewrite into standard order:

$f(x) = 1 - 4x + 0x^2 + 6x^3$

Enter:

- $a_0 = 1$
- $a_1 = -4$
- $a_2 = 0$
- $a_3 = 6$

Power Form

Standard form:

$f(x) = Ax^n$

Extract:

- A: coefficient
- n: exponent

Example:

$f(x) = 5x^4$

Enter:

- $A = 5$
- $n = 4$

Example:

$f(x) = -2x^{-3}$

Enter:

- $A = -2$
- $n = -3$

Exponential Form

Standard form:

$f(x) = Ae^{kx}$

Extract:

- A: front coefficient
- k: exponential rate

Example:

$f(x) = 7e^{0.4x}$

Enter:

- A = 7
- k = 0.4

Example:

$f(x) = -3e^{-2x}$

Enter:

- A = −3
- k = −2

Logarithmic Form

Standard form:

$f(x) = A \ln(x)$

Extract:

- A: front coefficient

Example:

$f(x) = 9 \ln(x)$

Enter:

- A = 9

Example:

$f(x) = -\ln(x)$

Enter:

- A = −1

Trigonometric Form

Standard form:

$f(x) = A \sin(\omega x + \varphi)$

or

$f(x) = A \cos(\omega x + \varphi)$

Extract:

- A: amplitude
- ω: angular frequency
- φ: phase

Example:

$f(x) = 4 \sin(3x + \pi/2)$

Enter:

- A = 4
- ω = 3
- φ = π/2

Example:

$f(x) = -2 \cos(5x - 1)$

Enter:

- A = −2
- ω = 5
- φ = −1

Damped Sinusoid Form

Standard form:

$f(x) = Ae^{-\sigma x} \sin(\omega x + \varphi)$

Extract:

- A: amplitude
- σ: damping value
- ω: angular frequency
- φ: phase

Example:

$f(x) = 6e^{-0.5x} \sin(4x + \pi/3)$

Enter:

- A = 6
- σ = 0.5
- ω = 4
- φ = π/3

Rational Form

Standard form:

$f(x) = (a_0 + a_1 x) / (b_0 + b_1 x)$

Extract numerator and denominator separately.

Example:

$f(x) = (3 - 2x) / (5 + 7x)$

Enter:

- $a_0 = 3$
- $a_1 = -2$
- $b_0 = 5$
- $b_1 = 7$

Two-Variable Quadratic Form

Standard form:

$f(x, y) = a_0 + a_x x + a_y y + a_{xx} x^2 + a_{xy} xy + a_{yy} y^2$

Extract:

- a_0: constant term
- a_x: coefficient of x
- a_y: coefficient of y
- a_{xx}: coefficient of x^2
- a_{xy}: coefficient of xy
- a_{yy}: coefficient of y^2

Example:

$f(x, y) = 1 + 4x - 3y + 2x^2 + 5xy - 7y^2$

Enter:

- $a_0 = 1$
- $a_x = 4$
- $a_y = -3$
- $a_{xx} = 2$
- $a_{xy} = 5$
- $a_{yy} = -7$

Missing Terms

A missing term does not mean that the field should be left blank. It means the coefficient is zero.

Example:

$f(x) = 5x^2 + 1$

This must be understood as:

$f(x) = 1 + 0x + 5x^2$

Enter:

- $a_0 = 1$
- $a_1 = 0$
- $a_2 = 5$

This rule is extremely important in structured prompt entry.

Sign Handling

The sign belongs to the value. It must not be lost.

Example:

$f(x) = 4 - 2x + x^2$

Enter:

- $a_0 = 4$
- $a_1 = -2$
- $a_2 = 1$

Typical mistakes include:

- entering a negative term as positive
- moving the minus sign to the wrong field
- treating subtraction as separate from the coefficient

Always preserve the sign exactly.

Implied Coefficients

Some coefficients are not written explicitly, but they are still present.

Examples:

- x^2 means coefficient 1
- $-x^2$ means coefficient -1
- xy means coefficient 1
- $-\sin(x)$ means amplitude -1

If a number is omitted, determine whether it is implicitly 1 or -1.

Order of Terms

The order in which an expression is written is often not the order in which CCalc expects the fields.

Example:

$f(x) = 7x^3 + 2 - 4x$

This should be reordered as:

$f(x) = 2 - 4x + 0x^2 + 7x^3$

Then entered according to the field structure.

What matters is not the printed order. What matters is the standard form.

Equations and Expressions

An expression is not the same as an equation.

Expression:

$3x^2 + 2x - 1$

Equation:

$3x^2 + 2x - 1 = 0$

When CCalc asks for coefficients, it usually expects the expression side in structured form.

Example:

$2x^2 - 7x + 3 = 0$

Extract:

- a = 2
- b = −7
- c = 3

The equation gives the mathematical problem. The coefficients give the input values.

Additional Inputs Beyond Coefficients

Many functions require more than the expression itself.

Additional extracted values may include:

- evaluation point
- lower and upper bounds
- interval for search
- center point
- step size
- direction vector
- mode
- timing choice
- unit selection

These are not coefficients, but they are still part of input extraction.

Evaluation Point

Example: derivative at x = 2

Extract:

- function definition
- $x_0 = 2$

Bounds

Example: definite integral from 1 to 4

Extract:

- function definition
- lower bound
- upper bound

Search Interval

Example: root, minimum, or maximum on a closed interval

Extract:

- function definition
- left endpoint
- right endpoint

Direction Vector

Example: directional derivative along ⟨3, 4⟩

Extract:

- function definition
- evaluation point
- $u_x = 3$

- $u_y = 4$

Special Workflows

Calculus-Specific Extraction

In calculus workflows, extraction usually happens at two levels.

Function Definition

The user must identify the structure of the function.

Operation Context

The user must then identify the condition under which the function is being used, such as:

- derivative at a point
- tangent line at a point
- integral over an interval
- Taylor expansion about a center
- multivariable point
- direction vector
- Fourier evaluation frequency

This means that calculus prompts often combine:

- function-definition data
- task-condition data

Both must be extracted correctly.

Multivariable Inputs

For multivariable tools, both the expression and the evaluation data must be identified.

Example:

$f(x, y) = 2 + x - 4y + 3xy$

Rewrite into full template form:

$2 + 1x - 4y + 0x^2 + 3xy + 0y^2$

Enter:

- $a_0 = 2$
- $a_x = 1$
- $a_y = -4$
- $a_{xx} = 0$
- $a_{xy} = 3$
- $a_{yy} = 0$

If the task is to evaluate at $(1, -1)$, also enter:

- $x = 1$
- $y = -1$

Combinatorics Inputs

Combinatorics generally uses integer parameters rather than symbolic coefficients.

Typical extracted values include:

- n
- r
- k
- grouped or listed values

Example: combinations of 10 objects taken 3 at a time

Extract:

- $n = 10$
- $r = 3$

Example: multinomial split into groups of sizes 3, 2, and 3

Extract:

- n = 8
- list = 3, 2, 3

Financial Inputs

Financial tools are usually parameter-driven rather than coefficient-driven.

Typical extracted values include:

- present value
- future value
- payment
- rate
- number of periods
- timing mode
- inflation
- discount
- tax
- cash-flow list

Example: loan payment setup

Extract:

- PV
- FV
- rate per period
- NPER
- payment timing

Example: APR to APY conversion

Extract:

- conversion mode
- percentage value
- compounds per year

Example: net present value

Extract:

- discount rate
- ordered cash-flow list

Converter Inputs

The Converter usually does not require coefficient extraction. Instead, it requires dimensional extraction.

Typical values to identify are:

- source value
- category
- source unit
- destination unit

Example:

Convert 25 foot-pound force to newton meter.

Extract:

- category = Torque
- value = 25
- source unit = Foot Pound Force
- destination unit = Newton Meter

Practical Notes

Word Problems

Word problems should often be translated into symbolic structure before entry.

Example:

“A quantity starts at 100 and grows at 8% per year for 5 years.”

Extract:

- initial value = 100
- rate = 8%
- time = 5

Example:

“A particle moves according to $s(t) = 4t^2 - 3t + 1$. Find velocity at $t = 2$.”

Extract:

- constant term = 1
- t coefficient = −3
- t^2 coefficient = 4
- evaluation point = 2

Example:

“A surface is $f(x, y) = x^2 + 2xy + 3y^2$. Find the gradient at $(1, -1)$.”

Extract:

- $a_0 = 0$
- $a_x = 0$
- $a_y = 0$
- $a_{xx} = 1$
- $a_{xy} = 2$
- $a_{yy} = 3$
- x = 1
- y = −1

Units and Interpretation

A number is never enough by itself. Its meaning matters.

Examples:

- a rate may be per period rather than per year
- angular frequency may be expected in radians
- temperature must use formula conversion rather than simple scaling
- torque and energy may share SI magnitude relations while still representing different concepts
- currency rates are dynamic rather than fixed conversion factors

Before computing, always ask:

What does this value represent here?

Common Mistakes

The most common extraction errors are:

Missing Zero Coefficients

A missing term should usually be entered as zero.

Lost Minus Signs

Signs must always be preserved.

Swapped Meaning

Do not confuse constant term, linear coefficient, exponent, denominator coefficient, or mode value.

Wrong Order

Always reorder into the standard prompt structure before entering the values.

Wrong Rate Interpretation

A rate per period is not the same as a rate per year.

Wrong Dimensional Meaning

A number can be correct while the selected unit or conceptual interpretation is wrong.

Treating Every Tool as Algebraic

Some docklets are expression-based. Others are parameter-based. Others are dimensional. The extraction habit remains general, but the extracted content changes by domain.

Recommendations for Use

A good extraction process usually answers the following questions:

1. What kind of problem is this?
2. What values define it?
3. What is the standard form expected by the tool?
4. Are any terms missing and therefore equal to zero?
5. Are the signs preserved correctly?
6. Are bounds, points, or modes also required?
7. Are the units or interpretations correct?

If those questions are answered clearly, the input is usually ready.

This guide is most directly applicable to:

- Calculus
- Applied and Engineering tools
- expression-template docklets
- multivariable tools
- formula-based utilities

It is still useful, in a broader sense, for:

- Combinatorics
- Financials
- Converter workflows
- other prompt-based docklets

Appendix B - Constants Reference

Overview

CCalc includes two constants resources.

1. Common Constants in the Constants Docklet

These are the compact, fast-access constants available from the π key.

2. Central Constants Repository

This is the larger constants database accessible from the Settings menu.

The Constants Docklet is optimized for quick insertion during active calculation. The Central Constants Repository is intended as the broader reference library.

C.1 - Common Constants in the Constants Docklet

The Constants Docklet contains a curated set of commonly used constants. These are shown in the compact docklet opened from the π key.

Mathematical, Geometry, and Logarithmic Constants

Symbol	Name	Value
π	Pi	3.14159265358979323846264338327950288
τ	Tau (2π)	6.28318530717958647692528676655900576
e	Euler's number	2.71828182845904523536028747135266249
ln(10)	Natural log 10	2.30258509299404568401799145468436421
$\log_{10}(e)$	Log10(e)	0.43429448190325182765112891891660508
2	Sqrt(2)	1.41421356237309504880168872420969808
3	Sqrt(3)	1.73205080756887729352744634150587236
φ	Golden ratio	1.61803398874989484820458683436563812
γ	Euler-Mascheroni	0.57721566490153286060651209008240243

Fundamental Physics Constants

Symbol	Name	Value
c	Speed of light	299792458
h	Planck constant	6.62607015e-34
ħ	Reduced Planck	1.054571817e-34
G	Gravitational	6.67430e-11
g	Std gravity	9.80665

Electricity and Electronics Constants

Symbol	Name	Value
e_0	Elem charge	1.602176634e-19
ε_0	Vac permittivity	8.854187817e-12
μ_0	Vac permeability	1.25663706212e-6
k_e	Coulomb const	8.9875517923e9
Z_0	Free-space Ω	376.730313668
α	Fine-structure	7.2973525693e-3

Chemistry and Mole Constants

Symbol	Name	Value
k_B	Boltzmann	1.380649e-23
N_A	Avogadro	6.02214076e23
R	Gas constant	8.314462618
F	Faraday	96485.33212
u	Atomic mass unit	1.66053906660e-27

Symbol	Name	Value
pK_w	Water @25°C	14

Practical Conversion References

Symbol	Name	Value
10^3	Thousand	1000
10^6	Million	1000000
10^9	Billion	1000000000
bar	1e5 Pa	100000
atm	101325 Pa	101325
L	1e3 m^3	0.001
hp	745.699872 W	745.699872

Air and Fluid Constants

Symbol	Name	Value
ρ_w	Water density	1000
R_s	Spec gas air	287.05
γ_air	Air heat ratio	1.4
c_p air	Air cp	1005

Materials and Particles

Symbol	Name	Value
E_steel	Steel modulus	200e9
E_al	Al modulus	69e9
ρ_steel	Steel density	7850
ρ_al	Al density	2700
m_e	Electron mass	9.1093837015e-31
m_p	Proton mass	1.67262192369e-27

Identity Reference

Symbol	Name	Value
e^(iπ)	Euler identity	-1

C.2 - Central Constants Repository

The Central Constants Repository is the larger constants database available from the Settings menu.

This repository is separate from the compact docklet. It acts as CCalc's broader constants reference library.

Astronomical

Name	Value
Astronomical unit m	1.49597871e+11
Earth mean (km)	12.756
Earth mass (kg)	5.97216787e+24
Earth/Moon mass ratio (kg)	81.3007
Earth escape velocity (m/s)	1.119e4
Gas const. R	8.31446262
Jupiter mean (km)	140.000
Jupiter mass (kg)	1.8981246e+27
Kiloparsec m	3.08567758e+19
General precession in longitude	5029.0966
Light-year (ly)	9.461e15
Luminosity of Sun (L_{sun})	3.85e26
Pogson's ratio	2.511886431509580
Mars mean (km)	6779

Name	Value
Mercury mean (km)	4879
Neptune mean (km)	49.200
Nominal Earth equatorial rad. (m)	6378100
Nominal Jupiter equat. rad. (m)	71492000
Nominal solar radius (m)	695700000
Parsec m	3.08567758e+16
Pluto mean (km)	2470
Rydberg const. Ryd	10973731.6
Saturn mean (km)	117.000
Sidereal year (y)	3.156e7
Solar const. (W/m^2)	1.368e3
Solar mass (kg)	1.98840987e+30
Solar parallax	8.794148
Speed of light in vacuum ms^{-1}	299792458
Standard atmosphere atm	101325
Uranus mean (km)	50.700
Venus mean (km)	12.104

Conversion Constants

Name	Value
log(2), Natural log of 2	0.693147180559945
Log(2), Decadic log of 2	0.301029995663981
log(10), Natural log of 10	2.302585092994045
log2(10), Binary log of 10	3.321928094887362
Log(e), Decadic log of e	0.434294481903251
log2(e), Binary log of e	1.442695040888963
1 rad in degs	57.295779513082320
1 deg in rads	0.017453292519943
1 rad in arcmin	3437.746770784939252
1 arcmin in rads	2.908882086657215
1 rad in arcsec	206264.806247096355156
1 arcsec in rads	4.848136811095359
Full solid angle of 4π ster. in deg^2	41252.961249419271031
1 sr in deg^2	3282.806350011743794
1 deg^2 in sr	0.000304617419786
1 sr in rad^2	1.041191803606873

Engineering Constants

Name	Value
Amplitude/Effective	1.414213562373095
Amplitude factor of 2 (dB)	6.020599913279623
±1db Ratio: Power	1.258925411794167
±1db Ratio: Inv. power	0.794328234724281
±1db Ratio: Amplitude	1.122018454301963
±1db Ratio: Inv. amplitude	0.891250938133745
±3db Ratio: Power	1.995262314968879
±3db Ratio: Inv. power	0.501187233627272
±3db Ratio: Amplitude	1.412537544622754
±3db Ratio: Inv. Amplitude	0.707945784384137
Half-note frequency ratio	1.059463094359295
Pythagorean comma	1.013643264770507

Name	Value
Power factor of 2 in dB	3.010299956639811
Rumors const.	0.203187869979979

Mathematical Constants

Name	Value
3 - Theodorus's constant	1.7320508075688772935
2D Cantor dust	1.261859507142914
e (Natural log base)	2.7182818284
π	3.1415926536
Alladi-Grinstead const.	0.809394020540
Apéry's const.	1.2020569031595942853
Apollonian gasket	1.305686729
Artin's conjecture	0.373955813619
Asymmetric Cantor set	0.694241913630617
Backhouse's const.	1.456074948582
Barban's const.	2.596536290450
Bernstein's const. β	0.280169499023
Besicovitch const.	0.149162536496
Blazys const.	2.566543832171
Boling's const.	1.805917418986
Brun's const. B2 for twin primes	1.902160583104
Brun's const. B4 for cousin primes	1.1970449
Brun's const. B'_4 for prime quads.	0.870588380
Buffon's const.	0.636619772367
Cahen's const.	0.6434105462883380261
Cantor set, removing 2nd third	0.630929753571457
Catalan's const. G	0.9159655941772190150
Champernowne's const.	0.12345678910111213141516
Conway's const. λ(3)	1.303577269034
Continued fractions const.	1.030640834100712
Copeland-Erdős const.	0.2357111317192329313
Delian const.	1.259921049894
DeVicci's tesseract const.	1.007434756884
Dottie number	0.739085133215
Efimov's scaling const.	22.694382595366
Embree-Trefethen const. β	0.70258
Euler-Mascheroni const.	0.5772156649
Euler's Identity e^(i*pi)	-1
Erdős-Borwein const.	1.6066951524152917637
Feigenbaum attractor-repeller	0.538045143580549
Feigenbaum reduction param. α	-2.502907875095892
Feigenbaum's const. δ	4.6692116609102990671
Feller-Tornier const. F	0.661317049469622
Flajolet-Odlyzko const.	0.757823011268492
Foias const. α	1.187452351126501
Foias-Ewing const. β	2.293166287411861
FoxTrot series sum	0.239560747340741
Fransén-Robinson const.	2.807770242028519
Gascoigne-Moore number	1801049058342701056
Gelfond's const.	23.140692632779
Gelfond-Schneider const.	2.665144142690225

Name	Value
Gibbs const. G	1.851937051982466
Gieseking's const. G	1.014941606409653
Glaisher-Kinkelin const. A	1.282427129100622
Golden ratio (Φ)	1.6180339887
Golomb-Dickman const. λ	0.624329988543550
Gompertz const. G	0.596347362323194
Graham's const. G(3)	0.783591464262726
Grossmann's const.	0.73733830336929
Hafner-Sarnak-McCurley const.	0.353236371854
Hard square entropy const.	1.503048082475332
Harmonic HM(1,π)	1.517093985989
Heat-Brown-Moroz const.	0.001317641154853
Heighway-Harter dragon bound	1.523627086202492
Infinite power tower of 1/π	0.539343498862
Infinite nested radical	2.341627718511478
Kepler-Bouwkamp const.	0.1149420448532962007
Khinchin's const.	2.6854520010653064453
Khinchin-Lévy const. β	1.186569110415625
Kinkelin const.	-0.165421143700450
Knuth's random-generators const.	0.211324865405187
Kolakoski const. γ	0.794507192779479
Komornik-Loreti const. q	1.787231650182965
Mills' const.	1.3063778838630806904
Landau-Ramanujan const.	0.7642236535892206629
Laplace limit const. λ	0.662743419349181
Lehmer's const.	0.5926327182016361971
Lemniscate const. L	2.622057554292119
Lemniscate const. LA	1.311028777146059
Lemniscate const. LB	0.599070117367796
Lévy's const.	0.3275822918721811159
Lévy's C curve	1.934007182988290
Lieb's square ice const.	1.539600717839002
Lieb's 20-Vertex entropy const.	2.598076211353315
Liouville's const.	0.110001000000000
Loch's const.	0.970270114392033
log(2π)/2	0.9189385332046
log(π) (real part log(log(-1)))	1.144729885849
log(π).π	3.596274999729
Madelung's const. M_3	-1.747564594633182
Meissel-Mertens const. B_1	0.2614972128476427837
Menger sponge	2.726833027860842
Mills' const. θ	1.306377883863080
Minkowski-Bower const. b	0.420372339423223
MRB const. (Marvin R. Burns)	0.187859642462067
Murata's const.	2.826419997067591
Niven's const. C	1.705211140105367
Norton's const. B	0.065351425923037
Odlyzko-Wilf const. K	1.622270502884767
Omega const. = Lambert $W_0(1)$	0.567143290409783
Oscillatory-integral MRB, mod.	0.687652368927694

Name	Value
Oscillatory-integral MRB, real part	0.070776039311528
Oscillatory-integral MRB, i part	-0.684000389437932
Otter's const. α	2.955765285651994
Otter's asymptotic const. β_u	0.5349496061
Otter's asymptotic const. β_r	0.439924012571
Plastic const.	1.3247179572447460260
Plouffe's const.	0.147583617650433
Polya's random-walk const. p3	0.340537329550999
Porter's const. C	1.467078079433975
Prévost's const. (recip. Fib)	3.359885666243177
Prévost's recip. even Fib. const.	1.535370508836252
Prévost's recip. odd Fib. const.	1.824515157406924
Prime const. (ρ)	0.4146825098511116602
Prince Rupert's cube const.	1.060660171779821
Pythagoras's const. (2)	1.4142135623
Quadratic Class Number const.	0.881513839725170
Rabbit const.	0.709803442861291
Ramanujan-Soldner const.	1.4513692348833810502
Rauzy fractal boundary r	1.093364164282306
Real numbers with no even dec.	0.698970004336018
Real root of $P(_x)$	-0.686777834460634
Rényi's parking const. m	0.747597920253411
Robbins cube line pick const.	0.661707182267176
Salem number σ_1	1.176280818259917
Sarnak's const.	0.723648402298200
Schwarzschild conic const.	7.389056098930
Shall-Wilson const. Π_2	0.660161815846869
Sierpinski carpet	1.892789260714372
Sierpiński's const.	2.5849817595792532170
Sierpiński's triangle	1.584962500721156
Silver ratio | Silver mean (δs)	2.4142135623
Somos quadratic recurrence σ	1.661687949633594
Sum $1+1/2^2+1/3^3+1/4^4+...$	1.291285997062663
Sum of reciprocals of distinct pow	0.874464368404944
Sum of reciprocals of exp. facts	1.611114925808376
Tanaka's number	906150257
Taniguchi's const.	0.678234491917391
Tetranacci const.	1.927561975482925
Theodorus' const.	1.732050807568877
Thue-Morse const.	0.412454033640107
Tribonacci const.	1.839286755214161
Viswanath's const.	1.1319882487943
Vijayaraghavan's nested radicals	1.7579327566180045327
Vojta's number	15170835645
Universal parabolic const.	2.295587149392
Wallis' const.	2.094551481542326
Weierstrass const. $\sigma(1\|1,_i)$	0.474949379987920
Wilbraham-Gibbs const. G'	1.178979744472167
Wyler's const.	0.007297348130031
Zagier's const.	0.180717104711806

Name	Value
Zolotarev-Schur const. σ	0.311078866704819

Physical Constants

Name	Value
Alpha particle mass in u mα	4.001506179129
Angstrom star	1.00001495e-10
Avogadro’s const.	6.02214076e23
Atomic mass const. m_u	1.66053906892e-27
Bohr magneton μ_B	9.2740100657e-24
Bohr radius a_0	5.29177210544e-11
Boltzmann’s const. k	1.380650e-23
Classical electron radius r_e	2.8179403205e-15
Compton wavelength λC	2.42631023538e-12
Conductance quantum	7.748091729e-5
Copper per unit	1.00207697e-13
Deuteron mass in u m_d	2.013553212544
Deuteron magnetic moment μ_d	4.330735087e-27
Deuteron-proton mass ratio m_d/m_p	1.9990075012699
Electron charge to mass quot$_e$	-1.75882000838e11
Electron magnetic moment μ_e	-9.2847646917e-24
Electron magnetic mom. anomaly	1.15965218046e-3
Electron mass m_e	9.1093837139e-31
Electron-muon mass ratio $m_e/m\mu$	4.83633170e-3
Electron-proton magn. mom. ratio	-658.21068789
Electron-proton mass ratio m_e/m_p	5.446170214889e-4
Elementary charge e	1.602176634e-19
Energy equivalent of 1 ton TNT (J)	4.2e9
Faraday const. F	96485.33212
Fermi coupling const.	1.1663787e-5
Fine-structure const. α	7.2973525643e-3
First radiation const. $2\pi hc^2$ c1	3.741771852e-16
Gauss’s const.	0.8346268416740731862
Gravitational const. ($m^3/kg/s^2$)	6.67430e-11
Hartree energy E_h	4.3597447222060e-18
Helion mass in u m_h	3.014932246932
Josephson const. K_j 2e/h	483597.8484e9
Inv. fine-structure const.	137.035999177
Impedance of vacuum	376.730313412
Lattice parameter of silicon	5.431020511e-10
Lattice spacing of ideal silicon	1.920155716e-10
Magnetic flux quantum θ_0	2.067833848e-15
Muon g-factor gμ	2.00233184123
Muon magnetic moment μμ	-4.49044830e-26
Muon magnetic moment anom.	1.16592062e-3
Muon mass in u mμ	0.1134289257
Muon-electron mass ratio $m\mu/m_e$	206.7682827
Muon-proton magn. mom. ratio	-3.183345146
Molar gas const. R	8.314462618
Molar volume of ideal gas V_m	22.41396954e-3
Molar volume of silicon	1.205883199e-5
Molybdenum per unit	1.00209952e-13

Name	Value
Neutron mass in u m_n	1.00866491606
Neutron mass energy eq.	939.56542194
Neutron magnetic moment μ_n	-9.6623653e-27
Neutron-proton mass ratio m_n/m_p	1.00137841946
Nuclear magneton μ_n	5.0507837393e-27
Planck's const. h (JHz^{-1})	6.62607015e-34
Planck lenght	1.616255e-35
Planck mass	2.176434e-8
Planck mass energy eq. in GeV	1.220890e19
Planck temperature	1.416784e32
Planck time	5.391247e-44
Proton gyromagnetic ratio γ_p	2.6752218708e8
Proton magnetic moment μ_p	1.4106067954e-26
Proton magnetic shielding corr.	2.56715e-5
Proton mass m_p	1.67262192595e-27
Proton-electron mass ratio m_p/m_e	1836.152673426
Rydberg frequency cR	3.2898419602500e15
Second radiation const. hc/k c^2	1.438776877e-2
Stefan-Boltzmann const. σ	5.670374419e-8
Tau mass	3.16754e-27
Tau mass in u	1.90754
Tau energy equivalent (J)	2.84684e-10
Thomson cross section σ_e	6.6524587051e-29
Vacuum magn. perm. μ0 $4\pi\alpha\hbar/e^2c$	1.25663706127e-6
Vacuum electric perm. ϱ0 $1/\mu 0c^2$	8.8541878188e-12
Von Klitzing const. R_k	25812.8074
Wien frequency displacement law	58789257570.0
Wien's wavelenght displ. law	2.897771955e-3

Software Engineering

Name	Value
Binary-to-decadic precision	0.301029995663981
Byte (8 bits)	255
Decadic-to-binary precision	3.321928094887362
Dword (double word, 32 bits)	4294967295
gibi	2e30
kibi	2e10
mebi	2e20
Qword (quad word, 64 bits)	18446744073709551
Word (16 bits)	65535

Appendix C - The Ticker

Overview

The ticker is the compact expression and status line used by CCalc to show the current calculation context, recent operations, docklet results, converter output, graphing transfers, warnings, and label-style results.

The ticker is not only a visual history line. It also helps the user understand whether a result is numeric, approximate, descriptive, converted, restored, invalid, or produced by a specialized tool.

Purpose of the ticker

The ticker provides a short explanation of what CCalc just did.

It may show:

- the active expression
- the operation being built
- the function that produced a result
- a docklet formula label
- a converter summary
- a graphing result
- a warning or error message
- a descriptive label instead of a plain number

The main display shows the current operand. The ticker explains the context of that operand.

Ticker structure

A ticker entry usually contains one or more of these parts:

- an expression or operation
- a function name
- input parameters
- a result indicator
- a result tag
- a warning or label message

Examples:

2 + 3 × 4 = 14
sin(2.3 rad) [APPROX]
A◯[L²](r:5) → 78.53981634 [NUM]
MID[point:L]((0;0)→(4;2)) → (2; 1) [LABEL]
Division by 0 [LABEL]

Ticker labels

Ticker labels are short textual descriptions attached to results.

They help distinguish ordinary arithmetic from results produced by docklets, constants, converters, graphing, or special functions.

A ticker label may include:

- the function symbol
- the input value
- a compact formula name
- a unit or dimension marker
- a result classification tag

Examples:

PMT[USD](pv:10000, fv:0, r:1%, n:36, when:0)
DIST[L]((0;0)→(3;4))
Graph y = sin(x), x = 1.5708 → y = 1

Ticker overrides

Some CCalc tools produce results that cannot be fully described by a single plain number.

In those cases, the ticker may use an override.

A ticker override replaces the ordinary expression-style ticker with a more descriptive message.

This is used for results such as:

- confidence intervals
- min/max summaries
- midpoint coordinates
- factorization results
- matrix results
- vector results
- converter summaries
- graphing trace results

Example:

MIN=2, MAX=9
factor(n) → $2^3 \times 3^2 \times 5$
CI=[12.4, 15.8]

Operation display

During ordinary calculation, the ticker can show the expression being built.

Example:

12 + 5 × 3

When the user presses equals, the ticker can show the completed expression and result.
Example:

12 + 5 × 3 = 27

The display contains the current numeric result. The ticker shows how the result was obtained.

Result display

CCalc distinguishes the main result display from the ticker.

The main display shows the active operand.

The ticker shows the context.

For example, after a trigonometric calculation:

Display:

0.74570521

Ticker:

sin(0.842 rad) [APPROX]

The display is the usable numeric result. The ticker identifies the function and indicates that the result is approximate.

Docklet ticker behavior

Docklets use the ticker to show which specialized function produced the result.

Docklet ticker entries may include:

- the function symbol
- the input values
- the dimensional meaning of the result
- the result tag
- a compact explanation of non-scalar results

Examples:

A◯[L^2](r:5) → 78.53981634 [NUM]
P□[L](a:4, b:2) → 19.37689651 [APPROX]
rank(A_2) [NUM]
$CROSS_3$(u,v) = (0; 0; 1) [LABEL]

Some docklet results flow back into the calculator as ordinary operands. Others are descriptive or structured results and are shown as label-style results.

Converter ticker behavior

The converter uses the ticker to explain the conversion that produced the displayed value.

A converter ticker entry may include:

- the source value
- the source unit
- the destination unit

- the converted result
- currency or unit context

Example:

10 meter → centimeter = 1000

Currency conversions may also depend on the selected base currency, cached exchange rates, and the current currency data available to CCalc.

Grapher ticker behavior

The Grapher can send values back to CCalc. When a graphed y-value is returned to the calculator, the ticker identifies the graphing context.

Example:

Graph y = sin(x), x = 1.5708 → y = 1

This allows the returned value to be used as an operand while still preserving its origin in the ticker.

Error and warning messages

The ticker is also used for warning and error states.

Examples include:

Division by 0

Invalid input

Out of range

No real solution

Singular matrix

No inverse

When an operation fails, CCalc may preserve or replace the display depending on the tool. The ticker explains the failure so the user can understand what happened.

Localization and separators

Ticker text follows the selected formatting behavior where supported.

Decimal separators may change depending on the selected locale or decimal separator setting.

For example, in dot-decimal mode:

sin(2.3 rad)

In comma-decimal mode:

sin(2,3 rad)

List separators may also change in some docklet dialogs. For example, statistical data entry may use commas in dot-decimal mode and semicolons in comma-decimal mode.

Technical ticker examples

Ordinary arithmetic:

2 + 3 × 4 = 14

Unary docklet result:

sin(2.3 rad) [APPROX]

Geometry result:

DIST[L]((0;0)→(3;4)) → 5 [NUM]

Statistics result:

$\bar{x}$(n=5) [NUM]

Number theory result:

factor(n) = $2^3 \times 3^2 \times 5$ [LABEL]

Linear algebra result:

A_2^{-1} = [[-2; 1]; [1.5; -0.5]] [LABEL]

Converter result:

100 USD → EUR = 92.13

Graphing result:

Graph y = x^2, x = 4 → y = 16

Warning result:

tan(π/2 rad) = Division by 0 [LABEL]

Appendix D - Result Tags: NUM, APPROX, and LABEL

Overview

CCalc uses result tags to clarify what kind of result has just been produced.

These tags appear in the ticker after many calculations, especially when the result comes from a docklet, converter, graphing tool, or specialized mathematical function.

The three main result tags are:

[NUM]
[APPROX]
[LABEL]

They are not part of the mathematical value itself. They are interpretation markers. Their purpose is to help the user understand whether the displayed result is an ordinary numeric value, an approximate numeric value, or a structured descriptive result.

In practical use:

[NUM] means the result is a normal numeric result.
[APPROX] means the result is numeric, but should be understood as approximate.
[LABEL] means the result is primarily descriptive, structured, symbolic, list-like, or explanatory.

These tags are especially useful because CCalc includes many tools that do more than simple arithmetic. Some tools return one scalar value. Others return vectors, matrices, intervals, lists, labels, equations, summaries, or warnings. Result tags help distinguish these cases without making the main display overly crowded.

Purpose of Result Tags

The purpose of result tags is to make CCalc's output more transparent.

A standard calculator often displays only a number. CCalc frequently displays both:

- the result value
- the meaning of that result

For example, a docklet may compute a statistical result, a trigonometric result, a financial quantity, a geometry value, a matrix property, or a number-theory object. In those cases, the user needs to know not only the number, but also what kind of result it is.

The result tag answers that question.

Example:

sin(2.3 rad) [APPROX]

This tells the user that the value comes from a trigonometric computation and should be read as an approximate numerical result.

Example:

det(A_2) [APPROX]

This tells the user that the determinant is a scalar numeric result, but it was produced through floating-point matrix computation.

Example:

factor(n) [LABEL]

This tells the user that the output is not just a single ordinary number. It may represent a factorization string, such as:

$2^3 \times 3 \times 5$

The tag therefore helps prevent misreading the result.

NUM Results

[NUM] indicates a normal numeric result.

A [NUM] result is a scalar value that can usually be treated as a regular calculator operand.

Typical [NUM] results include:

- arithmetic results
- exact integer-style results
- simple geometry values

- financial scalar results
- count results
- standard unit-conversion outputs
- simple docklet computations that produce one ordinary number

Examples:

A◯[L²](r:5) → 78.53981634 [NUM]
PMT[USD](pv:10000, fv:0, r:1%, n:36, when:0) [NUM]
τ(n) [NUM]
mod(a,n) [NUM]

A [NUM] result means that the displayed value is intended to behave like an ordinary number in the calculator flow. The user may normally continue with another operation.

Example:

A◯(r:5) → 78.53981634
Then:
× 2
continues from the area result.

The tag does not mean that the number is mathematically exact in every possible sense. It means that CCalc is presenting the result as a regular numeric scalar.

APPROX Results

[APPROX] indicates an approximate numeric result.

An [APPROX] result is still numeric. It can usually be used as a calculator operand. However, the user should understand that the value was produced by a method or domain where approximation is expected.

Typical sources of [APPROX] results include:

- trigonometric functions
- inverse trigonometric functions
- hyperbolic functions
- logarithmic and exponential calculations
- statistical distribution functions
- regression and correlation tools
- numerical solvers
- graphing values
- slope estimates
- matrix decompositions
- eigenvalue calculations
- condition numbers
- floating-point geometry approximations
- formulas involving π, square roots, or transcendental functions

Examples:

sin(2.3 rad) [APPROX]
Φ(x;μ,σ) [APPROX]
κ(A₂) [APPROX]
λ₁(A₃) [APPROX]
P□[L](a:5, b:3) → 25.52699886 [APPROX]

The approximation may come from the mathematical nature of the calculation, from decimal rounding, from numerical methods, or from floating-point evaluation.

For example, the perimeter of an ellipse does not have a simple elementary closed form. If CCalc uses an approximation formula, the result should be tagged as approximate.

Similarly, many statistical functions use approximations. The result is useful and often very accurate, but it is still not an exact symbolic value.

LABEL Results

[LABEL] indicates a descriptive, structured, symbolic, list-like, or non-scalar result.

A [LABEL] result is not always meant to behave like a single ordinary number.

Typical [LABEL] results include:

- vectors
- matrices

- factorization strings
- divisor lists
- confidence intervals
- regression equations
- midpoint pairs
- multiple-output summaries
- equation solutions
- symbolic forms
- warnings
- invalid input messages
- list-based results
- explanatory ticker output

Examples:

$\text{CROSS}_3(u,v) = (0; 0; 1)$ [LABEL]

$A_2^{-1} = [[0.6; -0.7]; [-0.2; 0.4]]$ [LABEL]

$\text{factor}(n) = 2^3 \times 3 \times 5$ [LABEL]

CI=[4.82, 6.31] [LABEL]

LINREG = y = 2.5x + 1.2 [LABEL]

MID[point:L] → (3; 5) [LABEL]

A [LABEL] result may still contain numbers, but the output should be read as a structured result rather than as one plain scalar.

This distinction is important.

For example, a midpoint has two coordinates. A matrix inverse has several entries. A factorization has a mathematical structure. A confidence interval has a lower and upper bound. These are meaningful results, but they are not ordinary single-number calculator results.

In some cases, CCalc may place one representative numeric value in the operand for internal continuity, but the meaningful user-facing result is the label shown in the ticker or display. When this happens, the [LABEL] tag tells the user not to interpret the operand alone as the full result.

When Tags Appear

Result tags usually appear in the ticker.

They are most common after:

- docklet calculations
- converter results
- graphing results
- statistical tools
- financial tools
- matrix and vector tools
- number-theory tools
- geometry tools
- error or warning states

A simple arithmetic calculation may not always need a visible tag, because its meaning is already obvious.

Example:

2 + 3 = 5

This does not need a tag.

A docklet result benefits from a tag because the result carries extra meaning.

Example:

SD(n=8) [NUM]

sin(π rad) [APPROX]

$\text{RREF}(A_3)$ [LABEL]

Tags are therefore part of CCalc's clarity system. They help the ticker carry technical information without forcing the main display to explain everything.

How Tags Affect Interpretation

The tag tells the user how to read the result.

NUM

Read the result as a normal scalar number.

Example:

RNG(n=6) [NUM]

This means the range of the data set is a numeric value.

APPROX

Read the result as a scalar number, but with approximation awareness.

Example:

tan(2.3 rad) [APPROX]

This means the value is numerical and usable, but should not be treated as a symbolic exact result.

LABEL

Read the result as an explanation, structure, list, interval, equation, or non-scalar output.

Example:

divisors=1, 2, 3, 6, 9, 18 [LABEL]

This result is not merely the number 6, even if CCalc may internally use a count or representative value. The user-facing result is the divisor list.

How Tagged Results Behave in Continued Calculation

The tag itself does not directly perform the calculation. The underlying result type determines how the result behaves.

However, the tag gives the user a practical clue.

NUM in Continued Calculation

A [NUM] result is generally safe to continue from.

Example:

A□[L²](s:4) → 16 [NUM]

Then:

+ 10 = 26

The result behaves as a normal operand.

APPROX in Continued Calculation

An [APPROX] result can usually also be continued from, but the user should remember that the value is approximate.

Example:

sin(2.3 rad) → 0.74570521 [APPROX]

Then:

× 100

continues from the approximate sine value.

This is valid, but the final result inherits the approximation.

LABEL in Continued Calculation

A [LABEL] result should be interpreted more carefully.

Some [LABEL] results are terminal. This means they are intended to explain or display something rather than become the next ordinary operand.

Examples:

A_2^{-1} = [[0.6; -0.7]; [-0.2; 0.4]] [LABEL]

$CROSS_3$(u,v) = (0; 0; 1) [LABEL]

LINREG = y = 2.5x + 1.2 [LABEL]

These are not single scalar values. Continuing with +, −, ×, or ÷ may not represent the full meaning of the displayed result.

Other [LABEL] results may still commit a representative scalar to the calculator, such as a count, first value, upper bound, or selected component. In those cases, the ticker label explains the full result while the operand holds the scalar value available to the calculator.

The safest user-facing rule is:

If a result is tagged [LABEL], read the ticker carefully before using it as the next operand.

How Tags Relate to the Tape

The tape records calculator activity in a compact form.

Result tags help explain what kind of result was produced, but not every visible ticker message necessarily behaves the same way in the tape.

A standard arithmetic result may appear as a normal expression:

12 × 8 = 96

A docklet result may appear with its function label:

sin(2.3 rad) [APPROX]

A structured result may appear as a descriptive label:

factor(n) = 2^3 × 3 × 5 [LABEL]

Depending on the specific result and the way it enters the calculator flow, the tape may emphasize either:

- the expression and final scalar result
- the docklet label
- the ticker explanation
- the structured output

For this reason, tags are useful when reviewing previous work. They tell the user whether a saved entry was a normal number, an approximate scalar, or a descriptive result.

This is especially helpful when reviewing a mixed session that contains arithmetic, graphing, conversion, statistics, and specialized docklet results.

Ticker Labels, Display Labels, and Operand Labels

Result tags often work together with CCalc's label systems.

Ticker Labels

The ticker is the main place where result meaning is shown.

Example:

PMT[USD](pv:10000, fv:0, r:1%, n:36, when:0) [NUM]

The ticker explains what was computed.

Display Labels

The main display usually shows the current operand or a compact visible result.

For normal scalar results, this is usually a number.

Example:

277.84

For angle-returning trigonometric results, the display may include an angle unit.

Example:

45 deg

Operand Labels

Some results require a label-like display because the result is not naturally represented by one ordinary scalar.

Example:

(3; 5)

or:

2^3 × 3 × 5

In these cases, an operand label may be used to show the meaningful result while preventing the interface from implying that the result is just a simple number.

[LABEL] is the tag most closely associated with this behavior.

Warning and Error States

Warning and error states usually behave like label results.

They are not ordinary numeric results, even when they occur during a numeric workflow.

Examples:

Division by 0 [LABEL]
Invalid input [LABEL]
Out of range [LABEL]
No real solution [LABEL]
Singular matrix [LABEL]

These messages tell the user that CCalc could not produce a valid ordinary result.

Common warning states include:

- invalid input

- missing required value
- division by zero
- out-of-range argument
- impossible geometry
- singular matrix
- no real solution
- unsupported complex result
- modulus not valid
- rate or period inconsistency
- statistical input too small

The tag helps identify these as explanatory states rather than results to continue from.

Invalid Input States

An invalid input state occurs when the values entered into a prompt cannot be parsed or do not satisfy the requirements of the selected function.

Examples:

Invalid input
n must be > 0
r must be ≥ 0
p must be 0..1
Size must be 2 or 3

These states are usually accompanied by warning sound feedback.

They should be interpreted as correction prompts. The user should revise the input and compute again.

Division by Zero

Division by zero is treated as a warning state.

It may occur in ordinary arithmetic or inside a docklet.

Examples:

Division by 0
csc(0 rad) = Division by 0
SLOPE = Division by 0
$\text{inv}_m(a;n)$ = No inverse

A division-by-zero result should not be treated as a number. It is a label-style warning that explains why no valid numeric result was produced.

Out-of-Range Results

Some functions only accept values in a specific domain.

Examples:

arcsin(x)
requires:

$-1 \le x \le 1$

arcosh(x)
requires:

$x \ge 1$

arsech(x)
requires:

$0 < x \le 1$

Geometry functions may also reject impossible dimensions.

Examples:

c must be ≤ 2r
h must be ≤ 2R
Invalid triangle

Linear algebra tools may reject singular or unstable matrices.

Examples:

Singular matrix
Complex eigenvalues not supported
In these cases, the result is not a failed formatting issue. It is a mathematical domain issue.

Formatting and Rounding Behavior

Result tags do not replace formatting rules.

CCalc still applies the active display settings, including:

- decimal places
- decimal separator
- trailing zero behavior
- scientific or standard notation where supported
- rounding behavior
- locale-sensitive separators

A result may be tagged [NUM] or [APPROX] and still be rounded for display.
Example:

sin(2.3 rad)
may display differently depending on decimal-place settings.

With 4 decimal places:

0.7457

With 8 decimal places:

0.74570521

The tag explains the nature of the result. The format settings control how the result is shown.

This distinction is important:

[APPROX] does not mean the value was rounded only because of display settings. It means the result itself should be understood as approximate.
[NUM] does not mean infinite precision. It means the result is being treated as a normal scalar result.
[LABEL] does not mean the output is less important. It means the output is structured or descriptive rather than a single ordinary scalar.

Suggested User-Facing Language

When explaining result tags to users, the following language is recommended.

For [NUM]:

This is a standard numeric result and can normally be used as the next calculator operand.

For [APPROX]:

This is a numeric result, but it was produced by an approximation, numerical method, or non-exact calculation.

For [LABEL]:

This is a descriptive or structured result. Read the ticker or displayed label carefully before using it as a continued calculation.

For warnings:

This is not a numeric result. The message explains why the calculation could not be completed.

Technical Examples

Normal Numeric Result

Input:

Circle Area, r = 5

Ticker:

A◯[L²](r:5) → 78.53981634 [NUM]
Meaning:

The result is an area scalar. It can normally be reused as a number.

Approximate Trigonometric Result

Input:

sin(2.3 rad)
Ticker:

sin(2.3 rad) [APPROX]
Meaning:
The result is numeric, but approximate.

Approximate Matrix Result

Input:
Eigenvalues of A_3
Ticker:
$\lambda_1(A_3)$ [APPROX]
Meaning:
The result is a scalar eigenvalue approximation.

Structured Vector Result

Input:
Cross Product
Ticker:
$CROSS_3(u,v) = (0; 0; 1)$ [LABEL]
Meaning:
The result is a vector, not a single scalar.

Structured Matrix Result

Input:
Inverse of a 2×2 matrix
Ticker:
$A_2^{-1} = [[0.6; -0.7]; [-0.2; 0.4]]$ [LABEL]
Meaning:
The result is a matrix. It should be read as a structured object.

Factorization Result

Input:
factor(120)
Ticker:
$2^3 \times 3 \times 5$ [LABEL]
Meaning:
The useful result is the factorization structure, not merely one scalar number.

Confidence Interval Result

Input:
CI for Mean
Ticker:
CI=[4.82, 6.31] [LABEL]
Meaning:
The result is an interval. The lower and upper limits should be interpreted together.

Regression Result

Input:
Linear Regression
Ticker:
y = 2.5x + 1.2 [LABEL]
Meaning:
The result is an equation describing the fitted line.

Warning Result

Input:
csc(0 rad)

Ticker:

csc(0 rad) = Division by 0 [LABEL]
Meaning:

No valid numeric result was produced.

Practical Reading Rule

When reading a result in CCalc, use this simple rule:

[NUM] means: use it as a normal number.

[APPROX] means: use it as a number, but remember it is approximate.

[LABEL] means: read the full label or ticker before continuing.

This makes result tags a compact but important part of CCalc’s technical language. They help the calculator remain readable even when it is working across arithmetic, geometry, statistics, finance, graphing, number theory, linear algebra, conversion, and other specialized domains.

Appendix E - Converter Categories and Base Units

Overview

The CCalc Converter lets the user take the current calculator value, assign it a unit, and convert it into another unit within the same category.

The converter is category-based. This means that units are grouped by physical or conceptual type. Length units convert to other length units. Mass units convert to other mass units. Torque units convert to other torque units. Currency units convert to other currencies.

Most converter categories use fixed conversion rules. Currency is different because it uses dynamic exchange-rate data. Number Systems are also different because their results may be alphanumeric rather than purely decimal.

Converter Categories

CCalc's converter includes the following categories:

Acceleration	Angle	Angular Velocity
Area	Bytes	Cooking
Currency	Density	Electricity
Energy	EV Efficiency	Force
Fuel Consumption	Length	Lighting
Magnetism	Mass	Moment of Inertia
Number Systems	Power	Pressure
Radiation	Solar Panel Physics	Speed
Temperature	Thermal Performance	Time
Torque	Volume	Weight

Each category contains one or more source units and destination units. The user selects the category first, then chooses the unit to convert from and the unit to convert to.

Base Unit Table

The following table gives the conceptual base unit used by each converter category.

Category	Base Unit
Acceleration	Meter per second squared
Angle	Radian
Angular Velocity	Radians per Second
Area	Square Meter
Bytes	Byte
Cooking	Cups
Currency	Selected base currency
Density	Grams per Cubic Meter
Electricity	Coulomb
Energy	Joule
EV Efficiency	Kilometers per Kilowatt-Hour
Force	Newton
Fuel Consumption	Kilometers per Liter
Length	Meter
Lighting	Lux
Magnetism	Tesla
Mass	Kilogram
Moment of Inertia	Kilogram per Square Meter
Number Systems	Decimal (Base 10)
Power	Watt

Category	Base Unit
Pressure	Pascal
Radiation	Watt per Square Meter
Solar Panel Physics	Solar Irradiance
Speed	Meter per second
Temperature	Kelvin
Thermal Performance	Watt per Square Meter
Time	Second
Torque	Newton per Meter
Volume	Liter
Weight	Kilogram

The base unit is the internal reference point for most conversion rules. A conversion may happen directly or by passing through the category's base unit.

Acceleration

The Acceleration category converts between units that describe change in velocity over time.

Base unit:

Meter per second squared

Typical units include:

Meter per second squared

Foot per second squared

Gal

Inches per second squared

Standard gravity

Use this category for physical acceleration, gravitational acceleration, and related engineering calculations.

Angle

The Angle category converts angular measurements.

Base unit:

Radian

Typical units include:

Radian

Degree

Gradian

Arc minute

Arc second

This category is separate from the calculator's default angle setting. The converter changes a numeric angle from one unit into another. The default angle setting controls how trigonometric and angle-aware tools interpret angle input.

Angular Velocity

The Angular Velocity category converts rotational speed.

Base unit:

Radians per Second

Typical units include:

Radians per Second

Degrees per Second

Revolutions per Minute

Revolutions per Second

Use this category when converting between rad/s, deg/s, RPM, and related rotational-speed units.

Area

The Area category converts surface measurement.

Base unit:

Square Meter

Typical units include:

Square Meter

Square Kilometer

Acre

Hectare

Area conversions are squared-length conversions. They should not be confused with ordinary length conversions.

Bytes

The Bytes category converts digital storage quantities.

Base unit:

Byte

Typical units include:

Byte

Kilobyte

Megabyte

Gigabyte

This category is for data quantity, not data rate. If binary and decimal storage conventions are expanded later, the displayed unit names should clarify which convention is being used.

Cooking

The Cooking category converts common kitchen volume measures.

Base unit:

Cups

Typical units include:

Cups

Teaspoons

Tablespoons

Ounces

Cooking conversions are practical approximations based on predefined factors. They should be interpreted as volume-style cooking units, not as mass conversions.

Currency

The Currency category converts between currencies using exchange-rate data.

Base unit:

Selected base currency

Unlike static unit categories, Currency is dynamic. Rates are loaded from the currency data source and cached. The selected base currency controls the reference currency used for fetching and interpreting rate data.

Typical units include:

USD

EUR

JPY

GBP

The visible list may include more currencies depending on the loaded rate data.

Currency results may use special formatting, such as a fixed number of decimal places, to avoid misleading display precision.

Density

The Density category converts mass-per-volume units.

Base unit:

Grams per Cubic Meter

Typical units include:

Grams per Cubic Meter

Kilograms per Cubic Meter

Density conversions should be interpreted as fixed unit-factor conversions.

Electricity

The Electricity category contains electrical quantity and capacity-style units.

Base unit:

Coulomb

Typical units include:

Coulomb

Ampere-hour

Farad

Because electricity contains different but related electrical concepts, the user should read the unit names carefully. A conversion is meaningful only when the selected units represent compatible quantities in the app’s rule set.

Energy

The Energy category converts work, heat, and energy units.

Base unit:

Joule

Typical units include:

Joule

Kilojoule

Calorie

Kilowatt-hour

Use this category for physical energy, electricity consumption, and heat-style energy quantities.

EV Efficiency

The EV Efficiency category converts electric-vehicle efficiency measures.

Base unit:

Kilometers per Kilowatt-Hour

Typical units include:

Kilometers per Kilowatt-Hour

Miles per Kilowatt-Hour

This is an efficiency category, not an energy category. It describes distance per energy unit.

Force

The Force category converts mechanical force.

Base unit:

Newton

Typical units include:

Newton

Dyne

Pound-force

Use Force for physical force quantities. Do not confuse pound-force with pound as a mass unit.

Fuel Consumption

The Fuel Consumption category converts fuel economy or fuel-use style measures.

Base unit:

Kilometers per Liter

Typical units include:

Kilometers per Liter

Miles per Gallon

Fuel-consumption conversions can be direction-sensitive because some systems express efficiency as distance per fuel, while others may express fuel per distance. Users should confirm that the selected source and destination units express the intended direction.

Length

The Length category converts distance and linear measurement.

Base unit:

Meter

Typical units include:

Meter

Kilometer

Centimeter

Inch

Foot

Length is one of the simplest converter categories. It should be used for one-dimensional distance, height, width, radius, and similar quantities.

Lighting

The Lighting category converts illumination units.

Base unit:

Lux

Typical units include:

Lux

Foot-candle

Use Lighting for illumination, not luminous flux or luminous intensity unless those units are explicitly added later.

Magnetism

The Magnetism category converts magnetic field units.

Base unit:

Tesla

Typical units include:

Tesla

Gauss

Use this category for magnetic flux density conversions.

Mass

The Mass category converts mass units.

Base unit:

Kilogram

Typical units include:

Kilogram

Gram

Pound

Ounce

Mass and Weight are separate categories in CCalc. Mass is the amount of matter. Weight is treated as a practical force/weight-style category depending on the unit set.

Moment of Inertia

The Moment of Inertia category converts rotational inertia units.

Base unit:

Kilogram per Square Meter

Typical units include:

Kilogram per Square Meter

Gram per Square Centimeter

Use this category for rotational mechanics quantities involving mass multiplied by squared distance.

Number Systems

The Number Systems category converts between numeric bases.

Base unit:

Decimal (Base 10)

Typical units include:

Decimal (Base 10)

Binary (Base 2)

Hexadecimal (Base 16)

Octal (Base 8)

Base 12 (Duodecimal)

Number Systems are special because outputs may be strings rather than ordinary decimal numbers. For example, converting decimal 123 to hexadecimal produces:

7B

This kind of result may require string display behavior rather than normal numeric display behavior.

Power

The Power category converts rate-of-work or rate-of-energy units.

Base unit:

Watt

Typical units include:

Watt

Kilowatt

Horsepower

Power should not be confused with energy. A watt is energy per unit time.

Pressure

The Pressure category converts force-per-area units.

Base unit:

Pascal

Typical units include:

Pascal

Bar

Atmosphere

Use this category for pressure in physics, engineering, and atmospheric contexts.

Radiation

The Radiation category contains radiation-related units.

Base unit:

Watt per Square Meter

Typical units include:

Watt per Square Meter

Gray

Sievert

Radiation units can represent different physical ideas, such as absorbed dose, equivalent dose, and irradiance. Users should confirm that the selected conversion is meaningful for the quantity being handled.

Solar Panel Physics

The Solar Panel Physics category contains solar-energy and solar-resource quantities.

Base unit:

Solar Irradiance

Typical units include:

Solar Irradiance

Sun Hours

This category is meant for solar-panel related calculations and should be interpreted according to the app’s specific rule definitions.

Speed

The Speed category converts distance per time.

Base unit:

Meter per second

Typical units include:

Meter per second

Kilometer per hour

Miles per hour

Use this category for linear velocity or speed.

Temperature

The Temperature category converts between temperature scales.

Base unit:

Kelvin

Typical units include:

Celsius

Fahrenheit

Kelvin

Temperature is special because it cannot always be converted by simple multiplication. Celsius, Fahrenheit, and Kelvin require offset formulas.

For example, converting Celsius to Kelvin requires adding an offset, not only applying a factor.

Thermal Performance

The Thermal Performance category converts heat-transfer performance quantities.

Base unit:

Watt per Square Meter

Typical units include:

Watt per Square Meter

BTU per Hour per Square Foot

Use this category for heat-transfer or thermal-output style quantities.

Time

The Time category converts time intervals.

Base unit:

Second

Typical units include:

Second

Minute

Hour

Day

Use this category for duration, not clock time or calendar dates.

Torque

The Torque category converts rotational force quantities.

Base unit:

Newton per Meter

Typical units include torque-style units such as newton meter and pound-force foot, depending on the available rule set.

Torque should not be confused with energy, even when some unit dimensions may look similar. Torque represents rotational moment.

Volume

The Volume category converts three-dimensional capacity or space.

Base unit:

Liter

Typical units include:

Liter

Milliliter

Cubic Meter

Gallon

Use Volume for liquid volume, capacity, and three-dimensional measurement.

Weight

The Weight category converts practical weight-style units.

Base unit:

Kilogram

Typical units include:

Kilogram

Gram

Pound

Ounce

In everyday use, weight and mass units are often used interchangeably. In strict physics, weight is a force. CCalc keeps Weight as a practical category for user convenience.

Conversion Rule Behavior

Most converter categories use predefined conversion rules.

A rule defines how to convert from one unit to another. In many categories, the conversion can be understood as:

source unit → base unit → destination unit

For example:

inch → meter → centimeter

or:

mile per hour → meter per second → kilometer per hour

The exact implementation may use direct rules or chained rules, but the user-facing idea is the same: units are converted within their own category through known factors or formulas.

A conversion is valid only when:

- the category is correct
- the source unit exists in that category
- the destination unit exists in that category
- the app has a rule path between the two units
- the input value is valid for that conversion

If no valid rule exists, the converter should report an invalid or unsupported conversion rather than guessing.

Dynamic Currency Behavior

Currency conversions behave differently from static unit conversions.

Currency uses exchange-rate data. The selected base currency controls the rate table. CCalc loads rates for the selected base and caches them to avoid unnecessary repeated network requests.

Important currency behaviors:

- rates are dynamic, not fixed constants
- the base currency is user-selectable
- cached rates may be reused until they expire
- currency results may use currency-specific decimal formatting
- historical currency data may be used for trend or fluctuation features
- if data is unavailable, cached fallback behavior may be used when valid

The base currency setting does not mean that the user can only convert from that currency. It means that exchange-rate data is organized around that selected reference currency.

For example, if the base is USD, CCalc can still support conversions such as:

EUR → GBP

BRL → CAD

JPY → CHF

as long as the necessary rates are available.

Number-System Conversion Behavior

Number-system conversions are special because they may produce non-decimal text.

Examples:

Decimal 123 → Hexadecimal 7B

Decimal 10 → Binary 1010

Decimal 64 → Octal 100

These results may not fit into a normal decimal operand field.

For this reason, Number Systems may use a string result for the destination display while still preserving calculator logic where possible.

Important behavior:

- decimal input may produce alphanumeric output
- hexadecimal output may contain A–F
- binary output contains only 0 and 1
- octal output contains digits 0–7
- base-12 output may require special digit representation depending on implementation
- the displayed result may not always be a standard Decimal value

The user should read number-system results as representation changes, not as physical unit conversions.

Technical Limitations

The Converter is intentionally structured and category-based. It is not a general symbolic dimensional-analysis engine.

Important limitations:

- conversions are limited to supported categories and supported units
- unsupported units cannot be inferred automatically
- inter-category conversion is not generally supported
- temperature conversions require formulas, not simple factors
- currency data depends on availability of exchange-rate data
- cached currency data may be temporarily reused
- number-system results may require string display handling
- some categories contain conceptual units that require careful interpretation
- unit names must be read literally to avoid confusing related quantities

Examples of conversions that should not be assumed unless explicitly supported:

mass → force

energy → torque

irradiance → dose

currency → commodity value

temperature difference → absolute temperature

CCalc’s converter is designed to be predictable. It performs conversions that are defined in its rule set and avoids guessing when the dimensional meaning is uncertain.

Appendix F - Grapher Function Support and Limitations

Overview

The CCalc Grapher is a full-screen graphing module for plotting mathematical functions, tracing values, reading slopes, analyzing visible behavior, and sharing graph output.

The Grapher is designed for calculator-style function exploration. It supports common mathematical operators, standard functions, constants, the variable x, the calculator-supplied parameter a, and the current ANS value.

It is not intended to be a full symbolic algebra system. It evaluates functions numerically over a selected visible range and then builds a plotted curve from sampled points.

Because it is numerical, the Grapher has specific support rules and limitations. This appendix explains those rules.

Supported Operators

The Grapher supports standard arithmetic operators.

Typical supported operators include:

- **+**
- **−**
- **×**
- **÷**
- **^**

These represent addition, subtraction, multiplication, division, and exponentiation.

The Grapher also supports parentheses:

- **(**
- **)**

Parentheses control grouping and override normal arithmetic precedence.

Example:

2 + 3 × 4

is interpreted as:

2 + (3 × 4)

while:

(2 + 3) × 4

forces the addition first.

Exponentiation binds more tightly than multiplication and division, and multiplication and division bind more tightly than addition and subtraction, unless parentheses change the grouping.

Supported Functions

The Grapher supports common mathematical functions used in graphing and numerical exploration.

Typical supported functions include:

- **sin(x)**
- **cos(x)**
- **tan(x)**
- **sqrt(x)**
- **ln(x)**
- **log10(x)**
- **abs(x)**

Depending on the current implementation, additional functions may also be present in the token tray or expression evaluator.

Function tokens are entered through the Grapher’s token system. Some function tokens automatically insert parentheses or place the cursor inside the function call.

Examples:

sin(x)

sqrt(x)

ln(x)

abs(x)

The Grapher evaluates these functions numerically for different values of x.

Supported Constants

The Grapher supports mathematical constants.

Typical supported constants include:

π

e

These constants can be inserted from the token tray.

π represents pi.

e represents Euler's number.

Constants may be combined with operators, functions, x, a, and ANS.

Examples:

sin(πx)

e^x

ln(x + e)

Supported Variables

The primary graphing variable is:

x

The Grapher plots functions of x.

Example:

sin(x)

means that the Grapher evaluates sin(x) repeatedly across the visible x-range and draws the resulting curve.

The Grapher may also support parameter-style values, especially:

a

ANS

These are not independent graph axes. They are values substituted into the expression while x remains the graphing variable.

Parameter a

The parameter a is a calculator-linked parameter.

In CCalc, a represents the current calculator operand passed into the Grapher as a reusable value.

Example:

If the current calculator operand is:

2.5

then in the Grapher:

sin(a × x)

is interpreted as:

sin(2.5 × x)

The parameter a is useful for exploring how the current calculator value affects a graph.

Examples:

a × x

sin(a × x)

x^2 + a

1 / (x − a)

When Compare-a behavior is available, CCalc can use nearby values of a to show how the curve changes when the parameter is nudged.

The important point is that a is not typed as an unknown variable to solve for. It is a numeric parameter supplied by the calculator.

ANS in Graphing

ANS represents the calculator's stored previous answer.

If ANS is set, the Grapher can use it as a constant value inside the expression.

Example:

If ANS is:

17

then:

x + ANS

is interpreted as:

x + 17

ANS is useful when the user wants to graph a function that depends on a result previously calculated in CCalc.

ANS is not a second graphing variable. It is substituted as a numeric value.

Domain Restrictions

Many functions have restricted domains.

Examples:

sqrt(x)

requires:

x ≥ 0

ln(x)

requires:

x > 0

1 / x

is undefined at:

x = 0

tan(x)

is undefined at odd multiples of:

π / 2

When a sampled x-value falls outside the valid domain, the Grapher should not draw a normal point there.

Instead, the Grapher treats the value as invalid or missing and breaks the curve as needed.

Discontinuities

A discontinuity occurs when a function cannot be drawn continuously across part of the visible range.

Examples:

1 / x

tan(x)

1 / (x − 2)

sqrt(x)

ln(x)

The Grapher detects many discontinuities numerically by checking for invalid, infinite, non-finite, or sharply unstable values.

When the function becomes invalid, the plotted curve should break rather than connecting unrelated points.

Example:

1 / x

should not draw a continuous line through x = 0.

Gaps

A gap is a visible or detected break in the plotted curve.

Gaps may come from:

- domain restrictions
- division by zero
- discontinuities
- invalid function values
- vertical asymptotes
- sampling breaks
- graph-boundary behavior

The Analysis panel may report gaps when it detects one or more discontinuous regions in the visible graph.

A gap count tells the user how many break regions were detected.

A gap range gives more detail about where a gap occurs.

Example:

Gaps: 1

Gap: [-1.9656, 0.0018]

These are not conflicting. The first item counts the detected gaps. The second item describes the approximate location of one detected gap.

Asymptotes

A vertical asymptote occurs when a function grows without bound near a specific x-value.

Examples:

1 / x

1 / (x − 3)

tan(x)

The Grapher may identify possible vertical asymptotes through numerical behavior, such as very large values, sign changes, invalid values, or steep growth near a break.

Asymptote detection is numerical and should be read as approximate.

The Grapher should not be expected to prove asymptotes symbolically. It identifies likely visible asymptotic behavior within the current graphing range.

Sampling Behavior

The Grapher draws curves by sampling the function across the visible x-range.

This means it evaluates the function at many x-values between the selected x-minimum and x-maximum.

Each valid evaluation produces a plotted point.

Invalid evaluations produce breaks.

The final visible curve is built from these points.

Sampling is affected by:

- the x-range
- the y-range
- the number of sample points
- function steepness
- discontinuities
- domain boundaries
- zoom level
- visible canvas size

Because the graph is sampled, very narrow features may be missed if they occur between sampled points.

Examples of narrow features include:

- very sharp spikes
- very narrow roots
- rapid oscillations
- thin removable gaps

- abrupt local behavior

Zooming in or changing the range may reveal details that were not visible at the previous scale.

Boundary Refinement

Boundary refinement improves how the Grapher handles functions whose domains begin or end at a boundary.

Example:

sqrt(x)

has a domain boundary at:

x = 0

A simple sampler might miss the exact boundary and begin drawing slightly after zero. Boundary refinement helps locate a better visible starting point.

This makes graphs such as:

sqrt(x)

sqrt(x − 2)

ln(x)

look more accurate near their valid domain edges.

Boundary refinement is still numerical. It improves the graph, but it is not a symbolic domain solver.

Trace Behavior

Trace lets the user inspect a point on the graph.

When tracing, CCalc reports the current x-position and the corresponding y-value.

Example:

x = 2.5

y = 0.59847214

Trace values depend on the evaluated function and the current graph range.

The user may send the traced y-value back to the main calculator, where it becomes available as a calculator value.

Trace is especially useful for:

- reading approximate roots
- checking function values
- exploring peaks and valleys
- inspecting discontinuities
- comparing nearby curve behavior
- sending graph-derived values back into CCalc

Trace values are numerical and should generally be treated as approximate.

Slope Calculation

The Grapher can estimate slope at the traced point.

Slope represents the local rate of change of the function near the traced x-value.

Example:

slope ≈ 2.75

Slope is calculated numerically, not symbolically.

This means it is an estimate based on nearby function values. It may be less reliable near:

- discontinuities
- sharp corners
- cusps
- gaps
- endpoints
- vertical asymptotes
- very steep regions
- noisy numerical behavior

For smooth functions, the slope estimate is usually useful as a practical derivative reading.

Analysis Panel Rules

The Analysis panel summarizes visible graph behavior.

Typical analysis items include:

- visible minimum
- visible maximum
- roots
- local minima
- local maxima
- domain gaps
- vertical asymptotes
- visible y-range
- gap details
- Compare-a information, when enabled

The key word is visible.

The Analysis panel analyzes the function over the currently visible graphing range. It does not necessarily describe the function globally.

For example, a function may have more roots outside the visible x-range. The Analysis panel only reports roots it detects in the visible range.

Similarly, minima and maxima are visible-range observations, not always absolute global extrema.

Roots

A root is an x-value where:

f(x) = 0

The Grapher detects roots numerically.

Roots may be detected through sign changes, refined crossings, or near-zero behavior.

Root detection may be limited when:

- the function only touches the x-axis without crossing
- the root is very narrow
- the root occurs near a discontinuity
- the sampling resolution misses the crossing
- the visible range is too broad
- the function is nearly flat near zero

Root values should be read as approximate.

Visible Minimum and Maximum

The visible minimum and maximum are the smallest and largest y-values detected within the visible plotted range.

They are not necessarily global extrema.

Example:

A sine wave has infinitely many maxima and minima globally, but the Analysis panel reports only what is visible in the current range.

Changing the range can change the reported minimum and maximum.

Local Minima and Maxima

Local minima and maxima are detected by changes in curve direction.

A local maximum is a visible peak.

A local minimum is a visible valley.

These detections are numerical and sampling-based. They may be missed or misidentified if the curve is very sharp, noisy, discontinuous, or undersampled.

Sharing Behavior

The Grapher supports sharing graph output.

Sharing may include:

- graph image
- compact caption
- trace data
- slope data
- analysis summary
- text report

A shared graph image captures the visual graph state.

A shared text report captures the function, range, trace information, slope, and analysis details where available.

If Compare-a is active, shared output may include relevant Compare-a information depending on implementation.

Sharing is intended to make graph results easier to document, send, or store outside the app.

Known Limitations

The Grapher is numerical, not symbolic.

Important limitations include:

- it does not prove algebraic identities
- it does not solve functions symbolically
- it does not guarantee all roots are found
- it may miss very narrow features
- it may approximate discontinuities
- it may approximate asymptotes
- slope is estimated numerically
- analysis is limited to the visible range
- rapid oscillations may be undersampled
- removable discontinuities may be difficult to distinguish from gaps
- some complex-valued behavior is not graphed
- invalid domains produce breaks rather than symbolic explanations
- parameter a is numeric, not an unknown variable
- ANS is numeric, not a symbolic expression

Examples of functions that may require careful interpretation:

sin(100x)

1 / (x − 0.0001)

sqrt(x − a)

ln(x)

tan(x)

abs(x)

1 / sin(x)

These functions may have sharp behavior, restricted domains, repeated discontinuities, or sampling-sensitive features.

The practical rule is:

Use the Grapher as a visual and numerical exploration tool. For exact symbolic conclusions, verify with algebraic reasoning or a dedicated symbolic tool.

Practical Reading Rule

When using the Grapher, read every result in relation to the current graph state.

The plotted curve depends on:

- the expression
- the visible x-range
- the visible y-range
- the selected angle behavior where relevant
- the current value of a

- the current ANS value
- the sampling resolution
- the zoom level
- the trace position

A graph result is therefore best understood as:

the numerical behavior of this expression over this visible range, not as a complete symbolic description of the function everywhere.

Appendix G — Function Reference and Formula Catalogue

Overview

Appendix G is the technical formula catalogue for CCalc's docklets. It is intended for users who want to understand what each docklet function calculates, what inputs it expects, what output it returns, and whether the result is numeric, approximate, label-only, exact-style, or invalid. The full appendix is organized alphabetically by docklet name. Each docklet section describes the formulas, parameters, assumptions, units, result behavior, approximation rules, and error conditions for that docklet. Appendix G does not replace the ordinary user guide. The user guide explains how to operate CCalc. Appendix G explains what the docklet functions mean. The docklet sections are arranged in this order:

1. Calculus
2. Combinatorics
3. Complex
4. Cryptography
5. Engineering and applied math
6. Equation-solving
7. Financial
8. Geometry
9. Linear algebra
10. Number theory
11. Statistics
12. Trigonometry

Each docklet-specific section should be read as a technical reference for that docklet.

Purpose of the Function Reference

The purpose of the function reference is to make CCalc's specialized functions transparent. Many CCalc docklets perform calculations that are more specialized than ordinary arithmetic. Some functions return a single number. Others return a structured result, such as a vector, matrix, interval, factorization, regression equation, confidence interval, complex value, diagnostic label, or formula-style output. Appendix G explains:

- what each function computes
- what formula or method is used
- what each input field means
- which inputs are required
- which inputs may default to the current operand
- what the output represents
- whether the output is scalar or structured
- whether the result may continue into ordinary calculation
- whether the result is exact-style or approximate
- which errors or warnings may appear

For example: NPV, det(A), P□, pow$_{m}$ These labels are intentionally short inside the docklet interface. Appendix G gives their expanded meaning, input requirements, result behavior, and limitations.

How the Catalogue Is Organized

Appendix G uses one section per docklet. Each docklet section should begin with its docklet name and then describe the functions exposed by that docklet. The sections are placed alphabetically by docklet name, not by difficulty or by app layout. The insertion order is:

Appendix G.1 — Calculus

Appendix G.2 — Combinatorics

Appendix G.3 — Complex

Appendix G.4 — Cryptography

Appendix G.5 — Engineering and Applied Math

Appendix G.6 — Equation-Solving

Appendix G.7 — Financial

Appendix G.8 — Geometry

Appendix G.9 — Linear Algebra

Appendix G.10 — Number Theory

Appendix G.11 — Statistics

Appendix G.12 — Trigonometry

Each of these sections may also be maintained as a separate working file while the manual is being edited. In the final manual, they belong inside Appendix G in the order shown above.

Docklet Technical Guides

Appendix G.1 — Calculus

Overview

The Calculus docklet provides numerical calculus tools for functions of one variable, functions of two variables, convolution-style calculations, Fourier-style evaluation over a finite interval, and basic numerical differential-equation solving. It is intended for users who need fast operational results from predefined function templates rather than symbolic algebra. The docklet does not accept arbitrary typed formulas. Instead, the user selects a calculation and chooses a function template, then enters the coefficients, bounds, points, step sizes, or model parameters required by that calculation. The Calculus docklet covers:

- limits and numerical derivatives
- tangent lines and linearization
- definite, absolute, improper, arc-length, volume, convolution, and Fourier integrals
- root, minimum, and maximum searches
- Taylor approximations up to third order
- partial derivatives, gradients, Hessians, directional derivatives, critical points, and rectangular double integrals
- first-order and second-order numerical differential-equation models

Most numerical results are approximate and should be interpreted according to the chosen template, interval, step size, and numerical method.

Opening and using the docklet

The Calculus docklet appears as a parked/expandable panel. When parked, only the compact docklet header is visible. Tapping or dragging the header expands the list of functions. Selecting a function opens a firm prompt dialog. Each function row shows a short symbol and a full label. For example:

• d/dx — d/dx at x_0

• ∫ — ∫ a→b

• ∇f — Gradient ∇f at P

• MSD — Mass-spring-damper

Most functions require the user to choose or accept a function template, then enter the numerical parameters for that template and for the requested operation. The prompt dialog contains:

- a title showing the selected symbol and label
- one or more template menus, where applicable
- numeric input fields

- a Compute button
- a Cancel button

The dialog is firm. Tapping outside the prompt does not dismiss it; it gives a warning. Use Cancel to close the prompt without computing.

Current operand behavior

The Calculus docklet does not use the current calculator operand as the function input, bound, coefficient, or default value. All required fields must be entered in the prompt dialog. Some fields are prefilled with default numerical values, such as: • h = 0.00001 for many numerical derivative and limit-style calculations • dt = 0.001 for differential-equation solvers • sign = 1 for positive-infinity direction fields These defaults come from the docklet, not from the current calculator operand. When a scalar numerical result is produced, it becomes the active calculator value and can continue in calculator expressions. When the result is a formula, vector, matrix, complex-like Fourier value, or descriptive label, it is shown as a terminal result and should be read as output text rather than used as an ordinary scalar expression result.

Prompt dialogs

Required fields All visible fields in the prompt must contain valid numeric input before Compute becomes active. A blank visible field keeps Compute disabled. The only practical exception is not a blank field but a prefilled default field: if the docklet supplies h, dt, or sign, that value is already present and may be accepted as-is. Optional fields Some fields are labeled as optional in the placeholder, such as h for step size. In practice, the prompt supplies a default value, so the field is not blank. The user may leave the default value unchanged. Blank-field behavior Blank fields do not fall back to the current operand. Blank fields prevent Compute from running. Mode fields and choice menus The Calculus docklet uses several menus. For single-variable functions, the Function template menu provides: Polynomial Power Exponential Log Sin Cos Damped Sin Rational Constant For multivariable functions, the Function template menu provides: • Constant • Plane • Quadratic 2D For convolution, two template menus are shown: • f(τ) template • g(x) template Each may be: • Constant • Rectangular Pulse • Causal Exponential For differential equations, the Forcing template menu provides: None Step Ramp Sinusoid Pulse Changing a menu choice rebuilds the prompt fields for that choice. Compute button behavior Compute becomes active only when all required fields contain valid numbers. Pressing Compute evaluates the selected numerical method and returns either a scalar result or a displayed non-scalar result. If the method fails because of invalid input, a domain restriction, a non-finite function value, a divergent integral, an invalid direction, or an unsupported interval, the prompt remains open and an error message is shown. Cancel behavior Cancel closes the prompt without changing the calculator value. The selected function may be visually marked as cancelled. Outside-tap behavior The prompt is firm. Tapping outside the prompt does not dismiss it. Use Cancel. Keyboard behavior The prompt uses numeric keyboard entry. The input sanitizer accepts ordinary decimal notation and scientific notation. It also normalizes common minus signs and decimal separators.

Decimal places and rounding

Numerical results are rounded according to the app's decimalPlaces setting. The setting is clamped to the range 0 through 18. Very small negative zero artifacts are normalized to 0. Several numerical routines also snap values close to integers when the result is within a tolerance. This affects display clarity for results such as roots, extrema locations, partial derivatives, Fourier components, differential-equation outputs, gradients, and Hessian entries. For example, a value numerically computed as 1.00000000003 may be shown as 1. Some compact labels, especially differential-equation state summaries, may use a shorter display format for readability.

Locale and separators

The Calculus docklet follows the app's selected numeric locale. In dot-decimal locales: • decimal separator: . • argument separator in ticker-style summaries: , In comma-decimal locales: • decimal separator: , • argument separator in ticker-style summaries: ; For example, a function call that would appear as: ∫(0,1) may appear with semicolon-separated arguments in comma-decimal mode: ∫(0;1) Input fields accept both . and , as decimal separators and normalize them internally. The Calculus docklet does not use list-entry fields in this version.

Angle-unit behavior

The Calculus docklet does not read the app's global angle-unit setting. Trigonometric templates use radians for their angular expressions. In the Sin, Cos, and Damped Sin templates: • ω is an angular frequency or coefficient • φ is a phase offset • expressions such as sin(ωx+φ) and cos(ωx+φ) are evaluated using radians The Fourier tool also uses angular frequency ω.

Result behavior

The docklet produces two broad result types. Scalar numerical results become the current calculator value. These can continue in normal calculator expressions. Examples: derivative value, integral value, root value • minimum or maximum function value • partial derivative value • differential-equation final value Terminal label results are shown as text or structured output. These are intended to be read, not reused as ordinary scalar values. Examples: tangent-line equation, Taylor polynomial linearization formula critical-point classification gradient vector display, Hessian matrix display, Fourier complex result display Some functions return a scalar main value while also displaying a richer label. For example: • Fourier returns the magnitude as the scalar result, while showing the complex real/imaginary result as text. • Gradient returns the vector norm as the scalar result, while showing the gradient vector. • Hessian returns the Frobenius norm as the scalar result, while showing the matrix.

Result tags

The docklet uses result tags to indicate the nature of the output. [NUM] means a numeric result. [APPROX] means a numerical approximation. Most Calculus results use this tag because they are produced by finite differences, quadrature, scanning, bisection, or numerical integration. [LABEL] means a descriptive result, symbolic-style expression, vector, matrix, classification, or other non-scalar output. In this docklet, most scalar calculus computations are approximate. A result tagged [APPROX] should be interpreted as a numerical estimate, not an exact symbolic result.

Function groups

The Calculus docklet is organized into these groups:

Group 1 — Limits: lim, lim−, lim+, lim∞

Group 2 — Derivatives: d/dx, d^2, d^3, tan

Group 3 — Integrals: ∫, ∫|f|, ∫∞, arc, vol, conv, FT

Group 4 — Roots & Extrema: root, min, max

Group 5 — Series / Approximation: TAY, LIN

Group 6 — Partials: ∂x, ∂y, ∂xx, ∂yy, ∂xy

Group 7 — Multivariable: ∇f, D_u, crit, H, ∬

Group 8 — Differential Equations: 1stLIN, COOL, 2ndSTD, MSD

Function templates

Single-variable templates

These templates are used by limits, derivatives, most integrals, roots/extrema, Taylor, and linearization. Polynomial

Formula: $f(x)=a_0+a_1x+a_2x^2+a_3x^3$

Fields: a_0: constant term a_1: coefficient of x a_2: coefficient of x^2 a_3: coefficient of x^3

Example: $a_0=1$, $a_1=2$, $a_2=0$, $a_3=0$ gives $f(x)=1+2x$. Power

Formula: $f(x)=A\cdot x^n$

Fields: • A: coefficient • n: power Restriction: for negative x, non-integer powers are not supported and produce non-finite values.

Example: A=3, n=2 gives $f(x)=3x^2$. Exponential

Formula: f(x)=A·e^(kx)

Fields: • A: amplitude • k: rate

Example: A=2, k=0.5 gives f(x)=2e^(0.5x). Log

Formula: f(x)=A·ln(x)

Fields: • A: coefficient Restriction: x must be greater than 0.

Example: A=1 gives f(x)=ln(x). Sin

Formula: f(x)=A·sin(ωx+φ)

Fields: • A: amplitude • ω: angular frequency • φ: phase Angles are in radians. Cos

Formula: f(x)=A·cos(ωx+φ)

Fields: • A: amplitude • ω: angular frequency • φ: phase Angles are in radians. Damped Sin

Formula: f(x)=A·e^(−σx)·sin(ωx+φ)

Fields: A: amplitude, σ: damping, ω: angular frequency, φ: phase Angles are in radians. Rational

Formula: f(x)=(a0+a1x)/(b0+b1x)

Fields: a0: numerator constant, a1: numerator x coefficient, b0: denominator constant b1: denominator x coefficient Restriction: points where b0+b1x is effectively zero are not valid. Constant

Formula: f(x)=K

Fields: • K: constant value

Multivariable templates

These templates are used by partial derivatives, gradient, directional derivative, critical point, Hessian, and double integral. Constant

Formula: f(x,y)=K

Fields: • K: constant value Plane

Formula: f(x,y)=a0+ax·x+ay·y

Fields: • a0: constant term • ax: x coefficient • ay: y coefficient Quadratic 2D

Formula: $f(x,y)=a0+ax\cdot x+ay\cdot y+axx\cdot x^2+axy\cdot xy+ayy\cdot y^2$

Fields: a0: constant term, ax: x coefficient, ay: y coefficient, axx: x^2 coefficient axy: xy coefficient, ayy: y^2 coefficient This is the default multivariable template.

Convolution templates

The convolution tool computes a numerical integral of the form: $\int f(\tau)\cdot g(t-\tau)\,d\tau$ over a user-specified τ interval. Each side has its own template. Constant

Formula: f(x)=K or g(x)=K

Field: • K: constant value Rectangular Pulse

Formula: value equals A inside [L,R], otherwise 0.

Fields: • A: amplitude • L: left edge • R: right edge Restriction: L ≤ R. Causal Exponential

Formula: A·e^(−kx) for x ≥ 0, otherwise 0.

Fields: • A: amplitude • k: decay rate Restriction: k > 0.

Forcing templates for differential equations

Differential-equation tools can include an external input u(t). None u(t)=0 No extra fields. Step u(t)=A for t ≥ t0, otherwise 0.

Fields: • A: step amplitude • t0: step start time Ramp u(t)=A(t−t0) for t ≥ t0, otherwise 0.

Fields: • A: ramp slope • t0: ramp start time Sinusoid u(t)=A·sin(ωt+φ)

Fields: • A: amplitude • ω: angular frequency • φ: phase Pulse u(t)=A within a pulse centered at t0 with width w, otherwise 0.

Fields: • A: pulse amplitude • w: pulse width • t0: pulse center time Restriction: w > 0.

Function reference

Limits

lim — Limit x→a

Purpose: Estimates the two-sided limit of the selected single-variable template as x approaches a point.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the approach point. h is shown but the limit estimator uses an internal decreasing sequence of h values.

Result: Approximate scalar limit value.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: The function must remain finite near the sampled approach points. A jump discontinuity or oscillatory behavior may produce an unreliable estimate.

Example: Polynomial $f(x)=x^2$, using $a_0=0$, $a_1=0$, $a_2=1$, $a_3=0$, with $x_0=2$ gives approximately 4.

lim− — Limit x→a−

Purpose: Estimates the left-hand limit as x approaches a point from below.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the approach point.

Result: Approximate scalar left-limit value.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Useful when the left and right limits may differ. The sampled values must remain finite.

Example: For a rational template with a denominator that changes near a point, use lim− to inspect behavior from the left side only.

lim+ — Limit x→a+

Purpose: Estimates the right-hand limit as x approaches a point from above.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the approach point.

Result: Approximate scalar right-limit value.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Useful when the function is one-sided or discontinuous at the approach point.

Example: For f(x)=ln(x), use lim+ at a positive point close to zero. At or below invalid log-domain points, evaluation fails.

lim∞ — Limit ±∞

Purpose: Estimates the behavior of a single-variable template as x moves toward positive or negative infinity.

Inputs: Function template fields, sign.

Parameters: sign=1 means positive infinity. sign=-1 means negative infinity. Any nonnegative sign is treated as positive; negative sign is treated as negative.

Result: Approximate scalar value based on large sampled x values.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: This is a finite sampling estimate, not a symbolic limit. Divergent, oscillatory, or slowly converging functions may not be represented reliably.

Example: For f(x)=1/x using the Rational template, a large-x estimate approaches 0.

Derivatives

d/dx — d/dx at x_0

Purpose: Computes the first derivative numerically at a point.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the evaluation point. h is the finite-difference step. Default h is 0.00001.

Result: Approximate first derivative.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses a central finite difference. Too large an h can blur local behavior; too small an h can increase cancellation error.

Example: For $f(x)=x^2$, at $x_0=3$, the result is approximately 6.

d² — d²/dx² at x₀

Purpose: Computes the second derivative numerically at a point.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the evaluation point. h is the finite-difference step.

Result: Approximate second derivative.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses a central second-difference stencil. Internally, h is not allowed below a small floor.

Example: For $f(x)=x^2$, at $x_0=3$, the result is approximately 2.

d³ — d³/dx³ at x₀

Purpose: Computes the third derivative numerically at a point.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the evaluation point. h is the finite-difference step.

Result: Approximate third derivative.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Third derivatives are more sensitive to numerical cancellation. The method uses a larger minimum h than the first and second derivative tools.

Example: For $f(x)=x^3$, at $x_0=1$, the result is approximately 6.

tan — Tangent line

Purpose: Builds a tangent-line approximation at a point.

Inputs: Function template fields, x_0, h.

Parameters: x_0 is the point of tangency. h is used for the numerical slope.

Result: A text formula of the form: $y \approx f(x_0)+f'(x_0)(x-x_0)$

Result behavior: Terminal label. The result is shown as a line equation rather than as a scalar value.

Notes / restrictions: Requires both $f(x_0)$ and the numerical derivative to be finite.

Example: For $f(x)=x^2$ at $x_0=2$, the tangent line is approximately: $y \approx 4+4(x-2)$

Integrals

∫ — ∫ a→b

Purpose: Computes a definite integral over a finite interval.

Inputs: Function template fields, a, b.

Parameters: a is the left endpoint. b is the right endpoint.

Result: Approximate signed integral.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses Simpson integration. If a > b, the result changes sign. If the function is non-finite anywhere sampled in the interval, the calculation fails.

Example: For f(x)=x, from a=0 to b=1, the result is approximately 0.5.

∫|f| — Area

Purpose: Computes the integral of the absolute value of the function over a finite interval.

Inputs: Function template fields, a, b.

Parameters: a and b define the interval.

Result: Approximate area under |f(x)|.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Unlike ∫, negative portions contribute positively.

Example: For f(x)=x, from a=-1 to b=1, the signed integral is 0, but the area is approximately 1.

∫∞ — Improper Integral

Purpose: Estimates a one-sided improper integral from a finite anchor toward positive or negative infinity.

Inputs: Function template fields, anchor, sign.

Parameters: anchor is the finite endpoint. sign=1 integrates toward +∞; sign=-1 integrates toward −∞.

Result: Approximate improper integral.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: The method uses a transformed integration plus tail checks. Divergent, unstable, or non-finite cases fail with an error.

Example: For a decaying exponential template, an integral from 0 to +∞ can be estimated. For a constant nonzero function, the improper integral diverges and should fail.

arc — Arc Length

Purpose: Computes the approximate arc length of a single-variable function over a finite interval.

Inputs: Function template fields, a, b.

Parameters: a and b define the interval.

Result: Approximate arc length.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses numerical derivative estimates inside Simpson integration. The function and its derivative estimate must remain finite over the sampled interval.

Example: For f(x)=0, from a=0 to b=3, arc length is approximately 3.

vol — Volume of Revolution

Purpose: Computes the volume generated by revolving f(x) around the x-axis over a finite interval.

Inputs: Function template fields, a, b.

Parameters: a and b define the interval.

Result: Approximate volume.

Formula: $\pi\int[f(x)]^2\,dx$

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: This is specifically x-axis revolution using the disk method.

Example: For f(x)=1 from 0 to 2, volume is approximately 2π.

conv — Convolution at t

Purpose: Computes a numerical convolution value at a specified t.

Inputs: t, τa, τb, f-template fields, g-template fields.

Parameters: t is the evaluation point. τa and τb define the integration interval for τ.

Result: Approximate scalar convolution value.

Formula: $\int[\tau a\to\tau b]\ f(\tau)\cdot g(t-\tau)d\tau$

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Both selected convolution templates must be valid. Rectangular-pulse templates require left edge ≤ right edge. Causal exponential templates require positive decay rate.

Example: Use two rectangular pulses to estimate overlap-like convolution behavior at a chosen t.

FT — Fourier at ω

Purpose: Computes a finite-interval Fourier-style value at angular frequency ω.

Inputs: Function template fields, ω, a, b.

Parameters: ω is angular frequency. a and b define the finite integration interval.

Result: The scalar result is the magnitude of the complex Fourier value. The displayed label shows the complex value as real ± imaginary i.

Formula: Real part: $\int f(x)\cos(\omega x)\,dx$ Imaginary part: $-\int f(x)\sin(\omega x)\,dx$

Result behavior: Returns the magnitude as a scalar approximate result and displays the complex result as label text.

Notes / restrictions: This is a finite-interval numerical evaluation, not a full symbolic Fourier transform.

Example: For a constant function over [0,1], ω=0 gives a purely real value equal to the integral of the constant over the interval.

Roots & Extrema

root — Solve f(x)=0

Purpose: Finds a root of the selected function on an interval.

Inputs: Function template fields, a, b.

Parameters: a and b bracket the search interval.

Result: Approximate root x-value.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Requires finite endpoint values and f(a)·f(b) ≤ 0. Uses bisection, with a maximum iteration count.

Example: For f(x)=x−2, use a polynomial with a_0=-2, a_1=1, a_2=0, a_3=0, interval [0,4]. Result is approximately 2.

min — Min [a,b]

Purpose: Searches for an approximate minimum value on a finite interval.

Inputs: Function template fields, a, b.

Parameters: a and b define the search interval.

Result: Minimum function value found. The result label includes the approximate x-location.
Result behavior: Flows back into the calculator as a scalar approximate result.
Notes / restrictions: Uses scanning followed by refinement. It is numerical and may miss narrow features if the function changes sharply between scan points.
Example: For $f(x)=x^2$ on [-2,3], the minimum value is approximately 0, near $x\approx0$.

max — Max [a,b]

Purpose: Searches for an approximate maximum value on a finite interval.
Inputs: Function template fields, a, b.
Parameters: a and b define the search interval.
Result: Maximum function value found. The result label includes the approximate x-location.
Result behavior: Flows back into the calculator as a scalar approximate result.
Notes / restrictions: Uses scanning followed by refinement. Like min, it is a numerical search over sampled behavior.
Example: For $f(x)=-x^2$ on [-2,3], the maximum value is approximately 0, near $x\approx0$.

Series / Approximation

TAY — Taylor (x_0,n)

Purpose: Builds a Taylor polynomial centered at x_0.
Inputs: Function template fields, x_0, n.
Parameters: x_0 is the center. n is the order.
Result: Text polynomial approximation.
Result behavior: Terminal label. The Taylor polynomial is displayed as text.
Notes / restrictions: This version supports orders 0 through 3. Derivatives are numerical, so coefficients are approximate.
Example: For $f(x)=e^x$, centered at $x_0=0$, order 3, the expected form is approximately: $T3(x)=1+1x+0.5x^2+0.16666667x^3$

LIN — Linearization

Purpose: Builds the first-order linear approximation at x_0.
Inputs: Function template fields, x_0, h.
Parameters: x_0 is the expansion point. h controls the numerical derivative.
Result: Text linearization formula.
Result behavior: Terminal label.
Formula: $L(x)=f(x_0)+f'(x_0)(x-x_0)$
Notes / restrictions: Requires finite function value and finite first derivative at x_0.
Example: For $f(x)=x^2$ at $x_0=2$, the linearization is approximately: $L(x)=4+4(x-2)$

Partials

∂x — ∂/∂x at P

Purpose: Computes the first partial derivative with respect to x at a point.
Inputs: Multivariable template fields, x, y, h.
Parameters: x and y define the point P. h is the finite-difference step.
Result: Approximate scalar partial derivative.
Result behavior: Flows back into the calculator as a scalar approximate result.
Notes / restrictions: Uses a central finite difference in x.
Example: For $f(x,y)=x^2+y$, at (2,3), $\partial x\approx4$.

∂y — ∂/∂y at P

Purpose: Computes the first partial derivative with respect to y at a point.
Inputs: Multivariable template fields, x, y, h.
Parameters: x and y define the point P. h is the finite-difference step.
Result: Approximate scalar partial derivative.
Result behavior: Flows back into the calculator as a scalar approximate result.
Notes / restrictions: Uses a central finite difference in y.
Example: For $f(x,y)=x^2+y$, at (2,3), $\partial y\approx1$.

∂xx — ∂²/∂x² at P

Purpose: Computes the second partial derivative with respect to x.
Inputs: Multivariable template fields, x, y, h.
Parameters: x, y, and step h.
Result: Approximate scalar second partial derivative.
Result behavior: Flows back into the calculator as a scalar approximate result.
Notes / restrictions: Uses a second-difference stencil in x. Internally h is not allowed below a small floor.

Example: For f(x,y)=3x², ∂xx≈6.

∂yy — ∂²/∂y² at P

Purpose: Computes the second partial derivative with respect to y.

Inputs: Multivariable template fields, x, y, h.

Parameters: x, y, and step h.

Result: Approximate scalar second partial derivative.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses a second-difference stencil in y.

Example: For f(x,y)=4y², ∂yy≈8.

∂xy — ∂²/∂x∂y at P

Purpose: Computes the mixed second partial derivative.

Inputs: Multivariable template fields, x, y, h.

Parameters: x, y, and step h.

Result: Approximate scalar mixed partial derivative.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses a central mixed-difference stencil. Smoothness is assumed for meaningful interpretation.

Example: For f(x,y)=5xy, ∂xy≈5.

Multivariable tools

∇f — Gradient ∇f at P

Purpose: Computes the gradient vector at a point.

Inputs: Multivariable template fields, x, y, h.

Parameters: x and y define the point. h controls finite differences.

Result: The scalar result is the norm of the gradient. The displayed label shows the vector ⟨fx,fy⟩.

Result behavior: Returns the gradient norm as a scalar approximate result and displays the gradient vector as text.

Notes / restrictions: Both partial derivatives must be finite.

Example: For f(x,y)=x²+y² at (3,4), the gradient is approximately ⟨6,8⟩, and the returned scalar norm is approximately 10.

D_u — Dir. Deriv. at P

Purpose: Computes the directional derivative at a point in a given direction.

Inputs: Multivariable template fields, x, y, ux, uy, h.

Parameters: ux and uy define the direction vector. The docklet normalizes this direction internally.

Result: Approximate scalar directional derivative.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: The direction vector must not be zero. The label shows the normalized direction.

Example: For f(x,y)=x²+y² at (1,1) in direction (1,0), the result is approximately 2.

crit — Critical points

Purpose: Reports critical-point information for the selected multivariable template.

Inputs: Multivariable template fields only.

Parameters: Depends on the selected template.

Result: Text classification.

Result behavior: Terminal label.

Notes / restrictions: For Constant, the result is all points. For Plane, the result is all points only if both slopes are zero; otherwise none. For Quadratic 2D, the docklet solves the linear critical-point equations and classifies using the Hessian determinant.

Example: For f(x,y)=x²+y², the critical point is (0,0) local min.

H — Hessian at P

Purpose: Computes the Hessian matrix at a point.

Inputs: Multivariable template fields, x, y, h.

Parameters: x and y define the point. h controls finite differences.

Result: The scalar result is the Frobenius norm of the Hessian. The displayed label shows the matrix.

Result behavior: Returns a scalar approximate norm and displays the Hessian matrix as text.

Notes / restrictions: The mixed partial is mirrored as both off-diagonal entries.

Example: For f(x,y)=x²+y², the Hessian is approximately: [[2,0],[0,2]] The returned scalar norm is approximately 2.82842712.

∬ — Double Integral

Purpose: Computes a rectangular double integral over x and y bounds.

Inputs: Multivariable template fields, xa, xb, ya, yb.

Parameters: xa and xb define x bounds. ya and yb define y bounds.

Result: Approximate scalar double integral.

Formula: ∫[xa→xb]∫[ya→yb] f(x,y) dy dx

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses nested Simpson integration over a rectangle. Non-finite sampled values cause failure.

Example: For f(x,y)=1 over x=0..2, y=0..3, the result is approximately 6.

Differential equations

1stLIN — 1st-order linear

Purpose: Solves a first-order linear model numerically.

Equation: $y' = -a \cdot y + b + u(t)$

Inputs: a, b, y0, t1, dt, plus optional forcing-template fields.

Parameters: a is the coefficient of y. b is constant input. y0 is the initial value at t=0. t1 is the final time. dt is the numerical step size.

Result: Approximate value y(t1).

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses RK4 integration. Requires dt > 0, t1 ≥ 0, and a reasonable number of integration steps.

Example: With no forcing, a=1, b=0, y0=1, t1=1, dt=0.001, the result approximates e^{-1}.

COOL — Newton cooling

Purpose: Solves Newton-style cooling or heating with optional forcing.

Equation: $T' = -k(T-Ta) + u(t)$

Inputs: k, Ta, T0, t1, dt, plus optional forcing-template fields.

Parameters: k is the cooling constant. Ta is ambient temperature. T0 is initial temperature. t1 is final time. dt is step size.

Result: Approximate temperature at t1.

Result behavior: Flows back into the calculator as a scalar approximate result.

Notes / restrictions: Uses RK4 integration. Interpret units consistently.

Example: With k=0.1, Ta=20, T0=80, t1=10, dt=0.001, the result estimates the temperature after 10 time units.

2ndSTD — 2nd-order standard

Purpose: Solves a standard second-order system numerically.

Equation: $y'' = -2\zeta\omega n \cdot y' - \omega n^2 \cdot y + K \cdot u(t)$

Inputs: ζ, ωn, K, y0, v0, t1, dt, plus optional forcing-template fields.

Parameters: ζ is damping ratio. ωn is natural frequency. K is input gain. y0 is initial displacement. v0 is initial velocity. t1 is final time. dt is step size.

Result: Approximate y(t1). The label also reports approximate final velocity v(t1).

Result behavior: Returns final displacement as a scalar approximate result. The displayed label includes both y and v.

Notes / restrictions: Uses RK4 integration for the two-state system. Requires stable numerical settings.

Example: With ζ=0.2, ωn=5, K=1, y0=0, v0=0, a step forcing input estimates the forced response at t1.

MSD — Mass-spring-damper

Purpose: Solves a mass-spring-damper system numerically.

Equation: $m \cdot y'' + c \cdot y' + k \cdot y = u(t)$ or: $y'' = (u(t) - c \cdot y' - k \cdot y)/m$

Inputs: m, c, k, y0, v0, t1, dt, plus optional forcing-template fields.

Parameters: m is mass. c is damping. k is spring constant. y0 is initial displacement. v0 is initial velocity. t1 is final time. dt is step size.

Result: Approximate y(t1). The label also reports approximate final velocity v(t1).

Result behavior: Returns final displacement as a scalar approximate result. The displayed label includes both y and v.

Notes / restrictions: m must be nonzero. Uses RK4 integration. Choose dt small enough for the system dynamics.

Example: For m=1, c=0.2, k=4, y0=1, v0=0, t1=5, dt=0.001, the result estimates displacement after 5 time units.

Errors and limitations

The Calculus docklet is numerical. It does not perform symbolic algebra. Common error messages include: Invalid function parameters, x_0 required, a, b required Limit failed / non-finite, Derivative failed / non-finite Second derivative failed / non-finite, Third derivative failed / non-finite Integral failed / non-finite, Area failed / non-finite Improper integral diverges or is numerically unstable Arc length failed / non-finite, Volume failed / non-finite Fourier real part failed / non-finite, Fourier imaginary part failed / non-finite Endpoint non-finite, Require $f(a) \cdot f(b) \le 0$, Root solve failed, Min search failed Max search failed, For v1, require $0 \le n \le 3$ Invalid multivariable parameters x, y required, Directional derivative failed / invalid direction No isolated critical point, Double integral failed / non-finite Invalid forcing parameters, Missing parameters Integration failed / Δt too small / interval too large m must be nonzero

Main limitations: Taylor order is limited to 0 through 3. Root solving requires a sign change or zero at an endpoint. Extremum search is scan-based and may miss very narrow features. Simpson integration uses a fixed bounded sampling strategy. Improper integrals are accepted only when the numerical stability checks pass. Differential-equation solvers limit the number of integration steps. Trigonometric template phases and Fourier frequency use radians. The Log template is valid only for x > 0. The Rational template is invalid where the denominator is effectively zero. Power template evaluation for negative x and non-integer powers is unsupported. The docklet reports numerical approximations, not proof-quality symbolic results.

Practical usage notes

Use limits when checking behavior near a point, but compare lim− and lim+ when discontinuities are possible. Use d/dx, d^2, and d^3 with care. A smaller h is not always better; very small steps can amplify numerical cancellation, especially for third derivatives. Use ∫ for signed accumulation and ∫|f| for geometric area. If the function crosses the x-axis, these two results can be very different. Use root only when the interval brackets a sign change. If no sign change exists, the bisection solver is not the right tool. Use min and max as numerical search tools, not as symbolic optimization. They are most reliable for smooth functions over moderate intervals. Use TAY and LIN when the goal is a local approximation formula rather than a scalar value. For multivariable work, use ∂x and ∂y for individual slopes, ∇f for the full local direction of steepest increase, and H for second-order curvature information. Use crit mainly with the Quadratic 2D template. Constant and plane templates have special cases, but only the quadratic template produces isolated critical-point classification. For differential equations, keep dt small enough for the model but not so small that the interval requires excessive steps. The result is a numerical state at t1, not a closed-form solution. The Calculus docklet is designed for fast specialist numerical calculations using structured templates. Results should be interpreted according to the selected template, interval, step size, numerical method, and the stated domain and stability limits.

Appendix G.2 — Combinatorics

Overview

The Combinatorics docklet provides counting, sequence, set-number, and probability tools for discrete mathematics and applied probability. It covers factorials, permutations, combinations, derangements, repeated selections, multinomial counts, Fibonacci/Catalan/Bell/Triangular numbers, Stirling numbers, Lah numbers, integer partitions, compositions, Eulerian numbers, hypergeometric probability, and binomial probability. The docklet is intended for users who need exact or approximate combinatorial values without building formulas manually in the main calculator.

Opening and using the docklet

The Combinatorics docklet appears as a parked, expandable panel. When expanded, it shows a vertical list of functions grouped by topic. Selecting a function opens a firm prompt dialog. The prompt contains the fields required by that function. After entering valid values, tap Compute to calculate the result. Tap Cancel to close the prompt without computing. Tapping outside the prompt does not dismiss it. Instead, the prompt remains open.

Current operand behavior

The Combinatorics docklet does not use the current calculator operand as an automatic input value. All required values must be entered manually in the prompt fields. Blank fields do not mean “use the current operand.” If a required field is blank or invalid, Compute remains disabled. When a result is a scalar numeric value, it returns to the calculator as the current numeric result and can continue in expressions. When a result is too large for ordinary scalar flow or is returned as an exact string or scientific approximation string, it behaves as a terminal label-style result.

Prompt dialogs

Each function opens a prompt with fields appropriate to that function.

General prompt behavior: Required fields must be filled. Fields must parse as valid decimal/integer values, depending on the function. Most combinatorics functions require integers. Compute is disabled until the prompt validates. Cancel closes the prompt without computing. Outside taps do not dismiss the prompt. The keyboard can be dismissed by moving through fields or dragging inside the prompt area. The Mult function has a special k list field. This field accepts a list of integers rather than a single decimal value.

Decimal places and rounding

Numeric scalar results are rounded using the app’s global decimalPlaces setting. The setting is clamped to the range 0…18. Exact integer string results are not rounded. They are displayed as full integer text. Large approximate values may be displayed in scientific notation, using the active decimalPlaces setting for the mantissa. Small probability results are clamped to zero when their absolute value is less than or equal to 1e-12. Negative zero display artifacts are normalized to 0.

Locale and separators

The docklet uses the selected numeric locale for displayed decimal text. In dot-decimal locales, function argument labels use commas: nCr[count](10,3) In comma-decimal locales, argument labels use semicolons: nCr[count](10;3) The Mult function’s k list placeholder changes by locale: k1,k2,k3... (sum=n) in dot-decimal locales, and: k1;k2;k3... (sum=n) in comma-decimal locales. The actual k list parser accepts commas, semicolons, spaces, tabs, and new lines as separators.

Angle-unit behavior

The Combinatorics docklet does not use angle units.

Currency/base-currency behavior

The Combinatorics docklet does not use currency or base-currency settings.

Result behavior

The docklet produces two main result styles. Scalar numeric results return to the calculator as ordinary numeric values. These can continue in calculator expressions. Label-style results are used when the output is too large for ordinary scalar handling, when the docklet returns an exact integer string, or when an approximate scientific string is used. These are displayed as terminal results. Probability functions return scalar decimal probabilities. The ticker shows a compact function label, such as: nCr[count](10,3) or: Bin[prob](10,0.5,3) The ticker does not repeat the numeric result; the result is shown in the main display or label display.

Result tags

[NUM] means the result is a numeric scalar value. [APPROX] means the result is an approximation, usually because the value is large and represented through log-gamma/scientific notation, or because a probability calculation uses log-space numerical evaluation. [LABEL] means the result is displayed as a label-style value, such as an exact large integer string or a non-flowing terminal output.

Function groups

The Combinatorics docklet is organized into these groups:

Group 1 — Basic Counting: n!, nPr, nCr, !n

Group 2 — With Repetition & Multinomial:, n^k, nHk, Mult

Group 3 — Sequences: F_n, C_n, B_n, T_n

Group 4 — Advanced: s(n, k), S(n, k), L(n, k), p(n), 2^{n-1}, A(n, k)

Group 5 — Probability / Applied: HypG, Bin

Function reference

n! — Factorial

Purpose: Computes the factorial of a nonnegative integer.

Inputs: n

Parameters: n is the integer whose factorial is required.

Result: Returns: $n! = n \times (n-1) \times \ldots \times 1$

Result behavior: For small values, returns a scalar numeric result. For larger values, returns an exact integer string or an approximate scientific string.

Notes / restrictions: n must be an integer. $n \geq 0$ is required. Direct numeric factorial is used up to $n = 27$. Exact string factorial is used only for a limited range. For larger values, the docklet uses a log-gamma approximation shown in scientific notation.

Example: $n = 5 \rightarrow 120$.

nPr — Permutations

Purpose: Computes ordered selections of r items from n.

Inputs: n, r

Parameters: n is the total number of items. r is the number selected.

Result: Returns: $nPr = n! / (n-r)!$

Result behavior: For small values, returns a scalar numeric result. For larger values, returns an exact integer string when possible. For very large values, returns an approximate scientific string.

Notes / restrictions: n and r must be integers. Requires: $0 \leq r \leq n$ n must be nonnegative. Exact integer string output is supported up to the docklet's large-value cap.

Example: $n = 5$, $r = 2 \rightarrow 20$.

nCr — Combinations

Purpose: Computes unordered selections of r items from n.

Inputs: n, r

Parameters: n is the total number of items. r is the number selected.

Result: Returns: $nCr = n! / (r!(n-r)!)$

Result behavior: For small values, returns a scalar numeric result. For larger values, returns an exact integer string when possible. For very large values, returns an approximate scientific string.

Notes / restrictions: n and r must be integers. Requires: $0 \leq r \leq n$ n must be nonnegative.

Example: $n = 10$, $r = 3 \rightarrow 120$.

!n — Derangements

Purpose: Computes the number of derangements of n objects: permutations with no fixed points.

Inputs: n

Parameters: n is the number of objects.

Result: Returns the derangement count !n.

Result behavior: For n ≤ 100, returns a scalar numeric result. For larger n, returns an approximate scientific string.

Notes / restrictions: n must be an integer. n ≥ 0 is required. Large values use the approximation: !n ≈ n! / e

Example: n = 4 → 9.

n^k — k-tuples (n^k)

Purpose: Computes the number of ordered k-tuples formed from n choices with repetition allowed.

Inputs: n, k

Parameters: n is the number of available choices. k is the tuple length.

Result: Returns: n^k

Result behavior: Scalar numeric result when the value fits the docklet's decimal computation.

Notes / restrictions: n and k must be integers. Requires: n≥0 k≥0 Overflow can occur for large values.

Example: n = 10, k = 3 → 1000.

nHk — Combinations w/ Rep

Purpose: Computes combinations with repetition, often called "multichoose."

Inputs: n, k

Parameters: n is the number of item types. k is the number selected.

Result: Returns: nHk = C(n+k−1, k)

Result behavior: For small values, returns a scalar numeric result. For larger values, returns an exact integer string when possible. For very large values, returns an approximate scientific string.

Notes / restrictions: n and k must be integers. Requires: n≥0 k≥0 If n = 0, the result is 1 only when k = 0; otherwise it is 0.

Example: n = 3, k = 2 → 6.

Mult — Multinomial

Purpose: Computes a multinomial coefficient.

Inputs: n, k list

Parameters: n is the total count. k list is a list of category counts.

Result: Returns: n! / (k1! k2! ... km!)

Result behavior: For small values, returns a scalar numeric result. For larger values, returns an exact integer string when possible. For very large values, returns a scientific-format result.

Notes / restrictions: n must be an integer. All ki values must be integers. Requires: sum(ki) = n all ki ≥ 0 The k list field accepts separators such as commas, semicolons, spaces, tabs, and line breaks.

Example: n = 6, k list = 2,1,3 → 60.

F_n — Fibonacci

Purpose: Computes the nth Fibonacci number.

Inputs: n

Parameters: n is the Fibonacci index.

Result: Returns F_n.

Result behavior: Scalar numeric result.

Notes / restrictions: n must be an integer. n ≥ 0 is required. The sequence uses: $F_0 = 0$ $F_1 = 1$

Example: n = 10 → 55.

C_n — Catalan

Purpose: Computes the nth Catalan number.

Inputs: n

Parameters: n is the Catalan index.

Result: Returns: $C_n = C(2n,n) / (n+1)$

Result behavior: For small values, returns a scalar numeric result. For larger values, returns an exact integer string up to a practical cap. For very large values, returns an approximate scientific string.

Notes / restrictions: n must be an integer. n ≥ 0 is required. Exact string Catalan output is capped for responsiveness.

Example: n = 5 → 42.

B_n — Bell Number

Purpose: Computes the nth Bell number, the number of partitions of a set of size n.

Inputs: n

Parameters: n is the set size.

Result: Returns B_n.

Result behavior: Scalar numeric result.

Notes / restrictions: n must be an integer. n ≥ 0 is required. The implemented cap is: n ≤ 80 If n > 80, the docklet reports n too large.

Example: n = 4 → 15.

T_n — Triangular

Purpose: Computes the nth triangular number.

Inputs: n

Parameters: n is the triangular-number index.

Result: Returns: $T_n = n(n+1)/2$

Result behavior: Scalar numeric result.

Notes / restrictions: n must be an integer. n ≥ 0 is required.

Example: n = 10 → 55.

s(n,k) — Stirling 1st

Purpose: Computes the unsigned Stirling number of the first kind.

Inputs: n, k

Parameters: n is the number of elements. k is the number of cycles.

Result: Returns unsigned s(n,k).

Result behavior: Scalar numeric result.

Notes / restrictions: n and k must be integers. Requires: 0≤k≤n The implemented cap is: n ≤ 120 If n > 120, the docklet reports n too large.

Example: s(4,2) → 11.

S(n,k) — Stirling 2nd

Purpose: Computes the Stirling number of the second kind.

Inputs: n, k

Parameters: n is the number of elements. k is the number of nonempty subsets.

Result: Returns S(n,k).

Result behavior: Scalar numeric result.

Notes / restrictions: n and k must be integers. Requires: 0≤k≤n The implemented cap is: n ≤ 120 If n > 120, the docklet reports n too large.

Example: S(5,2) → 15.

L(n,k) — Lah Numbers

Purpose: Computes Lah numbers.

Inputs: n, k

Parameters: n is the number of elements. k is the number of ordered blocks.

Result: Returns: L(n,k) = C(n−1,k−1) · n! / k!

Result behavior: Scalar numeric result.

Notes / restrictions: n and k must be integers. Requires: 0≤k≤n L(0,0) returns 1. If k = 0 and n > 0, the result is 0.

Example: L(4,2) → 36.

p(n) — Integer Partitions

Purpose: Computes the number of integer partitions of n.

Inputs: n

Parameters: n is the integer to partition.

Result: Returns p(n).

Result behavior: Scalar numeric result.

Notes / restrictions: n must be an integer. n ≥ 0 is required. The implemented cap is: n ≤ 250 If n > 250, the docklet reports n too large.

Example: n = 5 → 7.

2^{n-1} — Compositions

Purpose: Computes the number of compositions of a positive integer n.

Inputs: n

Parameters: n is the integer being composed.

Result: Returns: 2^(n−1)

Result behavior: Scalar numeric result when the value fits the decimal computation.

Notes / restrictions: n must be an integer. Requires: n≥1 Overflow can occur for large n.

Example: n = 5 → 16.

A(n,k) — Eulerian Numbers

Purpose: Computes Eulerian numbers.

Inputs: n, k

Parameters: n is the size parameter. k is the Eulerian index.

Result: Returns A(n,k).

Result behavior: Scalar numeric result.

Notes / restrictions: n and k must be integers. Requires: n≥1 0 ≤ k ≤ n−1 The implemented cap is: n ≤ 120 If n > 120, the docklet reports n too large.

Example: A(4,1) → 11.

HypG — Hypergeometric

Purpose: Computes the hypergeometric probability mass function.

Inputs: N, K, n, k

Parameters: N is the population size. K is the number of successes in the population. n is the number of draws. k is the observed number of successes.

Result: Returns: P(X = k) = C(K,k) C(N−K,n−k) / C(N,n)

Result behavior: Scalar probability result.

Notes / restrictions: All four fields must be integers. Requires valid hypergeometric constraints: N≥0 K≥0 n≥0 k≥0 K≤N n≤N max(0, n−(N−K)) ≤ k ≤ min(n,K) The calculation uses log-space evaluation for numerical stability.

Example: N = 52, K = 4, n = 5, k = 1 → probability of drawing exactly one success.

Bin — Binomial PMF

Purpose: Computes the binomial probability mass function.

Inputs: n, p, k

Parameters: n is the number of trials. p is the probability of success. k is the number of successes.

Result: Returns: P(X = k) = C(n,k) p^k (1−p)^(n−k)

Result behavior: Scalar probability result.

Notes / restrictions: n and k must be integers. p must be a decimal probability. Requires: n≥0 0≤k≤n 0≤p≤1 The function handles boundary cases near p = 0 and p = 1 to avoid logarithm errors. The calculation uses log-space evaluation for numerical stability.

Example: n = 10, p = 0.5, k = 3 → 0.1171875.

Errors and limitations

Common validation messages include: n must be integer n,r must be integers n,k must be integers N,K,n,k must be integers n,k must be integers; p must be decimal n must be ≥ 0 n must be ≥ 1 Require 0 ≤ r ≤ n Require n ≥ 0, k ≥ 0 Require 0 ≤ k ≤ n Require 0 ≤ k ≤ n-1 Require sum(ki)=n, all ki≥0 k list invalid (use e.g. 2,1,3) Invalid constraints, Invalid inputs, n too large, Overflow Not implemented

Important limitations: Many functions require exact integer inputs. Decimal values that are not effectively integers are rejected. Several functions have explicit caps to keep the docklet responsive. Some large results are returned as exact strings rather than scalar calculator values. Some very large results are returned in approximate scientific notation. Probability functions use log-space numerical computation and return decimal approximations. Bell Number is capped at n ≤ 80. Integer Partitions is capped at n ≤ 250. Stirling 1st, Stirling 2nd, and Eulerian Numbers are capped at n ≤ 120. Direct scalar factorial is capped at n = 27. Derangements use direct computation up to n = 100, then switch to approximation. Exact Catalan string output is capped for responsiveness. Overflow may occur in functions that grow very quickly, especially powers, factorial-based expressions, and large sequence values.

Practical usage notes

Use nPr when order matters. Use nCr when order does not matter. Use nHk when selections allow repetition and order does not matter. Use n^k when repetition is allowed and order matters. Use Mult when a total n is divided into specified category counts. Use S(n,k) for partitions into nonempty unlabeled subsets. Use Bell numbers for the total number of all such partitions over every possible k. Use p(n) for integer partitions, not set partitions. Use HypG when sampling without replacement. Use Bin when trials are independent with the same success probability. When a result appears as a long exact integer string or scientific string, treat it as a terminal result rather than a normal scalar value for continued expression entry.

Appendix G.3 — Complex

Overview

The Complex docklet provides tools for working with complex numbers in rectangular form, polar form, arithmetic, powers, roots, exponential and logarithmic functions, trigonometric functions, hyperbolic functions, mapping tools, roots of unity, and phase unwrapping. The docklet is intended for users who need compact complex-number operations while staying inside the CCalc workflow. It stores a working complex value: z = Re + Im·i Most functions operate on this stored value, with prompt fields allowing the user to confirm or override the real and imaginary parts before computing.

Opening and using the docklet

The Complex docklet appears as a parked, expandable panel. When expanded, it shows the list of complex functions. Selecting a function opens a firm prompt dialog. For most complex operations, the prompt shows fields for Re and Im, prefilled with the current stored complex value. The user may accept these values or edit them before tapping Compute. Tapping Cancel closes the prompt without computing. Tapping outside the prompt does not dismiss it; the dialog is firm and remains open.

Current operand behavior

The Complex docklet has two related but separate concepts:

1. The normal calculator operand, used for scalar numeric results such as |z|, arg, and unwrap.

2. The stored complex value z = Re + Im·i, used by most complex functions.

Re←x and Im←x explicitly take a numeric input x and assign it to the real or imaginary part of the stored complex value. Most functions do not use the current calculator operand directly. They use the stored complex value shown in the prompt fields. When the prompt opens, Re and Im are prefilled from the stored complex value. Scalar results can continue in calculator expressions. Complex or multi-value results are shown as label-style results and are not ordinary scalar calculator values.

Prompt dialogs

Prompt dialogs use decimal numeric fields unless a function asks for an integer n.

General behavior: Compute is enabled only when the required fields are valid. Cancel closes the prompt without computing. Outside taps do not dismiss the prompt. Decimal input accepts dot or comma decimal entry and normalizes internally. Scientific notation such as 1e-12 is accepted by the decimal sanitizer. Pair fields such as Re / Im and wRe / wIm may appear side by side. The prompt scrolls when there are more fields than fit comfortably. Blank-field behavior There is no general “blank = current operand” rule. For functions using Re and Im, those fields are usually prefilled with the current stored complex value. If the user clears one required field, Compute remains disabled until the field is valid again. Special prompt: Rect ⇄ Polar The ⇄ function supports two input modes in one prompt: Re, Im or: r, θ Only one complete pair may be used. If the user enters rectangular values, the polar fields are cleared. If the user enters polar values, the rectangular fields are cleared. The function requires exactly one complete pair. Integer fields The following functions require integer n: • z^n, * $\sqrt[n]{z}$, * ω^n z^n accepts positive, zero, or negative integer exponents, subject to division restrictions. $\sqrt[n]{z}$ requires n ≠ 0. ω^n requires n > 0.

Decimal places and rounding

Displayed numeric results use CCalc’s global decimalPlaces setting. The value is clamped to the range 0…18. Complex results round the real and imaginary parts separately. Very small display artifacts such as -0 are normalized to 0. Some display formatting also snaps values close to common clean values, such as zero, integers, or simple half-step values, when presenting readable complex lists such as roots.

Locale and separators

The Complex docklet follows the selected numeric locale for displayed decimal separators. When the locale uses comma decimal formatting, displayed decimals use commas. Internally, decimal input may still accept either comma or dot and is normalized for parsing. Multi-value label outputs use semicolon separators: 0: 1; 1: -0,5 + 0,8660254i; 2: -0,5 − 0,8660254i in comma-decimal locales, and: 0: 1; 1: -0.5 + 0.8660254i; 2: -0.5 − 0.8660254i in dot-decimal locales.

Angle-unit behavior

The Complex docklet uses the active CCalc angle unit for user-facing angle input and output in these functions: ⇄ when entering or displaying polar θ arg, cis, unwrap the displayed argument part of ln Supported angle labels are: deg rad grad rev Internally, complex exponential, logarithmic, trigonometric, and hyperbolic calculations use radians where required by the mathematical formulas. User-facing angle input and output are converted according to the active angle unit.

Result behavior

The Complex docklet produces two main kinds of result. Scalar results return an ordinary number to the calculator display and can continue in calculator expressions. These functions are: • |z| • arg • unwrap Complex results update the stored complex value and display a label-style complex result. These are not ordinary scalar calculator values. Collection-style results, such as $\sqrt[n]{z}$ and ω^n, display multiple complex values. The first value is stored as the current complex result, while the visible result text lists the collection.

Result tags

[NUM] means a numeric scalar result. [APPROX] means an approximate numerical result. Most complex transcendental, polar, root, trigonometric, logarithmic, and mapping functions use approximate numerical computation. [LABEL] means a descriptive or non-scalar result, including complex-number display, multi-value output, or error-style output.

Function groups

The Complex docklet is organized into these groups:

Group 1 — Entry & Representation:, Re←x, Im←x, swap, Im=0, ⇄
Group 2 — Core Transforms: z̄, |z|, arg, z/|z|, 1/z
Group 3 — Arithmetic: +, −, ×, ÷
Group 4 — Powers & Roots: z^2, z^n, √z, $\sqrt[n]{z}$
Group 5 — Exp & Logs: exp, ln, log
Group 6 — Trig: sin, cos, tan
Group 7 — Hyperbolic: sinh, cosh, tanh, asinh, acosh
Group 8 — Specialist / Mapping: proj, cis, Γ(z)
Group 9 — Collections: ω^n, unwrap

Function reference

Re←x — Re from x

Purpose: Sets the real part of the stored complex value from a decimal input.
Inputs: x
Parameters: x is the new real part.
Result: Updates: z = x + Im·i where Im remains unchanged.
Result behavior: Complex label-style result. The stored complex value is updated.
Notes / restrictions: x must be a valid decimal number.
Example: Current z = 3 + 4i, x = 10 → z = 10 + 4i.

Im←x — Im from x

Purpose: Sets the imaginary part of the stored complex value from a decimal input.
Inputs: x
Parameters: x is the new imaginary part.
Result: Updates: z = Re + x·i where Re remains unchanged.
Result behavior: Complex label-style result. The stored complex value is updated.
Notes / restrictions: x must be a valid decimal number.
Example: Current z = 3 + 4i, x = -2 → z = 3 − 2i.

swap — Swap Re/Im

Purpose: Swaps the real and imaginary parts of z.
Inputs: Re, Im
Parameters: Re is the real part. Im is the imaginary part.
Result: Returns: Im + Re·i
Result behavior: Complex label-style result. The stored complex value is updated.
Notes / restrictions: Both fields must be valid decimals.
Example: z = 3 + 4i → 4 + 3i. Im=0 — Clear Im (real)
Purpose: Clears the imaginary part of z.
Inputs: Re, Im
Parameters: Re is the real part to preserve. Im is shown for confirmation or editing.
Result: Returns: Re + 0i shown as a real value.
Result behavior: Complex label-style result. The stored complex value becomes real.
Notes / restrictions: Both fields must be valid decimals.
Example: z = 3 + 4i → 3.

⇄ — Rect ⇄ Polar

Purpose: Converts between rectangular and polar representation.
Inputs: Either Re, Im or r, θ.
Parameters: Re and Im define rectangular form. r and θ define polar form. θ uses the active angle unit.
Result: If rectangular input is used, the result displays: Re=<Re>; Im=<Im> | r=<magnitude>; θ=<angle> If polar input is used, the result converts to rectangular form: r(cos θ + i sin θ)
Result behavior: Rectangular-to-polar displays a label-style polar description and keeps the rectangular complex value. Polar-to-rectangular stores and displays the converted complex value.
Notes / restrictions: Use exactly one complete pair: either Re/Im or r/θ. r must be greater than or equal to 0.
Example: Re = 3, Im = 4 → r = 5, θ ≈ 0.927295 rad. r = 1, θ = 90 deg → 0 + 1i.

z̄ — Conjugate

Purpose: Computes the complex conjugate.
Inputs: Re, Im
Parameters: Re and Im define z.
Result: Returns: Re − Im·i

Result behavior: Complex label-style result. The stored complex value is updated.
Notes / restrictions: Both fields must be valid decimals.
Example: z = 3 + 4i → 3 − 4i.

|z| — Magnitude

Purpose: Computes the magnitude of z.
Inputs: Re, Im
Parameters: Re and Im define z.
Result: Returns: |z| = √(Re² + Im²)
Result behavior: Scalar approximate numeric result. The result can continue in calculator expressions.
Notes / restrictions: Both fields must be valid decimals.
Example: z = 3 + 4i → 5.

arg — Argument

Purpose: Computes the argument, or phase angle, of z.
Inputs: Re, Im
Parameters: Re and Im define z.
Result: Returns: atan2(Im, Re) converted to the active angle unit.
Result behavior: Scalar approximate numeric result. The result can continue in calculator expressions.
Notes / restrictions: Both fields must be valid decimals. The output unit follows the active angle setting.
Example: With radians active, z = 1 + i → 0.78539816.

z/|z| — Normalize

Purpose: Normalizes z to unit magnitude.
Inputs: Re, Im
Parameters: Re and Im define z.
Result: Returns: z / |z|
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: z must not be zero. If |z| = 0, the function reports Division by 0.
Example: z = 3 + 4i → 0.6 + 0.8i.

1/z — Reciprocal

Purpose: Computes the reciprocal of z.
Inputs: Re, Im
Parameters: Re and Im define z.
Result: Returns: 1 / (a + bi) = a/(a²+b²) − b/(a²+b²)i
Result behavior: Complex label-style result. The stored complex value is updated.
Notes / restrictions: z must not be zero. If Re = 0 and Im = 0, the function reports Division by 0.
Example: z = 3 + 4i → 0.12 − 0.16i.

+ — Add (z + w)

Purpose: Adds two complex numbers.
Inputs: Re, Im, wRe, wIm
Parameters: Re, Im define z. wRe, wIm define w.
Result: Returns: z+w
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: All fields must be valid decimals.
Example: z = 3 + 4i, w = 1 − 2i → 4 + 2i.

− — Subtract (z − w)

Purpose: Subtracts one complex number from another.
Inputs: Re, Im, wRe, wIm
Parameters: Re, Im define z. wRe, wIm define w.
Result: Returns: z−w
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: All fields must be valid decimals.
Example: z = 3 + 4i, w = 1 − 2i → 2 + 6i.

× — Multiply (z × w)

Purpose: Multiplies two complex numbers.
Inputs: Re, Im, wRe, wIm
Parameters: Re, Im define z = a + bi. wRe, wIm define w = c + di.

Result: Returns: (ac − bd) + (ad + bc)i
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: All fields must be valid decimals.
Example: z = 3 + 4i, w = 1 − 2i → 11 − 2i.

÷ — Divide (z ÷ w)

Purpose: Divides one complex number by another.
Inputs: Re, Im, wRe, wIm
Parameters: Re, Im define z. wRe, wIm define divisor w.
Result: Returns: z/w
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: w must not be zero. If wRe = 0 and wIm = 0, the function reports Division by 0.
Example: z = 3 + 4i, w = 1 − 2i → -1 + 2i.

z² — Square (z²)

Purpose: Squares a complex number.
Inputs: Re, Im
Parameters: Re, Im define z = a + bi.
Result: Returns: $z^2 = (a^2 - b^2) + 2ab \cdot i$
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: Both fields must be valid decimals.
Example: z = 3 + 4i → -7 + 24i.

zⁿ — Power (zⁿ)

Purpose: Raises z to an integer power.
Inputs: Re, Im, n
Parameters: Re, Im define z. n is an integer exponent.
Result: Returns: z^n using polar exponentiation.
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: n must be an integer. n = 0 returns 1 + 0i. If z = 0 and n < 0, the function reports Division by 0.
Example: z = 1 + i, n = 2 → 0 + 2i.

√z — Square Root

Purpose: Computes the principal square root of z.
Inputs: Re, Im
Parameters: Re, Im define z.
Result: Returns the principal square root.
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: Both fields must be valid decimals.
Example: z = -1 + 0i → 0 + 1i.

ⁿ√z — n-th Roots

Purpose: Computes all n-th roots of z.
Inputs: Re, Im, n
Parameters: Re, Im define z. n is an integer root order.
Result: Displays all roots in a numbered list: 0: root0; 1: root1; ... The first listed root is stored as the current complex result.
Result behavior: Collection-style label result. The stored complex value is set to the principal listed root.
Notes / restrictions: n must not be 0. For negative n, the function computes roots of the reciprocal behavior. If z = 0 and n < 0, the function reports Division by 0.
Example: z = 1 + 0i, n = 3 → roots 1, -0.5 + 0.8660254i, -0.5 − 0.8660254i.

exp — exp(z)

Purpose: Computes the complex exponential.
Inputs: Re, Im
Parameters: Re, Im define z = a + bi.
Result: Returns: e^(a+bi) = e^a(cos b + i sin b)
Result behavior: Complex approximate label-style result. The stored complex value is updated.
Notes / restrictions: Very large real parts may overflow to invalid or non-finite numerical results.
Example: z = 0 + πi → approximately -1.

ln — ln(z)

Purpose: Computes the principal natural logarithm of z.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Returns: ln(z) = ln|z| + i·arg(z) The displayed label shows ln|z| and arg in the active angle unit. The stored complex logarithm uses radians for the imaginary component.

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: ln(0) is undefined. If z = 0, the function reports ln(0) undefined.

Example: z = 1 + i → ln|z| ≈ 0.34657359, arg ≈ 0.78539816 rad.

log — log10(z)

Purpose: Computes the principal base-10 logarithm of z.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Returns: log10(z) = ln(z) / ln(10)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: log10(0) is undefined. If z = 0, the function reports log(0) undefined.

Example: z = 10 + 0i → 1.

sin — sin(z)

Purpose: Computes the complex sine.

Inputs: Re, Im

Parameters: Re, Im define z = a + bi.

Result: Returns: sin(a+bi) = sin(a)cosh(b) + i cos(a)sinh(b)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: The real and imaginary parts are used directly in the complex formula.

Example: z = 0 + 1i → 0 + 1.17520119i.

cos — cos(z)

Purpose: Computes the complex cosine.

Inputs: Re, Im

Parameters: Re, Im define z = a + bi.

Result: Returns: cos(a+bi) = cos(a)cosh(b) − i sin(a)sinh(b)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: The real and imaginary parts are used directly in the complex formula.

Example: z = 0 + 1i → 1.54308063.

tan — tan(z)

Purpose: Computes the complex tangent.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Returns: tan(z) = sin(z) / cos(z)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: If the complex cosine denominator is zero, the function reports Division by 0.

Example: z = 0 + 1i → approximately 0 + 0.76159416i.

sinh — sinh(z)

Purpose: Computes the complex hyperbolic sine.

Inputs: Re, Im

Parameters: Re, Im define z = a + bi.

Result: Returns: sinh(a+bi) = sinh(a)cos(b) + i cosh(a)sin(b)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: Large values may exceed normal floating-point range.

Example: z = 1 + 0i → 1.17520119.

cosh — cosh(z)

Purpose: Computes the complex hyperbolic cosine.

Inputs: Re, Im

Parameters: Re, Im define z = a + bi.

Result: Returns: cosh(a+bi) = cosh(a)cos(b) + i sinh(a)sin(b)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: Large values may exceed normal floating-point range.

Example: z = 1 + 0i → 1.54308063.

tanh — tanh(z)

Purpose: Computes the complex hyperbolic tangent.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Returns: tanh(z) = sinh(z) / cosh(z)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: If the complex hyperbolic cosine denominator is zero, the function reports Division by 0.

Example: z = 1 + 0i → 0.76159416.

asinh — asinh(z)

Purpose: Computes the inverse hyperbolic sine.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Uses: asinh(z) = ln(z + √(z² + 1))

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: If the logarithm expression becomes invalid numerically, the function reports Invalid input.

Example: z = 1 + 0i → approximately 0.88137359.

acosh — acosh(z)

Purpose: Computes the inverse hyperbolic cosine.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Uses: acosh(z) = ln(z + √(z+1)√(z−1))

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: The result is the principal complex value. If the logarithm expression becomes invalid numerically, the function reports Invalid input.

Example: z = 1 + 0i → 0.

proj — Riemann Proj

Purpose: Computes the stereographic projection of z onto the unit sphere.

Inputs: Re, Im

Parameters: Re = x, Im = y.

Result: Displays: X = 2x / (x²+y²+1) Y = 2y / (x²+y²+1) Z = (x²+y²−1) / (x²+y²+1)

Result behavior: Label-style mapping result. The stored complex value is set to X + Yi, while the visible result also shows Z.

Notes / restrictions: The denominator is always positive for finite x and y.

Example: z = 0 + 0i → X=0; Y=0; Z=-1.

cis — Unit Vector

Purpose: Creates the unit complex number from an angle.

Inputs: x

Parameters: x is an angle in the active angle unit.

Result: Returns: cos(x) + i sin(x)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: The input angle uses the active CCalc angle unit.

Example: With degrees active, x = 90 → 0 + i.

Γ(z) — Gamma

Purpose: Computes an approximation to the complex Gamma function.

Inputs: Re, Im

Parameters: Re, Im define z.

Result: Returns an approximate complex value for: Γ(z)

Result behavior: Complex approximate label-style result. The stored complex value is updated.

Notes / restrictions: The function uses an approximation suitable for many ordinary values but is not a symbolic Gamma implementation. Reflection is used for Re(z) < 0.5. Near poles or difficult regions, numerical instability or invalid results may occur.

Example: z = 5 + 0i → approximately 24.

ωⁿ — Roots of Unity

Purpose: Computes the n complex roots of unity.

Inputs: n

Parameters: n is the number of roots.

Result: Displays a numbered list: 0: 1; 1: ...; 2: ... The first root, 1 + 0i, is stored as the current complex value.

Result behavior: Collection-style label result. The displayed output is non-scalar.

Notes / restrictions: n must be an integer greater than 0.

Example: n = 4 → 1, i, -1, -i.

unwrap — Phase Unwrap

Purpose: Adjusts an angle so it is closest to a reference angle, within one principal wrap interval.

Inputs: θ, ref θ

Parameters: θ is the phase angle to unwrap. ref θ is the reference phase. Both use the active angle unit.

Result: Returns a scalar angle equivalent to θ, adjusted by multiples of a full turn so that it is close to ref θ.

Result behavior: Scalar approximate numeric result. The result can continue in calculator expressions.

Notes / restrictions: Both inputs must be valid decimals. The output uses the active angle unit.

Example: With degrees active, θ = 370, ref θ = 0 → 10.

Errors and limitations

Common error messages include: Invalid input Division by 0 ln(0) undefined log(0) undefined n must be ≠ 0 n must be > 0 r must be ≥ 0 Fill either (Re,Im) OR (r,θ) — not both Fill (Re, Im) or (r, θ), No roots Not implemented

Important limitations: Most advanced complex functions are numerical and approximate. Complex logarithms use the principal branch. Complex square roots use the principal square root. $\sqrt[n]{z}$ lists all roots but stores the first listed root as the current complex value. ω^n lists all roots of unity but stores the first root. tan(z) and tanh(z) can fail when their denominators are zero. 1/z, z/|z|, division, and negative powers of zero can produce division errors. ln(0) and log10(0) are undefined. Γ(z) is approximate and may be unstable near poles or extreme values. Very large inputs may overflow floating-point calculations. Very small numerical artifacts are cleaned for display, but results remain numerical approximations.

Practical usage notes

Use Re←x and Im←x to build or adjust the stored complex value one component at a time. Use ⇄ when converting between rectangular and polar interpretation. Enter only one pair: rectangular or polar. Use |z| and arg when a scalar result is needed for later calculator operations. Use +, −, ×, and ÷ for operations between the stored z and a second complex number w. Use z^n, √z, and $\sqrt[n]{z}$ for powers and roots. For multiple roots, read the full displayed list; the calculator stores only the principal listed result. Use cis to create a unit complex number from the current angle-unit system. Use unwrap for phase continuity when comparing a phase to a reference phase.

Appendix G.4 — Cryptography

Overview

The Cryptography docklet provides integer-based cryptography and number-theory tools for modular arithmetic, divisibility, prime testing, factorization, Chinese remainder calculations, small RSA and Diffie–Hellman demonstrations, discrete logarithm search, XOR, and bit shifting. The docklet is designed for compact educational, analytical, and verification-style calculations. It works with signed 64-bit integer input, not arbitrary-precision cryptographic integers. Large real-world cryptographic key sizes are outside its operating range.

Opening and using the docklet

The Cryptography docklet appears as a parked, expandable panel. When parked, only the docklet header is visible. Expanding the docklet shows the function list. Selecting a function opens a firm prompt dialog. Enter the required integer values and tap Compute. Tap Cancel to close the prompt without calculating. Tapping outside the prompt does not dismiss it. Outside taps trigger a warning response. This prevents accidental closure while entering multi-field modular or key parameters.

Current operand behavior

The Cryptography docklet does not use the current calculator operand as a blank-field default. Every prompt field shown by the selected function must be filled with a valid integer before Compute is enabled. There are no optional blank fields in this docklet. Some functions return a numeric scalar that flows back into the calculator. Other functions return descriptive or multi-value results, such as factorization, RSA key data, or CRT congruence output.

Prompt dialogs

All Cryptography prompt fields accept integers only. The sanitizer accepts digits and a leading sign. Decimal points, decimal commas, letters, spaces, and other symbols are removed from integer fields. A plus or minus sign is only kept when it appears at the beginning.

General prompt behavior: All fields are required. All fields must parse as signed 64-bit integers. Compute is disabled until all fields are valid. Cancel closes the prompt and performs no calculation. Outside taps do not dismiss the prompt. There are no choice menus. There are no list-entry fields. There is no blank-field operand behavior.

Decimal places and rounding

Most Cryptography calculations are integer calculations and therefore are not meaningfully affected by decimal rounding. When a scalar numeric result is returned to the calculator, it is still passed through CCalc's global decimalPlaces rounding system. The setting is clamped to 0...18. Since the results are integers, the displayed value normally remains unchanged. Negative zero is normalized to 0.

Locale and separators

Integer input is locale-independent. The docklet does not accept decimal values, decimal separators, or list separators. Displayed multi-value results use semicolons between fields, for example: g=6; x=-1; y=2 Locale decimal formatting has no practical effect on integer-only outputs.

Angle-unit behavior

The Cryptography docklet does not use angle units.

Result behavior

Cryptography results fall into two visible categories. Scalar numeric results place a number into the calculator display and can continue into later calculator operations. These include modular arithmetic, GCD, modular inverse, LCM, totient, next prime, and XOR. Label-style results display a descriptive or multi-part result. These are not intended to continue as a simple scalar calculator value. Examples include extended GCD coefficients, factorization, CRT congruence, RSA output, Diffie–Hellman output, discrete logarithm explanation, and shift explanation. The ticker line shows the selected operation and its compact parameter summary. For label-style results, the display area shows the computed text result.

Result tags

[NUM] means the result is a scalar numeric result that can be used as a calculator value. [APPROX] means the result is approximate or numerical. The supplied Cryptography docklet does not use approximate successful results. [LABEL] means the result is descriptive, multi-value, symbolic, boolean-style, factorized, congruence-style, or otherwise non-scalar.

Function groups

The Cryptography docklet is organized into these groups:

Group 1 — Modular: Arithmetic, MOD, M+, M−, M×, MPOW

Group 2 — Core Number: Theory, GCD, XGCD, MINV, LCM, PHI

Group 3 — Primes & Factors: PRIME, NPRIME, FACT, CRT

Group 4 — Public-Key: RSA, DH, DLOG

Group 5 — Bitwise / Classical: Utilities, XOR, SHIFT

Function reference

MOD — Mod Reduce

Purpose: Reduces an integer modulo another integer.

Inputs: x, m

Parameters: x is the integer to reduce. m is the modulus.

Result: Returns the normalized residue: x mod |m| The result is normalized to the range 0...|m|-1.

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: m must not be 0. m must not be the minimum signed 64-bit integer.

Example: x = -3, m = 10 → 7.

M+ — Mod Add

Purpose: Adds two integers modulo m.

Inputs: a, b, m

Parameters: a and b are the integers to add. m is the modulus.

Result: Returns: (a + b) mod |m|

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: m must not be 0. m must not be the minimum signed 64-bit integer.

Example: a = 8, b = 7, m = 10 → 5.

M− — Mod Sub

Purpose: Subtracts two integers modulo m.

Inputs: a, b, m

Parameters: a is the starting value. b is the value subtracted from a. m is the modulus.

Result: Returns: (a − b) mod |m|

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: m must not be 0. m must not be the minimum signed 64-bit integer.

Example: a = 3, b = 8, m = 10 → 5.

M× — Mod Mul

Purpose: Multiplies two integers modulo m.

Inputs: a, b, m

Parameters: a and b are the integers to multiply. m is the modulus.

Result: Returns: (a × b) mod |m|

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: m must not be 0. m must not be the minimum signed 64-bit integer.

Example: a = 7, b = 8, m = 10 → 6.

MPOW — Mod Pow

Purpose: Computes modular exponentiation.

Inputs: a, e, m

Parameters: a is the base. e is the exponent. m is the modulus.

Result: Returns: a^e mod |m|

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: e must be greater than or equal to 0. m must not be 0. m must not be the minimum signed 64-bit integer.

Example: a = 4, e = 13, m = 497 → 445.

GCD — GCD

Purpose: Computes the greatest common divisor of two integers.

Inputs: a, b

Parameters: a and b are integers.

Result: Returns: gcd(a,b) as a non-negative integer.

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: Inputs must be valid signed 64-bit integers.

Example: a = 48, b = 18 → 6.

XGCD — Ext. GCD

Purpose: Computes the extended greatest common divisor.

Inputs: a, b

Parameters: a and b are integers.

Result: Returns a label-style result: g=<gcd>; x=<x>; y=<y> where: ax + by = gcd(a,b)

Result behavior: Label-style result. The output is descriptive and does not behave as a scalar calculator result.

Notes / restrictions: Inputs must be valid signed 64-bit integers. Very large combinations may be limited by signed integer arithmetic.

Example: a = 30, b = 12 → g=6; x=1; y=-2 or another valid coefficient pair satisfying the identity.

MINV — Mod Inverse

Purpose: Computes the modular inverse of a modulo m.

Inputs: a, m

Parameters: a is the integer to invert. m is the modulus.

Result: Returns x such that: a·x ≡ 1 (mod |m|)

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: m must not be 0. m must not be the minimum signed 64-bit integer. An inverse exists only when gcd(a,m) = 1.

Example: a = 3, m = 11 → 4, because 3×4 ≡ 1 mod 11.

LCM — LCM

Purpose: Computes the least common multiple of two integers.

Inputs: a, b

Parameters: a and b are integers.

Result: Returns: lcm(a,b)

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: If internal multiplication overflows, the function reports Overflow. If one input is zero, the result is 0.

Example: a = 12, b = 18 → 36.

PHI — Totient

Purpose: Computes Euler’s totient function.

Inputs: n

Parameters: n is an integer.

Result: Returns: φ(n) using the absolute value of n.

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: n must not be 0. φ(1) returns 1.

Example: n = 9 → 6.

PRIME — Prime Test

Purpose: Tests whether an integer is prime.

Inputs: n

Parameters: n is the integer to test.

Result: Returns one of: PRIME, COMPOSITE

Result behavior: Label-style result. The result is descriptive and does not behave as a scalar calculator value.

Notes / restrictions: The test uses the absolute value of n. Values below 2 are not prime.

Example: n = 97 → PRIME.

NPRIME — Next Prime

Purpose: Finds the next prime after the given input.

Inputs: n

Parameters: n is the starting integer.

Result: Returns the next prime greater than the effective starting value.

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: The search starts after max(n,1). If the search would exceed the signed 64-bit range, the function reports Overflow.

Example: n = 100 → 101.

FACT — Factor

Purpose: Prime-factorizes an integer.

Inputs: n

Parameters: n is the integer to factor.

Result: Returns a factorization string such as: 2^3 · 3 · 5

Result behavior: Label-style result. The result is descriptive and does not behave as a scalar calculator value.

Notes / restrictions: The function factors abs(n). For n = 0, it returns 0. For |n| = 1, it returns 1. If |n| > 1,000,000,000,000, the function reports n too large.

Example: n = 360 → 2^3 · 3^2 · 5.

CRT — CRT

Purpose: Solves a two-congruence Chinese remainder problem.

Inputs: a1, m1, a2, m2

Parameters: The function solves: x ≡ a1 (mod m1) x ≡ a2 (mod m2)

Result: Returns a congruence result: x≡r (mod l) where l is the least common multiple of the moduli.

Result behavior: Label-style result. The result is descriptive and does not behave as a scalar calculator value.

Notes / restrictions: m1 and m2 must not be 0. Neither modulus may be the minimum signed 64-bit integer. If the congruences are incompatible, the function reports No solution. If arithmetic overflows, the function reports Overflow.

Example: a1 = 2, m1 = 3, a2 = 3, m2 = 5 → x≡8 (mod 15).

RSA — RSA

Purpose: Performs a small RSA key and message calculation from two primes, a public exponent, and an integer message.

Inputs: p, q, e, m

Parameters: p and q are prime numbers. e is the public exponent. m is the integer message.

Result: Returns a multi-value label-style result: n=<n>; φ=<phi>; d=<d>; c=<cipher>; dec=<decrypted> where: n = p·q φ = (p−1)(q−1) d = e^{-1} mod φ c = m^e mod n dec = c^d mod n

Result behavior: Label-style result. The output is multi-value and does not behave as a scalar calculator value.

Notes / restrictions: p and q must be prime. p and q must be greater than 1. p and q must differ. e must be greater than 0. gcd(e,φ) must equal 1. The calculation is limited to signed 64-bit integer arithmetic and is not suitable for real RSA key sizes.

Example: p = 61, q = 53, e = 17, m = 65 returns RSA parameters including n = 3233, φ = 3120, and the encrypted/decrypted integer values.

DH — Diffie–Hellman

Purpose: Computes a small Diffie–Hellman exchange.

Inputs: p, g, a, b

Parameters: p is a prime modulus. g is the base or generator value. a is private exponent A. b is private exponent B.

Result: Returns: A=<A>; B=<B>; s=<shared>; check=<shared-check> where: A = g^a mod p B = g^b mod p s = B^a mod p check = A^b mod p

Result behavior: Label-style result. The output is multi-value and does not behave as a scalar calculator value.

Notes / restrictions: p must be prime and greater than 2. g must satisfy 1 ≤ g ≤ p−1. a and b must be greater than or equal to 0. This is a small-integer educational calculation, not a secure key-exchange implementation.

Example: p = 23, g = 5, a = 6, b = 15 → public values and matching shared secret.

DLOG — Discrete Log

Purpose: Searches for a discrete logarithm by brute force.

Inputs: p, g, y

Parameters: p is a prime modulus. g is the base. y is the target residue.

Result: Returns a statement of the form: g^x≡y (mod p) where x is the first exponent found.

Result behavior: Label-style result. The output is descriptive and does not behave as a scalar calculator value.

Notes / restrictions: p must be prime and greater than 2. g must satisfy 1 ≤ g ≤ p−1. p must be no greater than 1,000,000. If no exponent is found, the function reports No solution.

Example: p = 23, g = 5, y = 8 → finds an exponent x such that 5^x ≡ 8 mod 23.

XOR — XOR

Purpose: Computes bitwise exclusive OR.

Inputs: a, b

Parameters: a and b are signed 64-bit integers.

Result: Returns: a XOR b using the 64-bit bit patterns of the two inputs.

Result behavior: Scalar numeric result. The result can continue in calculator expressions.

Notes / restrictions: Signed values are interpreted through their 64-bit bit patterns. Negative inputs therefore behave according to two’s-complement-style bit representation.

Example: a = 12, b = 10 → 6.

SHIFT — Bit Shift

Purpose: Shifts an integer left or right.

Inputs: x, k

Parameters: x is the integer to shift. k is the shift amount.

Result: Returns a label-style result: r=<result> (k≥0: <<; k<0: >>)

Result behavior: Label-style result. The result is descriptive and does not behave as a scalar calculator value.

Notes / restrictions: If k ≥ 0, the function shifts left. If k < 0, the function shifts right. The shift magnitude is capped at 63. Right shift is performed on the signed integer value. Left shift uses the 64-bit bit pattern and then returns the result as a signed 64-bit integer.

Example: x = 3, k = 2 → r=12. x = 16, k = -2 → r=4.

Errors and limitations

Common error messages include: Invalid input m must be ≠ 0 m out of range e must be ≥ 0 No inverse (gcd ≠ 1), Overflow n must be ≠ 0 n too large moduli must be ≠ 0 modulus out of range No solution, p, q must be > 1 p,q must be prime p and q must differ e must be > 0 gcd(e,φ) ≠ 1 p must be > 2 p must be prime g must be in 1..p-1 a,b must be ≥ 0 p too large Not implemented

Important limitations: • All inputs are signed 64-bit integers. • The docklet does not support arbitrary-precision cryptography. • RSA, Diffie–Hellman, and discrete logarithm functions are small-integer tools, not secure cryptographic implementations. Factorization is capped at |n| ≤ 1,000,000,000,000. Discrete logarithm search is capped at p ≤ 1,000,000. Modular arithmetic rejects modulus 0. Some modulus operations reject Int64.min because its absolute value cannot be represented as a positive signed 64-bit integer. XGCD, CRT, RSA, DH, DLOG, FACT, PRIME, and SHIFT return label-style results, not scalar calculator values. Overflow can occur in LCM, CRT, RSA, and other integer operations involving large values. Prime testing and factorization use direct integer methods suitable for small and medium integers, not large cryptographic primes.

Practical usage notes

Use the modular arithmetic group for residue calculations: reduction, addition, subtraction, multiplication, and exponentiation. Use GCD, XGCD, and MINV when working with modular inverses or checking coprimality. MINV requires gcd(a,m) = 1. Use PHI, PRIME, NPRIME, and FACT for small integer number-theory checks. Use CRT when combining two congruences into one congruence. Check that the moduli and residues are compatible if No solution appears. Use RSA, DH, and DLOG as educational small-number tools. They are useful for verifying textbook examples and modular arithmetic behavior, not for real cryptographic security. Use XOR when a scalar bitwise result is needed. Use SHIFT when the direction of the shift should remain visible in the result label.

Appendix G.5 — Engineering and Applied Math

Overview

The Engineering docklet provides compact engineering calculations for decibels, first-order circuits, RLC resonance, second-order dynamics, rotational and linear motion, fluids, heat transfer, tolerance analysis, basic chemistry, electrical networks, wire gauge, and elementary mechanics of materials. The docklet is intended for users who need fast scalar engineering results with clear assumptions. It is not a simulation environment, unit-conversion engine, or symbolic derivation tool. Each function expects values in the units implied by the prompt labels and returns one scalar result.

Opening and using the docklet

The Engineering docklet appears as a parked, expandable panel. When parked, only the docklet header is visible. Tapping or dragging the header expands the list of Engineering functions. Selecting a function opens a firm prompt dialog. Enter the required values, then tap Compute. Tap Cancel to close the prompt without calculating. Tapping outside the prompt does not dismiss it. Outside taps trigger a warning response. This prevents accidental prompt closure while entering several parameters.

Current operand behavior

The Engineering docklet does not use the current calculator operand as a blank-field default. All fields required by the selected function must be entered manually. For most functions, every shown field must contain a valid number before Compute is enabled. For network equivalent functions using up to four parts, at least two valid positive component values are required. Extra component fields may be left blank. All successful Engineering functions return scalar numeric results. The returned value is placed into the calculator result area and can continue into later calculator operations.

Prompt dialogs

Prompt dialogs contain numeric fields only. There are no choice menus in this docklet.

General prompt behavior: • Required fields must contain valid numeric input. • For most functions, every field shown is required. • For network functions RΣ, R∥, CΣ, C∥, LΣ, and L∥, at least two non-empty positive values are required; remaining optional fields may be blank. Decimal commas are accepted while typing and normalized internally. Scientific notation using e is accepted by the sanitizer. Compute is disabled until the visible requirements are met. Cancel closes the prompt and performs no calculation. Outside taps do not dismiss the prompt.

Decimal places and rounding

Numeric results are rounded according to CCalc's global decimalPlaces setting. The setting is clamped to the range 0…18. A rounded negative zero is normalized to 0. Displayed values in labels may use compact formatting for very large or very small numbers. Very large or very small intermediate label values may appear in scientific notation.

Locale and separators

The docklet formats displayed decimal values using the selected locale. In Italian-style comma-decimal locale, displayed decimals use commas. In dot-decimal locale, displayed decimals use dots. Where several values are shown in a compact label, they are separated by semicolons.

Example: RLCQ[Q](10; 0.1; 0.00001) There are no CSV/list-entry fields in this docklet. Network functions use separate fields, not typed lists.

Result behavior

All Engineering functions return scalar numeric results. The result appears in the calculator display and can continue into calculator expressions. The ticker line shows the compact function label and, where present, the formula or assumption used. Some functions are marked approximate because they rely on empirical formulas, standard engineering approximations, inverse formulas, or numerical-style engineering conventions. Examples include overshoot relations, settling-time relations, AWG conversions, and derived inverse damping calculations.

Result tags

[NUM] means the result is a direct numeric result from the formula used. [APPROX] means the result is approximate or based on an engineering approximation, empirical relation, or rounded standard formula. [LABEL] is reserved for descriptive, symbolic, non-scalar, or error-style results. In this docklet, successful functions are scalar; errors are shown as messages rather than usable scalar values.

Function groups

The Engineering docklet is organized into these groups:

Group 1 — dB toolkit: dBp, dBa, r←dBp, r←dBa, dBsum

Group 2 — 1st-order RC/RL: core, RCτ, RLτ, RCfc, LP|H|, LPφ

Group 3 — RLC resonance +: Q (series), RLCf0, RLCQ

Group 4 — 2nd-order: dynamics, %OS, Ts(2%), Tp, ζ←OS, ωn←Ts

Group 5 — Kinematics +: rotational, rpm→ω, ω→rpm, P=τω, v=v0+at, s(t)

Group 6 — Fluids / thermo: q=½ρv², dP, Re, Qcond, Qconv

Group 7 — Geometry +: measurement, Worst, RSS, interp, %err
Group 8 — Chemistry-lite: dilute, idealP
Group 9 — EE basics: RΣ, R‖, CΣ, C‖, LΣ, L‖, Xc, Xl, VDIV, P=I2R, P=V2R, P=VI
Group 10 — Wire gauge: AWG→d, d→AWG, AWG→Ωpm
Group 11 — Mechanics of: Materials, σ=F/A, ΔL, δcant

Function reference

dBp — dB (Power ratio)

Purpose: Computes decibels from a power ratio.
Inputs: P2, P1
Parameters: P2 is the output or comparison power. P1 is the reference power.
Result: Returns: 10 log10(P2/P1) in dB.
Result behavior: Scalar numeric result.
Notes / restrictions: P1 and P2 must be greater than zero.
Example: P2 = 100, P1 = 10 → 10 dB.

dBa — dB (Amp. ratio)

Purpose: Computes decibels from an amplitude ratio.
Inputs: A2, A1
Parameters: A2 is the output or comparison amplitude. A1 is the reference amplitude.
Result: Returns: 20 log10(A2/A1) in dB.
Result behavior: Scalar numeric result.
Notes / restrictions: A1 and A2 must be greater than zero.
Example: A2 = 10, A1 = 1 → 20 dB.

r←dBp — Ratio ← dB (P)

Purpose: Converts a power decibel value back to a power ratio.
Inputs: dB
Parameters: dB is a power-ratio decibel value.
Result: Returns: 10^(dB/10)
Result behavior: Scalar numeric ratio.
Notes / restrictions: Input must be finite.
Example: dB = 20 → 100.

r←dBa — Ratio ← dB (A)

Purpose: Converts an amplitude decibel value back to an amplitude ratio.
Inputs: dB
Parameters: dB is an amplitude-ratio decibel value.
Result: Returns: 10^(dB/20)
Result behavior: Scalar numeric ratio.
Notes / restrictions: Input must be finite.
Example: dB = 20 → 10.

dBsum — dB sum

Purpose: Adds two dB values arithmetically.
Inputs: dB1, dB2
Parameters: dB1 and dB2 are decibel values.
Result: Returns: dB1 + dB2
Result behavior: Scalar numeric result.
Notes / restrictions: This is a simple arithmetic dB sum. It is not a power-domain logarithmic summation.
Example: dB1 = 3, dB2 = 6 → 9 dB.

RCτ — RC τ (t const)

Purpose: Computes the RC time constant.
Inputs: R, C
Parameters: R is resistance in ohms. C is capacitance in farads.
Result: Returns: τ = RC in seconds.
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite. The code does not require positive values for this function, but physical use normally requires positive R and C.
Example: R = 1000, C = 0.000001 → 0.001 s.

RLτ — RL τ (t const)
Purpose: Computes the RL time constant.
Inputs: L, R
Parameters: L is inductance in henries. R is resistance in ohms.
Result: Returns: τ = L/R in seconds.
Result behavior: Scalar numeric result.
Notes / restrictions: R must not be zero.
Example: L = 0.5, R = 10 → 0.05 s.

RCfc — RC $f_{(c)}$ (cutoff)
Purpose: Computes the RC cutoff frequency.
Inputs: R, C
Parameters: R is resistance in ohms. C is capacitance in farads.
Result: Returns: fc = 1/(2πRC) in hertz.
Result behavior: Scalar numeric result.
Notes / restrictions: R and C must be greater than zero.
Example: R = 1000, C = 0.000001 → approximately 159.154943 Hz.

LP|H| — LP |H| (mag)
Purpose: Computes the magnitude of a first-order low-pass response.
Inputs: f, fc
Parameters: f is frequency. fc is cutoff frequency.
Result: Returns: $|H| = 1 / \sqrt{(1 + (f/fc)^2)}$
Result behavior: Scalar numeric ratio.
Notes / restrictions: fc must be greater than zero.
Example: f = 1000, fc = 1000 → approximately 0.70710678.

LPφ — LP φ (phase, deg)
Purpose: Computes the phase angle of a first-order low-pass response.
Inputs: f, fc
Parameters: f is frequency. fc is cutoff frequency.
Result: Returns: φ = -atan(f/fc) converted to degrees.
Result behavior: Scalar numeric result in degrees.
Notes / restrictions: fc must be greater than zero. This function always returns degrees; it does not use the app’s angle-unit setting.
Example: f = fc → -45°.

RLCf0 — RLC f_0 (res freq)
Purpose: Computes the resonant frequency of an ideal LC circuit.
Inputs: L, C
Parameters: L is inductance in henries. C is capacitance in farads.
Result: Returns: f0 = 1/(2π√LC) in hertz.
Result behavior: Scalar numeric result.
Notes / restrictions: L and C must be greater than zero.
Example: L = 0.1, C = 0.00001 → approximately 159.154943 Hz.

RLCQ — RLC Q (quality fact)
Purpose: Computes the quality factor for a series RLC model.
Inputs: R, L, C
Parameters: R is resistance in ohms. L is inductance in henries. C is capacitance in farads.
Result: Returns: Q = (1/R)√(L/C)
Result behavior: Scalar numeric result.
Notes / restrictions: R must not be zero. L and C must be greater than zero.
Example: R = 10, L = 0.1, C = 0.00001 → 10.

%OS — % Overshoot (OS)
Purpose: Computes percent overshoot for an underdamped second-order system.
Inputs: ζ
Parameters: ζ is damping ratio.
Result: Returns: $100 \cdot e^{(-(\zeta\pi)/\sqrt{(1-\zeta^2)})}$ as a percent.
Result behavior: Scalar approximate result.
Notes / restrictions: Requires: $0<\zeta<1$

Example: ζ = 0.5 → approximately 16.303353%.

Ts(2%) — T_s (settle 2%)

Purpose: Estimates 2% settling time for a second-order system.
Inputs: ζ, ωn
Parameters: ζ is damping ratio. ωn is natural frequency in rad/s.
Result: Returns: Ts = 4/(ζωn) in seconds.
Result behavior: Scalar approximate result.
Notes / restrictions: ζ must be greater than zero. ωn must be greater than zero.
Example: ζ = 0.5, ωn = 10 → 0.8 s.

Tp — T_p (peak time)

Purpose: Computes peak time for an underdamped second-order system.
Inputs: ζ, ωn
Parameters: ζ is damping ratio. ωn is natural frequency in rad/s.
Result: Returns: Tp = π/(ωn√(1-ζ²)) in seconds.
Result behavior: Scalar approximate result.
Notes / restrictions: Requires: 0<ζ<1 ωn > 0
Example: ζ = 0.5, ωn = 10 → approximately 0.36275987 s.

ζ←OS — ζ ← %OS

Purpose: Computes damping ratio from percent overshoot.
Inputs: %OS
Parameters: %OS is percent overshoot.
Result: Returns damping ratio ζ.
Result behavior: Scalar approximate result.
Notes / restrictions: Requires: 0 < %OS < 100 The formula assumes an underdamped second-order response.
Example: %OS = 16.303353 → approximately ζ = 0.5.

ωn←Ts — ωn ← Ts

Purpose: Computes natural frequency from settling time and damping ratio.
Inputs: Ts, ζ
Parameters: Ts is 2% settling time in seconds. ζ is damping ratio.
Result: Returns: ωn = 4/(ζTs) in rad/s.
Result behavior: Scalar approximate result.
Notes / restrictions: Ts must be greater than zero. ζ must be greater than zero.
Example: Ts = 0.8, ζ = 0.5 → 10 rad/s.

rpm→ω — rpm → ω (rad/s)

Purpose: Converts rotational speed from revolutions per minute to angular velocity.
Inputs: rpm
Parameters: rpm is revolutions per minute.
Result: Returns: ω = 2π rpm / 60 in rad/s.
Result behavior: Scalar numeric result.
Notes / restrictions: Input must be finite.
Example: rpm = 60 → approximately 6.28318531 rad/s.

ω→rpm — ω → rpm

Purpose: Converts angular velocity to revolutions per minute.
Inputs: ω
Parameters: ω is angular velocity in rad/s.
Result: Returns: rpm = ω · 60/(2π)
Result behavior: Scalar numeric result.
Notes / restrictions: Input must be finite.
Example: ω = 6.28318531 → approximately 60 rpm. P=τω — P = τω (power)
Purpose: Computes rotational mechanical power.
Inputs: τ, ω
Parameters: τ is torque in N·m. ω is angular velocity in rad/s.
Result: Returns: P = τω in watts.
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite.
Example: τ = 10, ω = 20 → 200 W. v=v0+at — v = v_0 + at (lin)

Purpose: Computes final velocity under constant acceleration.
Inputs: v0, a, t
Parameters: v0 is initial velocity. a is acceleration. t is time.
Result: Returns: v = v0 + at
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite.
Example: v0 = 2, a = 3, t = 4 → 14.

s(t) — s(t) (const a)

Purpose: Computes position under constant acceleration.
Inputs: s0, v0, a, t
Parameters: s0 is initial position. v0 is initial velocity. a is acceleration. t is time.
Result: Returns: s = s0 + v0t + ½at²
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite.
Example: s0 = 0, v0 = 2, a = 3, t = 4 → 32. q=½ρv² — q (dyn press)
Purpose: Computes dynamic pressure.
Inputs: ρ, v
Parameters: ρ is density. v is velocity.
Result: Returns: q = ½ρv² in pascals when SI units are used.
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite.
Example: ρ = 1.225, v = 10 → 61.25 Pa.

dP — ΔP (Bernoulli)

Purpose: Computes simplified Bernoulli pressure difference from two speeds.
Inputs: ρ, v1, v2
Parameters: ρ is fluid density. v1 is initial speed. v2 is final speed.
Result: Returns: ΔP = ½ρ(v2² − v1²) in pascals when SI units are used.
Result behavior: Scalar numeric result.
Notes / restrictions: This is a simplified velocity-pressure relation and does not include elevation or loss terms.
Example: ρ = 1.225, v1 = 10, v2 = 20 → 183.75 Pa.

Re — Re (Reynolds)

Purpose: Computes Reynolds number.
Inputs: ρ, v, D, μ
Parameters: ρ is density. v is velocity. D is characteristic diameter or length. μ is dynamic viscosity.
Result: Returns: Re = ρvD/μ
Result behavior: Scalar numeric ratio.
Notes / restrictions: μ must not be zero.
Example: ρ = 1000, v = 1, D = 0.05, μ = 0.001 → 50000.

Qcond — Q̇ (conduction)

Purpose: Computes conductive heat-transfer rate.
Inputs: k, A, ΔT, L
Parameters: k is thermal conductivity. A is area. ΔT is temperature difference. L is thickness or conduction length.
Result: Returns: Q = kAΔT/L in watts when SI units are used.
Result behavior: Scalar numeric result.
Notes / restrictions: L must not be zero.
Example: k = 200, A = 0.01, ΔT = 10, L = 0.1 → 200 W.

Qconv — Q̇ (convection)

Purpose: Computes convective heat-transfer rate.
Inputs: h, A, ΔT
Parameters: h is convective heat-transfer coefficient. A is area. ΔT is temperature difference.
Result: Returns: Q = hAΔT in watts when SI units are used.
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite.
Example: h = 25, A = 2, ΔT = 10 → 500 W.

Worst — Worst-case (tol)

Purpose: Computes worst-case tolerance stack-up for two tolerances.

Inputs: t1, t2
Parameters: t1 and t2 are tolerance contributions.
Result: Returns: |t1| + |t2|
Result behavior: Scalar numeric result.
Notes / restrictions: Only two tolerance values are accepted.
Example: t1 = 0.1, t2 = -0.2 → 0.3.

RSS — RSS (tol)

Purpose: Computes root-sum-square tolerance for two tolerance values.
Inputs: t1, t2
Parameters: t1 and t2 are tolerance contributions.
Result: Returns: $\sqrt{(t1^2 + t2^2)}$
Result behavior: Scalar numeric result.
Notes / restrictions: Only two tolerance values are accepted.
Example: t1 = 3, t2 = 4 → 5.

interp — Interp (linear)

Purpose: Performs linear interpolation.
Inputs: x0, y0, x1, y1, x
Parameters: (x0, y0) and (x1, y1) define the line segment. x is the target x-value.
Result: Returns: $y = y0 + (x - x0)(y1 - y0)/(x1 - x0)$
Result behavior: Scalar numeric result.
Notes / restrictions: x1 must not equal x0.
Example: x0 = 0, y0 = 10, x1 = 100, y1 = 20, x = 50 → 15.

%err — % Error (meas)

Purpose: Computes percent error.
Inputs: meas, true
Parameters: meas is the measured value. true is the reference or true value.
Result: Returns: $100 \cdot (meas - true)/true$ as a percent.
Result behavior: Scalar numeric result.
Notes / restrictions: true must not be zero.
Example: meas = 102, true = 100 → 2%.

dilute — Dilution (V_2)

Purpose: Computes final volume in a dilution relation.
Inputs: M1, V1, M2
Parameters: M1 is initial concentration. V1 is initial volume. M2 is final concentration.
Result: Returns: V2 = M1V1/M2
Result behavior: Scalar numeric result.
Notes / restrictions: M2 must not be zero.
Example: M1 = 2, V1 = 1, M2 = 0.5 → 4.

idealP — Ideal gas (P)

Purpose: Computes pressure from the ideal gas law.
Inputs: n, T, V
Parameters: n is amount of substance in mol. T is temperature in kelvin. V is volume in m^3.
Result: Returns: P = nRT/V using: R = 8.314462618
Result behavior: Scalar numeric result in pascals when SI units are used.
Notes / restrictions: V must not be zero.
Example: n = 1, T = 273.15, V = 0.022414 → approximately atmospheric pressure.

RΣ — R eq Σ (series)

Purpose: Computes equivalent resistance for series resistors.
Inputs: R1, R2, optional R3, optional R4
Parameters: Each entered value is a resistance in ohms.
Result: Returns: Req = R1 + R2 + ...
Result behavior: Scalar numeric result in ohms.
Notes / restrictions: At least two positive values are required. Blank optional fields are ignored.
Example: R1 = 100, R2 = 200 → 300 Ω.

R‖ — R eq ‖ (parallel)
Purpose: Computes equivalent resistance for parallel resistors.
Inputs: R1, R2, optional R3, optional R4
Parameters: Each entered value is a resistance in ohms.
Result: Returns: Req = 1 / Σ(1/R)
Result behavior: Scalar numeric result in ohms.
Notes / restrictions: At least two positive values are required. Blank optional fields are ignored.
Example: R1 = 100, R2 = 100 → 50 Ω.

CΣ — C eq Σ (series)
Purpose: Computes equivalent capacitance for series capacitors.
Inputs: C1, C2, optional C3, optional C4
Parameters: Each entered value is a capacitance in farads.
Result: Returns: Ceq = 1 / Σ(1/C)
Result behavior: Scalar numeric result in farads.
Notes / restrictions: At least two positive values are required. Blank optional fields are ignored.
Example: C1 = 0.000001, C2 = 0.000001 → 0.0000005 F.

C‖ — C eq ‖ (parallel)
Purpose: Computes equivalent capacitance for parallel capacitors.
Inputs: C1, C2, optional C3, optional C4
Parameters: Each entered value is a capacitance in farads.
Result: Returns: Ceq = C1 + C2 + ...
Result behavior: Scalar numeric result in farads.
Notes / restrictions: At least two positive values are required. Blank optional fields are ignored.
Example: C1 = 0.000001, C2 = 0.000001 → 0.000002 F.

LΣ — L eq Σ (series)
Purpose: Computes equivalent inductance for series inductors.
Inputs: L1, L2, optional L3, optional L4
Parameters: Each entered value is an inductance in henries.
Result: Returns: Leq = L1 + L2 + ...
Result behavior: Scalar numeric result in henries.
Notes / restrictions: At least two positive values are required. Blank optional fields are ignored.
Example: L1 = 0.1, L2 = 0.2 → 0.3 H.

L‖ — L eq ‖ (parallel)
Purpose: Computes equivalent inductance for parallel inductors.
Inputs: L1, L2, optional L3, optional L4
Parameters: Each entered value is an inductance in henries.
Result: Returns: Leq = 1 / Σ(1/L)
Result behavior: Scalar numeric result in henries.
Notes / restrictions: At least two positive values are required. Blank optional fields are ignored.
Example: L1 = 0.1, L2 = 0.1 → 0.05 H.

Xc — Xc (react)
Purpose: Computes capacitive reactance.
Inputs: f, C
Parameters: f is frequency in hertz. C is capacitance in farads.
Result: Returns: Xc = 1/(2πfC) in ohms.
Result behavior: Scalar numeric result.
Notes / restrictions: f and C must be greater than zero.
Example: f = 159.154943, C = 0.00001 → approximately 100 Ω.

Xl — Xl (react)
Purpose: Computes inductive reactance.
Inputs: f, L
Parameters: f is frequency in hertz. L is inductance in henries.
Result: Returns: Xl = 2πfL in ohms.
Result behavior: Scalar numeric result.
Notes / restrictions: f and L must be greater than zero.
Example: f = 159.154943, L = 0.1 → approximately 100 Ω.

VDIV — V div (Vout)

Purpose: Computes voltage-divider output.
Inputs: Vin, Rtop, Rbot
Parameters: Vin is input voltage. Rtop is the upper resistor. Rbot is the lower resistor.
Result: Returns: Vout = Vin · Rbot/(Rtop + Rbot)
Result behavior: Scalar numeric result in volts.
Notes / restrictions: Rtop and Rbot must be greater than zero.
Example: Vin = 10, Rtop = 1000, Rbot = 1000 → 5 V. P=I2R — P = I^2R (power)
Purpose: Computes electrical power from current and resistance.
Inputs: I, R
Parameters: I is current in amperes. R is resistance in ohms.
Result: Returns: P = I^2R in watts.
Result behavior: Scalar numeric result.
Notes / restrictions: R must be greater than zero.
Example: I = 2, R = 10 → 40 W. P=V2R — P = V^2/R (power)
Purpose: Computes electrical power from voltage and resistance.
Inputs: V, R
Parameters: V is voltage. R is resistance.
Result: Returns: P = V^2/R in watts.
Result behavior: Scalar numeric result.
Notes / restrictions: R must be greater than zero.
Example: V = 10, R = 5 → 20 W. P=VI — P = VI (power)
Purpose: Computes electrical power from voltage and current.
Inputs: V, I
Parameters: V is voltage. I is current.
Result: Returns: P = VI in watts.
Result behavior: Scalar numeric result.
Notes / restrictions: Inputs must be finite.
Example: V = 12, I = 2 → 24 W.

AWG→d — AWG → d (mm)

Purpose: Converts AWG gauge to wire diameter.
Inputs: AWG
Parameters: AWG is the wire gauge number.
Result: Returns diameter in millimeters.
Result behavior: Scalar approximate result.
Notes / restrictions: The function accepts finite numeric AWG input. It does not require the value to be an integer for this diameter conversion.
Example: AWG = 18 → approximately 1.0237 mm.

d→AWG — d (mm) → AWG

Purpose: Converts wire diameter in millimeters to approximate AWG gauge.
Inputs: d
Parameters: d is wire diameter in millimeters.
Result: Returns approximate AWG gauge.
Result behavior: Scalar approximate result.
Notes / restrictions: d must be greater than zero.
Example: d = 1.0237 → approximately 18 AWG.

AWG→Ωpm — AWG → Ω/m

Purpose: Computes copper wire resistance per meter from AWG gauge.
Inputs: AWG
Parameters: AWG is integer wire gauge.
Result: Returns resistance in ohms per meter.
Result behavior: Scalar numeric result.
Notes / restrictions: AWG must be an integer from 0 through 40. The calculation assumes copper at 20°C using resistivity: ρ = 1.724e-8 Ω·m
Example: AWG = 18 → resistance per meter for 18 AWG copper wire. σ=F/A — σ = F/A
Purpose: Computes normal stress.
Inputs: F, A

Parameters: F is force in newtons. A is area in square meters.

Result: Returns: σ = F/A in pascals.

Result behavior: Scalar numeric result.

Notes / restrictions: A must not be zero.

Example: F = 1000, A = 0.01 → 100000 Pa.

ΔL — ΔL (axial)

Purpose: Computes axial elongation or shortening.

Inputs: F, L, A, E

Parameters: F is axial force. L is original length. A is cross-sectional area. E is Young’s modulus.

Result: Returns: ΔL = FL/(AE) in meters when SI units are used.

Result behavior: Scalar numeric result.

Notes / restrictions: A must not be zero. E must not be zero.

Example: F = 1000, L = 2, A = 0.001, E = 200000000000 → 0.00001 m.

δcant — δ (cantilever)

Purpose: Computes end deflection for a cantilever beam with point load using the standard formula.

Inputs: P, L, E, I

Parameters: P is load. L is beam length. E is Young’s modulus. I is second moment of area.

Result: Returns: $\delta = PL^3/(3EI)$ in meters when SI units are used.

Result behavior: Scalar numeric result.

Notes / restrictions: E must not be zero. I must not be zero. This is the simple cantilever point-load formula; boundary conditions and load placement are assumed by the formula.

Example: P = 100, L = 1, E = 200000000000, I = 0.000001 → approximately 0.00016667 m.

Errors and limitations

Common error messages include: Invalid input, Invalid result, Enter at least two positive values P1 and P2 must be > 0 A1 and A2 must be > 0 R must be ≠ 0 R and C must be > 0 fc must be > 0 L and C must be > 0 Requires 0 < ζ < 1 ζ must be > 0 ωn must be > 0 Requires 0 < %OS < 100 Ts must be > 0 μ must be ≠ 0 L must be ≠ 0 x1 must be ≠ x0 true must be ≠ 0 M2 must be ≠ 0 V must be ≠ 0 Rtop/Rbot must be > 0 R must be > 0 d must be > 0 AWG must be integer AWG out of range A must be ≠ 0 E must be ≠ 0 I must be ≠ 0 Not implemented

Important limitations: The docklet does not perform automatic unit conversion. Values must be entered in the units implied by the field labels. Low-pass phase is always returned in degrees. dBsum is arithmetic addition, not logarithmic power addition. Ts(2%), %OS, Tp, ζ←OS, and ωn←Ts use standard second-order approximation formulas. AWG→Ωpm assumes copper at 20°C. Network equivalent functions accept up to four component values. Worst and RSS accept exactly two tolerance values. δcant assumes the standard cantilever point-load relation $PL^3/(3EI)$. Numerical overflow or invalid floating-point results are reported as invalid results. Successful outputs are scalar; the docklet does not return matrices, curves, symbolic derivations, or multi-line reports.

Practical usage notes

Use the dB tools when converting between ratios and decibel expressions. Use dBp for power ratios and dBa for amplitude ratios. Use RCτ, RLτ, RCfc, LP|H|, and LPφ for simple first-order circuit work. The inputs must already be in compatible SI units if SI output is expected. Use RLCf0 and RLCQ for ideal series RLC estimates. These are compact formulas, not circuit simulations. Use the second-order dynamics tools for standard underdamped control-system estimates. The damping ratio restrictions matter: overshoot and peak-time formulas require 0 < ζ < 1. Use RΣ, R‖, CΣ, C‖, LΣ, and L‖ for quick two-to-four component equivalents. Leave unused optional fields blank. Use the mechanics tools only with consistent units. For SI use: newtons, meters, square meters, pascals, and meters to obtain pascal or meter outputs.

Appendix G.6 — Equation-Solving

Overview

The Equations docklet provides algebra and equation-solving tools for linear equations, quadratic equations, cubic equations, polynomial evaluation, factoring aids, simplification tools, systems of equations, and numerical solving. It is intended for users who need fast algebraic results without manually rebuilding each formula from the main calculator keypad. Some functions return a single numeric value that can continue into later calculator operations. Other functions return structured equation results, such as roots, factor forms, systems solutions, or symbolic-style expressions. Those structured results are shown as labels rather than ordinary scalar calculator values. The docklet works with decimal numeric input. Integer-only tools require integer-valued input even though the prompt field accepts typed decimal characters.

Opening and using the docklet

The Equations docklet appears as a parked, expandable panel. When parked, only the compact docklet header is visible. Tapping or dragging the header expands the list of equation functions. Selecting a function opens a firm prompt dialog. The dialog contains the fields needed by that function. After entering the values, tap Compute. Tap Cancel to close the dialog without calculating. Tapping outside the dialog does not dismiss it. Outside taps produce a warning response. This prevents accidental loss of entered coefficients.

Current operand behavior

The Equations docklet does not use the current calculator operand as a blank-field default in the supplied code. All prompt fields required by the selected function must be entered manually. The current operand may be replaced by a scalar result when the function returns a numeric value, but it is not automatically inserted into blank fields. For structured results, such as x=(1; 2), V=(h; k), factor forms, or systems solutions, the result is shown as a label-style result rather than as an ordinary scalar operand.

Prompt dialogs

All Equations prompt dialogs use numeric text fields. The input sanitizer accepts digits, decimal separators, signs, and scientific notation using e. It also normalizes comma decimal input to dot decimal internally. Prompt behavior: Compute is enabled only when all required fields contain valid numeric input. Cancel closes the prompt and marks the selected function as cancelled. Outside tap does not close the prompt. Keyboard Next/Done moves through fields or dismisses the keyboard. There are no choice menus in this docklet. There is no blank-field operand behavior in this docklet. For SYS, the first field n controls the system size. Entering 2 shows a 2×2 system; entering 3 expands the prompt to a 3×3 system. The prompt updates dynamically when n is changed.

Decimal places and rounding

Numeric scalar results are rounded according to CCalc's global decimalPlaces setting. The setting is clamped to the range 0...18. Very small signed zero results are normalized to 0. Some displayed double-precision values also snap near-integers to clean integer form when they are within a small numerical tolerance. Structured label results use the same displayed precision rules where numeric components are formatted into the result text.

Locale and separators

The docklet respects the selected numeric locale when formatting displayed numbers. In comma-decimal locale, displayed decimal values use a comma. In dot-decimal locale displayed decimal values use a dot. For multi-value outputs, the docklet uses semicolon-style separation: x=(1; 2) V=(h; k) x=1; y=2 Typed decimal commas are accepted and normalized internally. There are no list-entry fields such as CSV cashflows in this docklet.

Result behavior

The docklet produces two broad result types. A scalar result is a single numeric value placed into the calculator display and can continue into later calculator expressions. Examples include the linear root, discriminant, polynomial evaluation, rationalized numeric value, and binomial expansion numeric evaluation. A label result is a structured result intended for reading rather than continuing as a normal scalar calculation. Examples include quadratic roots, cubic roots, vertex coordinates, complete-square form, synthetic division output, factor forms, systems solutions, and Newton solver reports. The ticker line shows the operation label or compact formula context. It does not need to repeat the full numeric operand when the result already appears in the main display or result label.

Result tags

The docklet uses result tags to clarify how the output should be interpreted. [NUM] means the result is numeric and exact or direct within the formula used. [APPROX] means the result is approximate or numerically computed. [LABEL] means the result is descriptive, structured, symbolic-style, complex, multi-value, or otherwise not a simple scalar. A result tagged [LABEL] should be read as an output report rather than treated as an ordinary number.

Function groups

The Equations docklet is organized into these logical groups:

Group 1 — Equation Solvers: LIN, QUAD, CUBIC

Group 2 — Quadratic Tools: Δ, V(h, k), x=h, a(x-h)²+k

Group 3 — Polynomial Tools: P(x), r→c, syndiv, p/q, ±roots

Group 4 — Factoring: factor, GCF, a²-b²

Group 5 — Simplification: √simp, 1/√, expand

Group 6 — Systems: SYS, X-SYS

Group 7 — Numerical Solvers: SOLVE

Function reference

LIN — Linear (ax+b=0)

Purpose: Solves a linear equation of the form: ax + b = 0

Inputs: a, b

Parameters: a is the coefficient of x. b is the constant term.

Result: Returns: x = -b/a

Result behavior: The numeric root is placed into the calculator as a scalar result. The displayed label shows the root as x=(...).

Notes / restrictions: a must not be zero.

Example: For a = 2, b = -8: 2x - 8 = 0 x=4

QUAD — Quadratic

Purpose: Solves a quadratic equation: $ax^2 + bx + c = 0$

Inputs: a, b, c

Parameters: a is the quadratic coefficient. b is the linear coefficient. c is the constant term.

Result: Returns the root or roots. Complex conjugate roots are shown using i.

Result behavior: The result is a label-style output. It may show one root, two real roots, two complex roots, all real solutions, or no solution depending on the coefficients.

Notes / restrictions: If a = 0, the function reduces to a linear equation. If a = 0 and b = 0, the function reports either all real numbers or no solution depending on c.

Example: For a = 1, b = -3, c = 2: x=(1; 2)

CUBIC — Cubic

Purpose: Solves a cubic equation: $ax^3 + bx^2 + cx + d = 0$

Inputs: a, b, c, d

Parameters: a is the cubic coefficient. b is the quadratic coefficient. c is the linear coefficient. d is the constant term.

Result: Returns real roots, or one real root plus a complex conjugate pair.

Result behavior: The result is a label-style output. It can show multiple roots separated by semicolons. Complex roots are shown as real ± imaginary i.

Notes / restrictions: If a = 0, the solver reduces the equation to a quadratic, linear, or constant case. The cubic solver uses numerical methods with bracketing, refinement, and deflation. Multiple roots and near-multiple roots are handled with tolerance-based polishing, but results remain numerical.

Example: For a = 1, b = -6, c = 11, d = -6: x=(1; 2; 3)

Δ — Discriminant Δ

Purpose: Computes the quadratic discriminant: $\Delta = b^2 - 4ac$

Inputs: a, b, c

Parameters: Coefficients of $ax^2 + bx + c$.

Result: Returns the discriminant as a scalar number.

Result behavior: The result flows back into the calculator as a numeric scalar.

Notes / restrictions: The discriminant is interpreted in the usual quadratic sense: positive for two real roots, zero for a repeated real root, negative for complex conjugate roots.

Example: For a = 1, b = -3, c = 2: Δ=1

V(h,k) — Vertex

Purpose: Computes the vertex of a quadratic function: $y = ax^2 + bx + c$

Inputs: a, b, c

Parameters: a, b, and c are the quadratic coefficients.

Result: Returns the vertex: V=(h; k) where: $h = -b / (2a)$ $k = ah^2 + bh + c$

Result behavior: The result is a label-style coordinate pair.

Notes / restrictions: a must not be zero.

Example: For a = 1, b = -4, c = 3: V=(2; -1) x=h — Axis of Symmetry

Purpose: Computes the axis of symmetry of a quadratic function.

Inputs: a, b, c

Parameters: a, b, and c are the coefficients of $ax^2 + bx + c$. The value of c is accepted for consistency with the quadratic tools, but the axis depends only on a and b.

Result: Returns: $x = -b / (2a)$

Result behavior: The result is shown as a label-style equation such as x=(2).

Notes / restrictions: a must not be zero.

Example: For a = 1, b = -4, c = 3: x=(2)

a(x-h)²+k — Complete Square

Purpose: Converts a quadratic expression into vertex form: $a(x-h)^2 + k$

Inputs: a, b, c

Parameters: Coefficients of: $ax^2 + bx + c$

Result: Returns the completed-square form.

Result behavior: The result is a label-style algebraic expression.

Notes / restrictions: a must not be zero. The function computes h and k numerically, so irrational or repeating values are shown in rounded decimal form.

Example: For a = 1, b = -4, c = 3: 1·(x-2)²-1

P(x) — Evaluate P(x)

Purpose: Evaluates a quadratic polynomial at a selected value of x.

Inputs: a, b, c, x

Parameters: a, b, and c define: P(x) = ax² + bx + c x is the evaluation point.

Result: Returns the numeric value of P(x).

Result behavior: The result flows back into the calculator as a scalar numeric result.

Notes / restrictions: Only quadratic polynomial evaluation is implemented in this function.

Example: For a = 1, b = 2, c = 3, x = 4: P(4) = 27

r→c — Roots → Coeffs

Purpose: Builds quadratic coefficients from two roots and a leading coefficient.

Inputs: a, r1, r2

Parameters: a is the leading coefficient. r1 and r2 are the roots.

Result: Returns coefficients for: a(x-r1)(x-r2) Expanded as: ax² + bx + c where: b = -a(r1+r2) c = a(r1r2)

Result behavior: The result is a label-style coefficient report.

Notes / restrictions: a must not be zero.

Example: For a = 1, r1 = 2, r2 = 5: a=1; b=-7; c=10

syndiv — Synthetic Division

Purpose: Performs synthetic division of a cubic polynomial by (x-r).

Inputs: a, b, c, d, r

Parameters: a, b, c, and d define: ax³ + bx² + cx + d r is the divisor root for (x-r).

Result: Returns the quotient coefficients and remainder.

Result behavior: The result is a label-style output: q=(qa; qb; qc) rem=...

Notes / restrictions: a must not be zero. Inputs must be finite decimals.

Example: For x³ - 6x² + 11x - 6 divided by x - 1: q=(1; -5; 6) rem=0

p/q — Rational Root Test

Purpose: Tests possible rational roots for a quadratic polynomial with integer coefficients.

Inputs: a, b, c

Parameters: Integer coefficients of: ax² + bx + c

Result: Reports rational roots if found, or shows a list of candidate rational roots.

Result behavior: The result is a label-style output.

Notes / restrictions: All coefficients must be integer-valued. a must not be zero. If c = 0, the function reports 0 and the factor-candidate structure rather than enumerating all candidates normally.

Example: For a = 1, b = -3, c = 2: Rational roots: 1; 2

±roots — Descartes' Signs

Purpose: Applies Descartes' rule of signs to estimate possible positive and negative real root counts for a cubic polynomial.

Inputs: a, b, c, d

Parameters: Coefficients of: ax³ + bx² + cx + d

Result: Reports possible counts of positive and negative real roots.

Result behavior: The result is a label-style output.

Notes / restrictions: Zero coefficients are ignored when counting sign changes. The result gives possible counts, not actual roots.

Example: For signs + - + -: +roots: 3/1; −roots: 0

factor — Factor (simple)

Purpose: Attempts simple real factorization of a quadratic polynomial.

Inputs: a, b, c

Parameters: Coefficients of: ax² + bx + c

Result: Returns a simple factor form when the roots and leading coefficient are close to integers.

Result behavior: The result is a label-style algebraic expression.

Notes / restrictions: This is a simple factoring tool, not a full symbolic algebra engine. If the roots are complex or not clean integer-style roots, it reports no simple factorization.

Example: For a = 1, b = -3, c = 2: (x-1)(x-2)

GCF — GCF Extract

Purpose: Extracts the greatest common factor from integer quadratic coefficients.

Inputs: a, b, c

Parameters: Integer coefficients of: $ax^2 + bx + c$

Result: Returns: g·(reduced polynomial)

Result behavior: The result is a label-style algebraic expression.

Notes / restrictions: All coefficients must be integer-valued.

Example: For a = 6, b = 12, c = 18: $6\cdot(x^2 + 2x + 3)$

a²-b² — Difference Squares

Purpose: Applies the difference-of-squares identity: $a^2 - b^2 = (a-b)(a+b)$

Inputs: a, b

Parameters: a and b are the two terms in the identity.

Result: Returns the two factors.

Result behavior: The result is a label-style expression.

Notes / restrictions: The function computes the numeric values a-b and a+b.

Example: For a = 5, b = 3: (2)(8)

√simp — Simplify Radical

Purpose: Simplifies an integer square root by extracting square factors.

Inputs: n

Parameters: n is an integer radicand.

Result: Returns a simplified radical form.

Result behavior: The result is a label-style expression.

Notes / restrictions: n must be an integer. Negative radicands are expressed using i.

Example: For n = 72: 6√2 For n = -72: 6√2i

1/√ — Rationalize

Purpose: Computes and displays the rationalized form of: $1 / \sqrt{n}$

Inputs: n

Parameters: n is an integer radicand.

Result: Returns a numeric approximation while showing the rationalized form.

Result behavior: The numeric approximation flows back as a scalar approximate result. The label shows either 1/k for perfect squares or √n/n for non-square positive integers.

Notes / restrictions: n must be an integer. n must not be zero. n must not be negative.

Example: For n = 2: √2/2 with numeric result approximately: 0.70710678

expand — Expand (a+b)ⁿ

Purpose: Evaluates the expansion value of: $(a+b)^n$

Inputs: a, b, n

Parameters: a and b are decimal values. n is an integer exponent.

Result: Returns the numeric value of $(a+b)^n$.

Result behavior: The numeric value flows back as an approximate scalar result.

Notes / restrictions: n must be an integer from 0 through 12. The function returns the evaluated numeric result, not the full symbolic expanded polynomial.

Example: For a = 2, b = 3, n = 4: (2+3)^4 = 625

SYS — System 2x2 / 3x3

Purpose: Solves a linear system of two or three equations.

Inputs: For n = 2: a11, a12, a21, a22, b1, b2 For n = 3: a11, a12, a13, a21, a22, a23, a31, a32, a33, b1, b2, b3

Parameters: n selects system size: 2 or 3. aij are matrix coefficients. b1, b2, and b3 are right-hand-side values.

Result: Returns the solution vector. For a 2×2 system: x=...; y=... For a 3×3 system: x=...; y=...; z=...

Result behavior: The result is a terminal label-style output.

Notes / restrictions: The system must have a unique solution. Singular or dependent systems report no unique solution.

Example: For: x+y=5 x-y=1 the result is: x=3; y=2

X-SYS — Non-Linear Intersect

Purpose: Finds a numerical intersection between a line and a sine function using Newton’s method. The line is: $y = mx + b$ The sine function is: $y = A \sin(\omega x + \varphi)$

Inputs: m, b, A, ω, φ, x0, tol, iter

Parameters: m is the line slope. b is the line intercept. A is sine amplitude. ω is angular frequency. φ is phase. x0 is the initial guess. tol is the convergence tolerance. Default: 0.000001. iter is the maximum iteration count. Default: 40.

Result: Returns: x=...; y=...; iter=...

Result behavior: The result is a terminal label-style numerical report.

Notes / restrictions: tol must be greater than zero. iter must be greater than zero. The result depends on the initial guess. Different guesses may converge to different intersections or fail to converge.

Example: For a line and sine curve near a known crossing, enter an initial guess close to the expected intersection: x=...; y=...; iter=...

SOLVE — Newton Solver

Purpose: Finds a numerical root of a polynomial up to cubic degree using Newton's method. The polynomial is: $a3x^3 + a2x^2 + a1x + a0 = 0$

Inputs: a3, a2, a1, a0, x0, tol, iter

Parameters: a3 is the cubic coefficient. a2 is the quadratic coefficient. a1 is the linear coefficient. a0 is the constant term. x0 is the initial guess. tol is the convergence tolerance. Default: 0.000001. iter is the maximum iteration count. Default: 40.

Result: Returns: x≈...; f(x)≈...; iter=...

Result behavior: The result is a terminal label-style approximate numerical report.

Notes / restrictions: tol must be greater than zero. iter must be greater than zero. At least one of a3, a2, or a1 must be nonzero. The method may fail if the derivative is too close to zero, if the initial guess is poor, or if convergence is not reached within the iteration limit.

Example: For: $x^2 - 2 = 0$ enter: a3=0, a2=1, a1=0, a0=-2, x0=1, tol=0.000001, iter=40

Result: x≈1.41421356; f(x)≈0; iter=...

Errors and limitations

The Equations docklet validates required fields before computing. Common error messages include: Invalid input a must be ≠ 0 b must be ≠ 0 Requires integer coefficients, Requires integer n n must be integer n must be ≥ 0 n must be ≠ 0 n too large (max 12) No real solution, No simple factor No real factorization No unique solution, No convergence tol must be > 0 iter must be > 0 Not an equation

Important limitations: P(x) evaluates only quadratic polynomials. factor performs simple real factorization and is not a full symbolic factoring engine. p/q requires integer coefficients. GCF requires integer coefficients. √simp and 1/√ require integer n. expand evaluates (a+b)^n; it does not print the symbolic binomial expansion. expand is capped at n ≤ 12. SYS supports only 2×2 and 3×3 systems. SYS requires a unique solution. X-SYS and SOLVE are Newton-based numerical tools; convergence is not guaranteed. Cubic solving is numerical and uses tolerances. Very close or repeated roots may be displayed using tolerance-based simplification. Complex results are displayed for quadratic and cubic solving where supported, but the docklet is not a general complex algebra system.

Practical usage notes

Use LIN, QUAD, and CUBIC when the goal is to solve an equation directly. Use Δ, V(h,k), x=h, and $a(x-h)^2+k$ when studying the structure of a quadratic rather than simply finding its roots. Use P(x) when you need a scalar polynomial value that can continue into another calculator operation. Use r→c, syndiv, p/q, and ±roots when analyzing polynomial structure, possible roots, or division behavior. Use factor, GCF, and a^2-b^2 for simple algebraic rewriting. These tools are intentionally limited and should not be read as full symbolic algebra. Use SYS for exact-size linear systems with a unique solution. Use X-SYS and SOLVE when a numerical approximation is acceptable and an initial guess is available. For Newton-based functions, choose an initial guess close to the expected solution.

Appendix G.7 — Financial

Overview

The Financials docklet provides time-value-of-money tools, interest and growth calculations, investment appraisal functions, loan and pricing utilities, and consumer quick tools. It is intended for users who need compact financial calculations without building a full spreadsheet. Inputs are entered through prompt dialogs. Most results are scalar numeric values that become the current calculator result and can continue into later calculations. The Financials docklet exposes 24 functions: PMT, PV, FV, RATE, NPER, APY, CAGR, Δ%, R72, SI, CI, INFL, NPV, IRR, MIRR, PI AMORT, BAL, MARG, BEP, ROI, DISC, TAX, TIP

Opening and using the docklet

The Financials docklet appears as a parked or expandable panel. When parked, the header shows: Financials When expanded, the user can scroll through the available financial functions. Selecting a function opens a prompt dialog with the fields required for that calculation. The prompt is firm. Tapping outside the prompt does not dismiss it; it gives warning feedback. Use Cancel to leave the prompt without computing. Compute becomes available only when the required fields contain valid values.

Current operand behavior

The Financials docklet does not use blank fields as a shortcut for the current calculator operand. Every required field must be entered in the prompt. Optional fields are explicitly marked as optional, such as: Split (optional) Some fields have default values, but these are prompt defaults, not automatic current-operand insertion. Examples: FV defaults to 0 in PMT, PV, RATE, and NPER. Payment timing defaults to End. Compounds / year defaults to 12. Split defaults to 1. Balance after payment # defaults to 0 in AMORT.

Prompt dialogs

Financials prompts use numeric text fields and, where needed, choice menus. Required fields All fields must be filled unless the field title says optional. Examples of required fields: PV, FV, PMT, Rate % / period, NPER, Principal, Years, Cashflows CSV Optional fields Only fields whose title contains “optional” may be left blank. In the supplied code, this applies to: Split (optional) If blank, the split value is treated as 1. Choice fields Some prompts use menu choices. Payment timing choices: End, Begin Internally this corresponds to ordinary annuity vs annuity due behavior: End = payments at the end of each period Begin = payments at the beginning of each period APY conversion choices: APR → APY APY → APR Rule 72 calculation choices: Years from rate % Rate % from years Inflation Adj. conversion choices: Nominal → Real Real → Nominal Compute behavior Compute validates the fields and either produces a result or shows an error message. If the input is invalid, the dialog remains open and the error is shown in the prompt. Cancel behavior Cancel closes the prompt and does not compute. Keyboard behavior Numeric fields accept digits, a leading minus sign, and one decimal separator. Cashflow fields also allow list separators.

Decimal places and rounding

Financial results are rounded according to the app’s Decimal Places setting. The docklet clamps rounding to: 0…18 decimal places The docklet also snaps values very close to zero or very close to an integer. This avoids display artifacts such as: -0 1.9999999999 0.0000000001 These may be displayed as: 020

Locale and separators

The Financials docklet accepts both dot and comma decimal input in ordinary numeric fields. Examples: 12.5, 12,5 Both are parsed as the same decimal value. Cashflow lists use a locale-dependent list separator. In dot-decimal locales: CF0, CF1, CF2 ...

Example: -1000, 300, 400, 500 In comma-decimal locales: CF0; CF1; CF2 ...

Example: -1000; 300; 400; 500 This avoids conflict between comma-as-decimal and comma-as-list-separator.

Angle-unit behavior

The Financials docklet does not use angle units. The active angle setting has no effect on any Financials function.

Currency/base-currency behavior

The Financials docklet labels money-related results using the app’s selected base currency code. The default base currency is: USD If the app’s base currency is changed, money labels reflect that code. Examples: PMT[USD], PV[USD], FV[USD], NPV[USD], AMORT[USD], DISC[USD], TAX[USD], TIP[USD] The Financials docklet does not fetch exchange rates and does not convert currencies. The currency code is a label for the monetary unit used in the calculation.

Result behavior

Most Financials functions return scalar numeric results. These become the current calculator value and can continue into later calculations. Examples: PMT, PV, FV, RATE, NPER, APY, CAGR, Δ%, R72, CI, NPV, IRR, MIRR, PI, BAL, MARG ROI, DISC, TAX, TIP Some functions show a richer result line with multiple values, while still returning one primary scalar value. Examples: SI returns the total amount, while also showing interest and total. AMORT returns the payment, while also showing balance and approximate total interest. BEP returns units, while also showing revenue. TIP returns total, while also showing tip, total, and each-person amount.

Result tags

The Financials docklet uses result tags in the result line. [NUM] [NUM] means the result is numeric and can be used as the next calculator value. Most Financials results use [NUM].

Example: ROI%(Vi:1000, Vf:1350) [NUM], [APPROX] [APPROX] is available as part of the docklet’s result system, but the Financials functions in this supplied code generally tag financial outputs as numeric. Some methods are numerical or approximate in practice, such as RATE, IRR, and MIRR, but they are displayed as numeric results in this implementation. [LABEL] [LABEL] is available for descriptive or non-scalar output, but this Financials implementation primarily returns numeric financial results and error messages.

Function groups

The Financials docklet is organized into these logical groups.

Group 1 — Time Value of: Money, PMT, PV, FV, RATE, NPER

Group 2 — Interest & Growth: APY, CAGR, Δ%, R72, SI, CI, INFL

Group 3 — Investment: Appraisal, NPV, IRR, MIRR, PI

Group 4 — Loans & Pricing: AMORT, BAL, MARG, BEP, ROI

Function reference

PMT — Payment

Purpose: Computes the periodic payment for a loan, annuity, or savings target.

Inputs: PV, FV, Rate % / period, NPER, Payment timing Prompt defaults: FV = 0 Payment timing = End

Parameters: PV is present value. FV is future value. Rate % / period is the periodic rate as a percentage. NPER is the number of periods. Payment timing is End or Begin.

Result: Periodic payment. For zero rate: PMT = −(PV + FV) / NPER For nonzero rate: PMT = −((PV(1+r)^n + FV)r / ((1+r)^n − 1)) If payments occur at the beginning, the payment is adjusted by dividing by (1+r).

Result behavior: Scalar money result.

Notes / restrictions: NPER must be > 0 Possible errors: Invalid input NPER must be > 0 Invalid rate/periods

Example: PV = 10000 FV = 0 Rate % / period = 1 NPER = 36 Payment timing = End PMT ≈ -332.143

PV — Present Value

Purpose: Computes present value from payment, future value, rate, periods, and payment timing.

Inputs: PMT, FV, Rate % / period, NPER, Payment timing Prompt defaults: FV = 0 Payment timing = End

Parameters: PMT is periodic payment. FV is future value. Rate % / period is the periodic rate as a percentage. NPER is number of periods. Payment timing is End or Begin.

Result: Present value. For zero rate: PV = −(PMT × NPER + FV)

Result behavior: Scalar money result.

Notes / restrictions: NPER must be ≥ 0 Possible errors: Invalid input NPER must be ≥ 0

Example: PMT = -300 FV = 0 Rate % / period = 1 NPER = 36 Payment timing = End PV ≈ 9032.33

FV — Future Value

Purpose: Computes future value from present value, payment, rate, periods, and payment timing.

Inputs: PV, PMT, Rate % / period, NPER, Payment timing Prompt defaults: PMT = 0 Payment timing = End

Parameters: PV is present value. PMT is periodic payment. Rate % / period is the periodic rate as a percentage. NPER is number of periods. Payment timing is End or Begin.

Result: Future value. For zero rate: FV = −(PV + PMT × NPER)

Result behavior: Scalar money result.

Notes / restrictions: NPER must be ≥ 0 Possible errors: Invalid input NPER must be ≥ 0

Example: PV = -1000 PMT = -100 Rate % / period = 1 NPER = 12 Payment timing = End FV ≈ 2380.93

RATE — Rate

Purpose: Computes the periodic rate that satisfies a present value, future value, payment, and number of periods.

Inputs: PV, FV, PMT, NPER, Payment timing Prompt defaults: FV = 0 Payment timing = End

Parameters: PV is present value. FV is future value. PMT is periodic payment. NPER is number of periods. Payment timing is End or Begin.

Result: Rate per period, expressed as a percent.

Result behavior: Scalar numeric percent result.

Notes / restrictions: The calculation searches numerically for a rate. A valid bracket must exist. Possible errors: Invalid input NPER must be > 0 No bracket for RATE

Example: PV = 10000 FV = 0 PMT = -332.143 NPER = 36 Payment timing = End RATE ≈ 1 % / period

NPER — Periods

Purpose: Computes the number of periods required for a present value, future value, payment, and rate.

Inputs: PV, FV, PMT, Rate % / period, Payment timing Prompt defaults: FV = 0 Payment timing = End

Parameters: PV is present value. FV is future value. PMT is periodic payment. Rate % / period is the periodic rate as a percentage. Payment timing is End or Begin.

Result: Number of periods. For zero rate: NPER = −(PV + FV) / PMT

Result behavior: Scalar numeric result.

Notes / restrictions: If rate is zero, payment cannot be zero. For nonzero rate, the logarithmic solution must be real. Possible errors: Invalid input PMT cannot be 0 when rate is 0 No real solution

Example: PV = 10000 FV = 0 PMT = -300 Rate % / period = 1 Payment timing = End NPER ≈ 39.29

APY — APR ↔ APY

Purpose: Converts between nominal APR and effective APY.

Inputs: Conversion, Value %, Compounds / year Prompt defaults: Conversion = APR → APY Compounds / year = 12 Choice options: APR → APY APY → APR

Parameters: Value % is APR or APY depending on mode. Compounds / year is compounding frequency.

Result: Converted percentage. APR to APY: APY = (1 + APR/m)^m − 1 APY to APR: APR = m((1 + APY)^(1/m) − 1)

Result behavior: Scalar percent result.

Notes / restrictions: Compounds / year must be > 0 Possible errors: Invalid input Compounds must be > 0 Invalid APY

Example: Conversion = APR → APY Value % = 5 Compounds / year = 12 APY ≈ 5.11619

CAGR — CAGR

Purpose: Computes compound annual growth rate.

Inputs: Start, End, Years

Parameters: Start is initial value. End is ending value. Years is elapsed time.

Result: CAGR = (End / Start)^(1 / Years) − 1 The displayed result is a percentage.

Result behavior: Scalar percent result.

Notes / restrictions: Start > 0 End > 0 Years > 0 Possible errors: Invalid input Start/End/Years must be > 0

Example: Start = 100 End = 160 Years = 5 CAGR ≈ 9.856

Δ% — % Change

Purpose: Computes percentage change from an old value to a new value.

Inputs: Old, New

Result: Δ% = (New − Old) / Old × 100

Result behavior: Scalar percent result.

Notes / restrictions: Old value cannot be zero. Possible errors: Invalid input Old cannot be 0

Example: Old = 80 New = 100 Δ% = 25

R72 — Rule 72

Purpose: Estimates doubling time from rate, or required rate from doubling time.

Inputs: Calculation, Value Prompt default: Calculation = Years from rate % Choice options: Years from rate % Rate % from years

Result: Years ≈ 72 / rate% Rate% ≈ 72 / years

Result behavior: Scalar numeric result.

Notes / restrictions: Value must be positive. Possible errors: Invalid input Value must be > 0

Example: Calculation = Years from rate % Value = 8 Years ≈ 9

SI — Simple Interest

Purpose: Computes simple interest and total amount.

Inputs: Principal, Rate % / year, Years

Parameters: Principal is starting amount. Rate % / year is annual simple rate. Years is time.

Result: I=P×r×t A=P+I Primary result: Total amount A.

Result behavior: Scalar money result. The result line also shows interest and total.

Notes / restrictions: Years must be ≥ 0 Possible errors: Invalid input Years must be ≥ 0

Example: Principal = 1000 Rate % / year = 5 Years = 3 I = 150 A = 1150

CI — Compound Interest

Purpose: Computes compound amount.

Inputs: Principal (P), Rate % / year, Compounds / year, Years Prompt default: Compounds / year = 12

Result: A = P(1 + r/m)^(mt)

Result behavior: Scalar money result.

Notes / restrictions: Principal must be ≥ 0 Compounds/year must be > 0 Years must be ≥ 0 Possible errors: Invalid input Principal must be ≥ 0 Compounds/year must be > 0 Years must be ≥ 0 Invalid rate/compounds Invalid result

Example: Principal = 1000 Rate % / year = 5 Compounds / year = 12 Years = 3 A ≈ 1161.47

INFL — Inflation Adj.

Purpose: Converts nominal amount to real amount, or real amount to nominal amount, using an inflation rate.

Inputs: Conversion, Amount, Inflation % / year, Years Prompt default: Conversion = Nominal → Real Choice options: Nominal → Real Real → Nominal

Result: Nominal → Real: Real = Amount / (1 + i)^years Real → Nominal: Nominal = Amount × (1 + i)^years

Result behavior: Scalar money result.

Notes / restrictions: Years must be ≥ 0 1 + inflation rate must be positive Possible errors: Invalid input Years must be ≥ 0 Invalid inflation, Invalid factor, Invalid result

Example: Conversion = Nominal → Real Amount = 1000 Inflation % / year = 3 Years = 5 Real ≈ 862.61

NPV — Net PV

Purpose: Computes net present value from a discount rate and a cashflow series.

Inputs: Rate % / period, Cashflows CSV Cashflow format: Dot-decimal locale: CF0, CF1, CF2 … Comma-decimal locale: CF0; CF1; CF2 …

Result: NPV = Σ CFt / (1 + r)^t where CF0 is at period 0.

Result behavior: Scalar money result.

Notes / restrictions: The rate must not make 1 + r = 0. Possible errors: Invalid input, Invalid rate

Example: Rate % / period = 10 Cashflows CSV = -1000, 400, 400, 400 NPV ≈ -5.26

IRR — IRR

Purpose: Computes internal rate of return for a cashflow series.

Inputs: Cashflows CSV

Result: Internal rate of return as a percent. The function solves: NPV(rate) = 0

Result behavior: Scalar percent result.

Notes / restrictions: At least two cashflows are required. The cashflows must include at least one positive and one negative value. The solver must find a valid bracket. Possible errors: Invalid cashflows, Need ≥ 2 cashflows IRR needs + and - cashflows No bracket for IRR

Example: Cashflows CSV = -1000, 400, 400, 400 IRR ≈ 9.7

MIRR — Mod. IRR

Purpose: Computes modified internal rate of return using separate finance and reinvestment rates.

Inputs: Finance rate % (rf), Reinvest rate % (rr), Cashflows CSV

Parameters: rf discounts negative cashflows. rr grows positive cashflows.

Result: Modified internal rate of return as a percent.

Result behavior: Scalar percent result.

Notes / restrictions: At least two cashflows are required. The series must contain positive and negative cashflows. Possible errors: Invalid input, Need ≥ 2 cashflows MIRR needs + and - cashflows No real MIRR

Example: Finance rate % = 5 Reinvest rate % = 7 Cashflows CSV = -1000, 400, 400, 400 MIRR ≈ 8.5

PI — Prof. Index

Purpose: Computes profitability index.

Inputs: Rate % / period, CF0, Future cashflows CSV

Parameters: CF0 is the initial investment. Future cashflows begin at period 1.

Result: PI = PV(future cashflows) / |CF0|

Result behavior: Scalar numeric ratio.

Notes / restrictions: CF0 cannot be zero. Possible errors: Invalid input CF0 cannot be 0

Example: Rate % / period = 10 CF0 = -1000 Future cashflows CSV = 400, 400, 400 PI ≈ 0.995

AMORT — Amortization

Purpose: Computes loan payment, balance after a selected payment, and approximate total interest.

Inputs: Principal (PV), Rate % / period, NPER, Balance after payment # (k) Prompt default: Balance after payment # (k) = 0

Parameters: PV is loan principal. Rate % / period is periodic rate. NPER is total number of periods. k is the payment number after which balance is shown. Primary result: Periodic payment amount. Displayed detail: PMT, Bal(k), Int≈

Result behavior: Scalar money result. The primary calculator result is the payment.

Notes / restrictions: NPER must be > 0 k must be 0…NPER Possible errors: Invalid input NPER must be > 0 k must be 0…NPER Invalid rate/periods

Example: Principal = 10000 Rate % / period = 1 NPER = 36 k = 12 Result shows: PMT, Balance after payment 12, Approximate interest

BAL — Loan Balance

Purpose: Computes remaining loan balance after payment number k.

Inputs: Principal (PV), Rate % / period, NPER, k (0..NPER)

Result: Remaining balance after payment k.

Result behavior: Scalar money result.

Notes / restrictions: NPER must be > 0 k must be 0…NPER Possible errors: Invalid input NPER must be > 0 k must be 0…NPER Invalid rate/periods

Example: Principal = 10000 Rate % / period = 1 NPER = 36 k = 12 BAL = remaining balance after 12 payments

MARG — Margins

Purpose: Computes gross margin percentage from cost and price.

Inputs: Cost, Price

Result: Gross margin % = (Price − Cost) / Price × 100

Result behavior: Scalar percent result.

Notes / restrictions: Price cannot be zero. Possible errors: Invalid input Price cannot be 0

Example: Cost = 60 Price = 100 Gross margin = 40%

BEP — Breakeven

Purpose: Computes breakeven units and breakeven revenue.

Inputs: Fixed Costs, Price / unit, Variable cost / unit

Result: Contribution margin = Price − Variable cost Units = Fixed Costs / Contribution margin Revenue = Units × Price Primary result: Breakeven units.

Result behavior: Scalar numeric result. The result line also shows breakeven revenue.

Notes / restrictions: Price must exceed variable cost. Possible errors: Invalid input Price must exceed variable cost

Example: Fixed Costs = 10000 Price / unit = 25 Variable cost / unit = 10 Units ≈ 666.67 Revenue ≈ 16666.67

ROI — ROI

Purpose: Computes return on investment percentage.

Inputs: Initial (Vi), Final (Vf)

Result: ROI% = (Vf − Vi) / Vi × 100

Result behavior: Scalar percent result.

Notes / restrictions: Initial value cannot be zero. Possible errors: Invalid input Initial cannot be 0

Example: Initial = 1000 Final = 1350 ROI = 35%

DISC — Discount

Purpose: Computes final price after a percentage discount.

Inputs: Price, Discount %

Result: Discount amount = Price × discount% Final price = Price − discount amount Primary result: Final discounted price.

Result behavior: Scalar money result. The result line also shows the discount amount. Possible error: Invalid input

Example: Price = 100 Discount % = 15 Discount amount = 15 Final price = 85

TAX — Tax

Purpose: Computes total amount after adding tax.

Inputs: Amount, Tax %

Result: Tax = Amount × tax% Total = Amount + Tax Primary result: Total with tax.

Result behavior: Scalar money result. The result line also shows the tax amount. Possible error: Invalid input

Example: Amount = 100 Tax % = 22 Tax = 22 Total = 122

TIP — Tip

Purpose: Computes tip, total, and per-person amount.

Inputs: Bill, Tip %, Split (optional) Prompt default: Split = 1

Result: Tip = Bill × tip% Total = Bill + Tip Each = Total / Split Primary result: Total with tip.

Result behavior: Scalar money result. The result line also shows tip amount, total, and each-person amount.

Notes / restrictions: Split must be at least 1. Possible errors: Invalid input Split must be ≥ 1

Example: Bill = 80 Tip % = 10 Split = 2 Tip = 8 Total = 88 Each = 44

Errors and limitations

General invalid input Most functions return: Invalid input when a required field is missing or cannot be parsed. Period restrictions Time-value functions restrict period counts: PMT: NPER must be > 0 PV: NPER must be ≥ 0 FV: NPER must be ≥ 0 RATE: NPER must be > 0 AMORT: NPER must be > 0 BAL: NPER must be > 0 Zero-rate handling PMT, PV, FV, NPER, AMORT, and BAL include special handling for zero rate. This avoids division by zero in ordinary annuity formulas. Numerical solving limits RATE and IRR use numerical root-finding. If no valid bracket is found, the function reports: No bracket for RATE No bracket for IRR This usually means the entered signs and magnitudes do not produce a solvable rate in the searched range. Cashflow requirements IRR and MIRR require at least two cashflows and must include both positive and negative cashflows. Possible errors: Need ≥ 2 cashflows IRR needs + and - cashflows MIRR needs + and - cashflows Profitability index PI requires a nonzero initial cashflow: CF0 cannot be 0 Breakeven Breakeven requires price to exceed variable cost: Price must exceed variable cost Split validation TIP requires: Split ≥ 1 Currency limits The currency code is only a label. The Financials docklet does not convert currencies, fetch exchange rates, or adjust for changing currency values.

Practical usage notes

Use PMT, PV, FV, RATE, and NPER together for loan, savings, and annuity calculations. Keep signs consistent: outflows are commonly entered as negative values and inflows as positive values. Use APY when comparing rates with different compounding frequencies. Use CAGR for multi-year growth between a starting and ending value. Use NPV, IRR, MIRR, and PI for project or investment appraisal. For cashflow tools, enter the first cashflow as period 0. Use AMORT when you need payment plus a compact loan summary. Use BAL when you only need remaining balance after a specific payment number. Use DISC, TAX, and TIP for quick consumer calculations where the final amount is the primary result.

Appendix G.8 — Geometry

Overview

The Geometry docklet provides 2D geometry, coordinate geometry, 3D geometry, angle conversion, triangle-law, and basic topology tools. It is designed for users who need direct formula-based calculations from known geometric parameters. Most functions return one scalar value, such as a length, area, volume, angle, ratio, or count. One function, Midpoint 2D, returns a point-style result. The Geometry docklet exposes 48 functions: A◯, C◯, ARC, SECT, SEGM, CHORD, SAG, A▭, P▭, A□, A▱, A, A△, HERON, P△, PYTH, INR A⬠, P⬠, POLY, APOT, PICKS, DIST, DIST3, MID, SLOPE, ∠3PT, PTLINE, V●, S●, CAP, V□ S□, VBOX, SBOX, VCYL, SCYL, VCONE, SCONE, VPYR, SPYR, FRUST, TORUS, D↔R, LOC, LOS χ, GENUS

Opening and using the docklet

The Geometry docklet appears as a parked or expandable panel. When parked, the header shows: Geometry When expanded, the user can scroll through the geometry functions. Selecting a function opens a prompt dialog. Most functions ask for one or more numeric fields. Examples include: r h a b c x1 y1 x2 y2 θ The prompt is firm. Tapping outside the prompt does not dismiss it; it gives warning feedback. Use Cancel to leave the prompt without computing. Most functions compute immediately after all required fields are entered and Compute is tapped.

Current operand behavior

Some Geometry functions can use the current calculator value as an input. A field that says: blank = operand means that leaving that field empty uses the current calculator operand. This applies to these functions: A◯, C◯, V●, S●, A□, V□, S□, D↔R For these tools, the single radius, side length, or conversion value can be taken directly from the calculator display.

Example: Current calculator value: 10, Function: A◯, Field r / s: left blank

Result: Circle area using r = 10 Other functions require manual entry in their prompt fields.

Prompt dialogs

Each function opens a prompt dialog with fields appropriate to the selected formula. Required fields All visible fields must be filled before Compute can run, except fields that explicitly say: blank = operand Numeric input The prompt accepts numeric values such as: 3 -4 2.5 2,5 Dot and comma decimal input are both accepted when parsing numbers. Integer fields Some fields are integer-only: n, Mode, I, V, E, F These are used for polygon side counts, mode selection, lattice-point counts, and topology counts. Mode fields Some functions use a mode field. PYTH uses: Mode 0 = hypotenuse from legs Mode 1 = missing leg from hypotenuse and one leg D↔R uses: Mode 0 = degrees to radians Mode 1 = radians to degrees Mode values are typed as integers. Compute behavior Compute is enabled only when required fields are filled. If invalid values are entered, the prompt remains open and shows an error message. Cancel behavior Cancel closes the prompt without computing and does not change the calculator result.

Decimal places and rounding

Geometry results are rounded according to the app’s Decimal Places setting. The docklet clamps Decimal Places to the range: 0…18 The docklet also snaps values very close to zero or very close to an integer. This avoids display artifacts such as: -0 0.00000000001 1.99999999999 These may be displayed as: 02 Most Geometry results are tagged as numeric results and can be used as the next calculator operand.

Locale and separators

The Geometry docklet accepts both dot and comma decimal input. Comma decimal input is normalized before calculation.

Examples: 2.5, 2,5 Both are treated as the same decimal value. The displayed result follows the selected locale where applicable. Coordinate-style ticker text uses semicolons to separate coordinate components: (1; 2) (1; 2; 3) This is display formatting, not list input.

Angle-unit behavior

The Geometry docklet uses the app’s active angle unit for functions that involve angles. Supported angle units: degrees radians gradians turns User-facing angle labels: degrees → ° radians → rad gradians → grad turns → rev Angle-input functions convert the entered angle internally to radians before calculation. Affected functions include: ARC, SECT, SEGM, CHORD, ∠3PT, LOC, LOS ∠3PT returns the output angle in the active angle unit. D↔R is different: it is an explicit degrees/radians conversion tool and does not depend on the active angle-unit setting.

Result behavior

Most Geometry functions return scalar numeric results. Scalar numeric results become the current calculator value and can continue in later calculations. Examples: circle area, rectangle perimeter, triangle area, distance, slope sphere volume, torus volume Euler characteristic genus One function returns a point-style result: MID For Midpoint 2D, the x-coordinate is placed into the numeric result, while the displayed/ticker result shows the full point: (xmid; ymid) This should be interpreted as a point result, not as a single scalar measurement.

Result tags

The Geometry docklet uses the standard CCalc result tags. [NUM] [NUM] means the result is a numeric value. Most Geometry functions produce [NUM] results. Examples: A◯[L²](r:10) → 314.15926536 [NUM] DIST[L]((0;0)→(3;4)) → 5 [NUM] [APPROX] [APPROX] means the result uses an approximation. In this docklet, ellipse perimeter uses an approximation.

Example: P▯[L](a:5, b:3) → 25.526999 [APPROX] [LABEL] [LABEL] means the result is descriptive, non-scalar, point-style, or an error/warning message.

Example: MID[point:L]((0;0)→(4;6)) → (2; 3) [LABEL] Error messages also use label-style output.

Function groups

The Geometry docklet is organized into these logical groups.

Group 1 — 2D: Circles A◯, C◯, ARC, SECT, SEGM, CHORD, SAG

Group 2 — 2D: Quadrilaterals A▭, P▭, A□, A▱, A

Group 3 — 2D: Triangles A△, HERON, P△, PYTH, INR

Group 4 — 2D: Other Shapes A▯, P▯, POLY, APOT, PICKS

Group 5 — Coordinate: Geometry, DIST, DIST3, MID, SLOPE, ∠3PT, PTLINE

Group 6 — 3D: Spheres V●, S●, CAP

Group 7 — 3D: Prisms and Boxes, V□, S□, VBOX, SBOX, VCYL, SCYL

Group 8 — 3D: Pyramids and Cones, VCONE, SCONE, VPYR, SPYR, FRUST, TORUS

Group 9 — Conversions: D↔R

Group 10 — Triangle Laws: LOC, LOS

Group 11 — Topology and: Shape Analysis, χ, GENUS

Function reference

A◯ — Circle Area

Purpose: Computes the area of a circle from its radius.

Inputs: r/s

Parameters: r is the circle radius.

Result: $A = \pi r^2$

Result behavior: Scalar numeric result.

Notes / restrictions: The radius must be greater than or equal to zero. If the field is left blank, the current calculator operand is used. Possible error: r must be ≥ 0

Example: r = 10 A = 314.15926536

C◯ — Circumference

Purpose: Computes the circumference of a circle from its radius.

Inputs: r/s

Parameters: r is the circle radius.

Result: $C = 2\pi r$

Result behavior: Scalar numeric result.

Notes / restrictions: The radius must be greater than or equal to zero. If the field is left blank, the current calculator operand is used. Possible error: r must be ≥ 0

Example: r = 10 C = 62.83185307

ARC — Arc Length

Purpose: Computes circular arc length.

Inputs: rθ

Parameters: r is the radius. θ is the central angle in the active angle unit.

Result: $L = r\theta$ where θ is converted to radians internally.

Result behavior: Scalar numeric result.

Notes / restrictions: r ≥ 0. The angle must be valid. Possible errors: r must be ≥ 0 Invalid angle, Invalid result

Example: r=5 θ = 90° L = 7.85398163

SECT — Sector Area

Purpose: Computes the area of a circular sector.

Inputs: rθ

Parameters: r is the radius. θ is the central angle in the active angle unit.

Result: $A = 1/2\ r^2\theta$ where θ is converted to radians internally.

Result behavior: Scalar numeric result.

Notes / restrictions: r ≥ 0. Possible errors: r must be ≥ 0 Invalid angle, Invalid result

Example: r=5 θ = 90° Sector area = 19.63495408

SEGM — Segment Area

Purpose: Computes circular segment area from radius and central angle.
Inputs: rθ
Parameters: r is the radius. θ is the central angle in the active angle unit.
Result: $A = (r^2 / 2)(\theta - \sin\theta)$ where θ is converted to radians internally.
Result behavior: Scalar numeric result.
Notes / restrictions: r ≥ 0. Possible errors: r must be ≥ 0 Invalid angle, Invalid result
Example: r=5 θ = 90° Segment area ≈ 7.13495408

CHORD — Chord Length

Purpose: Computes chord length from radius and central angle.
Inputs: rθ
Parameters: r is the radius. θ is the central angle in the active angle unit.
Result: $c = 2r \sin(\theta / 2)$ where θ is converted to radians internally.
Result behavior: Scalar numeric result.
Notes / restrictions: r ≥ 0. Possible errors: r must be ≥ 0 Invalid angle, Invalid result
Example: r=5 θ = 90° c ≈ 7.07106781

SAG — Sagitta (Arc)

Purpose: Computes sagitta from radius and chord length.
Inputs: rc
Parameters: r is the radius. c is the chord length.
Result: $s = r - \text{sqrt}(r^2 - (c/2)^2)$
Result behavior: Scalar numeric result.
Notes / restrictions: r≥0 c≥0 c ≤ 2r Possible errors: r,c must be ≥ 0 c must be ≤ 2r No real solution
Example: r=5 c=6 s=1

A▭ — Rectangle Area

Purpose: Computes rectangle area.
Inputs: wh
Parameters: w is width. h is height.
Result: $A = wh$
Result behavior: Scalar numeric result.
Notes / restrictions: Both dimensions must be greater than or equal to zero. Possible error: w,h must be ≥ 0
Example: w=4 h=5 A = 20

P▭ — Rectangle Perim.

Purpose: Computes rectangle perimeter.
Inputs: wh
Result: $P = 2(w + h)$
Result behavior: Scalar numeric result.
Notes / restrictions: w,h ≥ 0. Possible error: w,h must be ≥ 0
Example: w=4 h=5 P = 18

A□ — Square Area

Purpose: Computes square area from side length.
Inputs: r/s
Parameters: s is side length.
Result: $A = s^2$
Result behavior: Scalar numeric result.
Notes / restrictions: The side length must be greater than or equal to zero. If the field is left blank, the current calculator operand is used. Possible error: side must be ≥ 0
Example: s=6 A = 36

A▱ — Parallelogram

Purpose: Computes parallelogram area.
Inputs: bh
Parameters: b is base. h is height.
Result: $A = bh$
Result behavior: Scalar numeric result.
Notes / restrictions: b,h ≥ 0. Possible error: b,h must be ≥ 0

Example: b=7 h=3 A = 21

A — Trapezoid Area

Purpose: Computes trapezoid area from two bases and height.

Inputs: abh

Parameters: a and b are the parallel bases. h is height.

Result: A = ((a + b) / 2)h

Result behavior: Scalar numeric result.

Notes / restrictions: a,b,h ≥ 0. Possible error: a,b,h must be ≥ 0

Example: a=6 b = 10 h=4 A = 32

A△ — Triangle Area (b·h/2)

Purpose: Computes triangle area from base and height.

Inputs: bh

Result: A = bh / 2

Result behavior: Scalar numeric result.

Notes / restrictions: b,h ≥ 0. Possible error: b,h must be ≥ 0

Example: b=8 h=5 A = 20

HERON — Triangle Area

Purpose: Computes triangle area from three side lengths using Heron’s formula.

Inputs: abc

Parameters: a, b, and c are triangle side lengths.

Result: s = (a + b + c) / 2 A = sqrt(s(s − a)(s − b)(s − c))

Result behavior: Scalar numeric result.

Notes / restrictions: The sides must be positive and must satisfy the triangle inequality. Possible errors: Sides must be > 0 Invalid triangle, No real area

Example: a=3 b=4 c=5 A=6

P△ — Triangle Perim.

Purpose: Computes triangle perimeter from three sides.

Inputs: abc

Result: P=a+b+c

Result behavior: Scalar numeric result.

Notes / restrictions: The sides must be greater than or equal to zero. This function does not enforce triangle inequality; it only sums the entered side lengths. Possible error: Sides must be ≥ 0

Example: a=3 b=4 c=5 P = 12

PYTH — Right Triangle

Purpose: Computes a right-triangle hypotenuse or missing leg.

Inputs: Mode (0 hyp from legs, 1 leg from hyp) a b or c Mode behavior: Mode 0: a = leg b or c = other leg result = hypotenuse c Mode 1: a = known leg b or c = hypotenuse c result = missing leg Results: Mode 0: c = sqrt(a² + b²) Mode 1: b = sqrt(c² − a²)

Result behavior: Scalar numeric result.

Notes / restrictions: Inputs must be nonnegative. Mode 1 requires c² − a² ≥ 0. Possible errors: Invalid input Legs must be ≥ 0 Inputs must be ≥ 0 No real solution

Example: Mode = 0 a=3 b=4 c=5

INR — Incircle Radius

Purpose: Computes triangle incircle radius from three side lengths.

Inputs: abc

Result: r = 2A / (a + b + c) where A is computed using Heron’s formula.

Result behavior: Scalar numeric result.

Notes / restrictions: The sides must be positive and must form a valid triangle. Possible errors: Sides must be > 0 Invalid triangle, No real area, Invalid perimeter

Example: a=3 b=4 c=5 r=1

A□ — Ellipse Area

Purpose: Computes ellipse area from semi-major and semi-minor axes.

Inputs: ab

Parameters: a is semi-major axis. b is semi-minor axis.

Result: A = πab

Result behavior: Scalar numeric result.

Notes / restrictions: a,b ≥ 0. Possible error: a,b must be ≥ 0
Example: a=5 b=3 A ≈ 47.1238898

P□ — Ellipse Perim.

Purpose: Approximates ellipse perimeter.
Inputs: ab
Result: Approximate ellipse perimeter using Ramanujan’s second approximation.
Formula: h = ((a − b)²) / ((a + b)²) P ≈ π(a + b) [1 + 3h / (10 + sqrt(4 − 3h))]
Result behavior: Scalar approximate result.
Notes / restrictions: a,b ≥ 0. If both are zero, the result is zero. Possible errors: a,b must be ≥ 0 Invalid input, Invalid a, b, Invalid result
Example: a=5 b=3 P ≈ 25.526999

POLY — Polygon (n-sides)

Purpose: Computes regular polygon area from number of sides and side length.
Inputs: ns
Parameters: n is number of sides. s is side length.
Result: A = n s² / (4 tan(π/n))
Result behavior: Scalar numeric result.
Notes / restrictions: n≥3 s>0 Possible errors: n must be ≥ 3 side must be > 0 Invalid n, Invalid result
Example: n=4 s=5 A = 25

APOT — Polygon Apothem

Purpose: Computes the apothem of a regular polygon.
Inputs: ns
Parameters: n is number of sides. s is side length.
Result: a = s / (2 tan(π/n))
Result behavior: Scalar numeric result.
Notes / restrictions: n≥3 s>0 Possible errors: n must be ≥ 3 side must be > 0 Invalid n, Invalid result
Example: n=4 s = 10 apothem = 5

PICKS — Pick’s Theorem

Purpose: Computes lattice polygon area using Pick’s theorem.
Inputs: I, B
Parameters: I is number of interior lattice points. B is number of boundary lattice points.
Result: A = I + B/2 − 1
Result behavior: Scalar numeric result.
Notes / restrictions: I≥0 B>0 Possible errors: I must be ≥ 0 B must be ≥ 0 B must be > 0
Example: I=1 B=4 A=2

DIST — Distance 2D

Purpose: Computes distance between two 2D points.
Inputs: x1 y1 x2 y2
Result: d = sqrt((x2 − x1)² + (y2 − y1)²)
Result behavior: Scalar numeric result.
Example: (0;0) to (3;4) d=5

DIST3 — Distance 3D

Purpose: Computes distance between two 3D points.
Inputs: x1 y1 z1 x2 y2 z2
Result: d = sqrt((x2 − x1)² + (y2 − y1)² + (z2 − z1)²)
Result behavior: Scalar numeric result.
Example: (0;0;0) to (1;2;2) d=3

MID — Midpoint 2D

Purpose: Computes the midpoint between two 2D points.
Inputs: x1 y1 x2 y2
Result: ((x1 + x2)/2; (y1 + y2)/2)
Result behavior: Point-style result. The x-coordinate becomes the numeric result, while the full point is shown in the result text.
Notes / restrictions: Interpret the displayed pair as the meaningful result.
Example: (0;0) to (4;6) midpoint = (2; 3)

SLOPE — Slope

Purpose: Computes slope between two 2D points.

Inputs: x1 y1 x2 y2

Result: $m = (y2 - y1) / (x2 - x1)$

Result behavior: Scalar numeric result.

Notes / restrictions: A vertical line has undefined slope and produces division by zero. Possible error: Division by 0

Example: (0;0) to (2;6) m=3

∠3PT — Angle (3 pts)

Purpose: Computes the angle formed by three points, with the angle at point B.

Inputs: Ax, Ay, Bx, By, Cx, Cy

Parameters: The angle is ∠ABC.

Result: Angle in the active angle unit.

Formula: $\cos \theta = ((A - B) \cdot (C - B)) / (|A - B| \, |C - B|)$

Result behavior: Scalar numeric result.

Notes / restrictions: Points A and C must not coincide with B in a way that creates a zero vector. Possible errors: Invalid points, Degenerate points, Invalid result

Example: A = (1;0) B = (0;0) C = (0;1) θ = 90° if the active angle unit is degrees.

PTLINE — Pt-to-Line Dist

Purpose: Computes distance from a point to a line defined by two points.

Inputs: x0 y0 x1 y1 x2 y2

Parameters: P = (x0, y0) is the point. The line passes through (x1, y1) and (x2, y2).

Result: distance from P to line AB

Result behavior: Scalar numeric result.

Notes / restrictions: The two line points must not be identical. Possible errors: Invalid points, Degenerate line, Invalid result

Example: P = (0;2) Line through (0;0) and (4;0) distance = 2

V● — Sphere Volume

Purpose: Computes sphere volume from radius.

Inputs: r/s

Parameters: r is sphere radius.

Result: $V = 4/3 \, \pi r^3$

Result behavior: Scalar numeric result.

Notes / restrictions: r ≥ 0. If the field is left blank, the current calculator operand is used. Possible error: r must be ≥ 0

Example: r=3 V ≈ 113.09733553

S● — Sphere Surface

Purpose: Computes sphere surface area from radius.

Inputs: r/s

Result: $S = 4\pi r^2$

Result behavior: Scalar numeric result.

Notes / restrictions: r ≥ 0. If the field is left blank, the current calculator operand is used. Possible error: r must be ≥ 0

Example: r=3 S ≈ 113.09733553

CAP — Spherical Cap

Purpose: Computes spherical cap volume.

Inputs: R, h

Parameters: R is sphere radius. h is cap height.

Result: $V = \pi h^2(3R - h) / 3$

Result behavior: Scalar numeric result.

Notes / restrictions: R≥0 h≥0 h ≤ 2R Possible errors: R,h must be ≥ 0 h must be ≤ 2R Invalid result

Example: R=5 h=2 V ≈ 46.07669225

V□ — Cube Volume

Purpose: Computes cube volume from side length.

Inputs: r/s

Parameters: s is side length.

Result: $V = s^3$

Result behavior: Scalar numeric result.

Notes / restrictions: s ≥ 0. If the field is left blank, the current calculator operand is used. Possible error: side must be ≥ 0

Example: s=4 V = 64

S□ — Cube Surface

Purpose: Computes cube surface area from side length.
Inputs: r/s
Result: S = 6s²
Result behavior: Scalar numeric result.
Notes / restrictions: s ≥ 0. If the field is left blank, the current calculator operand is used. Possible error: side must be ≥ 0
Example: s=4 S = 96

VBOX — Box Volume

Purpose: Computes rectangular box volume.
Inputs: lwh
Result: V = lwh
Result behavior: Scalar numeric result.
Notes / restrictions: l,w,h ≥ 0. Possible error: l,w,h must be ≥ 0
Example: l=2 w=3 h=4 V = 24

SBOX — Box Surface

Purpose: Computes rectangular box surface area.
Inputs: lwh
Result: S = 2(lw + lh + wh)
Result behavior: Scalar numeric result.
Notes / restrictions: l,w,h ≥ 0. Possible error: l,w,h must be ≥ 0
Example: l=2 w=3 h=4 S = 52

VCYL — Cylinder Volume

Purpose: Computes cylinder volume.
Inputs: rh
Result: V = πr²h
Result behavior: Scalar numeric result.
Notes / restrictions: r,h ≥ 0. Possible error: r,h must be ≥ 0
Example: r=2 h=5 V ≈ 62.83185307

SCYL — Cylinder Surface

Purpose: Computes total cylinder surface area.
Inputs: rh
Result: S = 2πr(h + r)
Result behavior: Scalar numeric result.
Notes / restrictions: r,h ≥ 0. Possible error: r,h must be ≥ 0
Example: r=2 h=5 S ≈ 87.9645943

VCONE — Cone Volume

Purpose: Computes cone volume.
Inputs: rh
Result: V = πr²h / 3
Result behavior: Scalar numeric result.
Notes / restrictions: r,h ≥ 0. Possible error: r,h must be ≥ 0
Example: r=3 h=6 V ≈ 56.54866776

SCONE — Cone Surface

Purpose: Computes total cone surface area.
Inputs: rh
Result: l = sqrt(r² + h²) S = πr(r + l)
Result behavior: Scalar numeric result.
Notes / restrictions: r,h ≥ 0. Possible errors: r,h must be ≥ 0 No real slant height
Example: r=3 h=4 l=5 S ≈ 75.39822369

VPYR — Pyramid Volume

Purpose: Computes pyramid volume from base area and height.
Inputs: B, h
Parameters: B is base area. h is height.
Result: V = Bh / 3

Result behavior: Scalar numeric result.

Notes / restrictions: B,h ≥ 0. Possible error: B,h must be ≥ 0

Example: B = 30 h=9 V = 90

SPYR — Sq Pyramid Surface

Purpose: Computes total surface area of a square pyramid.

Inputs: sh

Parameters: s is base side length. h is vertical height.

Result: l = sqrt((s/2)² + h²) S = s² + 2sl

Result behavior: Scalar numeric result.

Notes / restrictions: s,h ≥ 0. Possible errors: s,h must be ≥ 0 No real slant height

Example: s=6 h=4 l=5 S = 96

FRUST — Frustum Volume

Purpose: Computes conical frustum volume.

Inputs: R, rh

Parameters: R is outer radius. r is inner radius. h is height.

Result: V = (πh / 3)(R² + Rr + r²)

Result behavior: Scalar numeric result.

Notes / restrictions: R,r,h ≥ 0. Possible error: R,r,h must be ≥ 0 Invalid result

Example: R=5 r=3 h=6 V ≈ 307.87608005

TORUS — Torus Volume

Purpose: Computes torus volume.

Inputs: R, r

Parameters: R is major radius. r is minor radius.

Result: V = 2π²Rr²

Result behavior: Scalar numeric result.

Notes / restrictions: R,r ≥ 0. Possible error: R,r must be ≥ 0 Invalid result

Example: R = 10 r=2 V ≈ 789.56835209

D↔R — Degrees ↔ Radians

Purpose: Converts between degrees and radians.

Inputs: Mode (0 deg→rad, 1 rad→deg) Value (blank = operand) Mode behavior: 0 = degrees to radians 1 = radians to degrees

Result: rad = deg × π / 180 deg = rad × 180 / π

Result behavior: Scalar numeric result.

Notes / restrictions: This function is explicit and independent of the active angle unit setting. If the value field is left blank, the current calculator operand is used. Possible errors: Invalid input, Invalid value, Invalid result

Example: Mode = 0 Value = 180 Result = 3.14159265

LOC — Law of Cosines

Purpose: Computes the third side of a triangle using the Law of Cosines.

Inputs: ab, C

Parameters: a and b are known sides. C is the included angle in the active angle unit.

Result: c² = a² + b² − 2ab cos(C) c = sqrt(c²)

Result behavior: Scalar numeric result.

Notes / restrictions: a,b ≥ 0. The entered angle follows the active angle unit. Possible errors: a,b must be ≥ 0 Invalid angle, No real solution, Invalid result

Example: a=3 b=4 C = 90° c=5

LOS — Law of Sines

Purpose: Computes a side using the Law of Sines.

Inputs: a, A, B

Parameters: a is a known side. A is the angle opposite side a. B is the angle opposite the unknown side.

Result: b = a sin(B) / sin(A)

Result behavior: Scalar numeric result.

Notes / restrictions: a ≥ 0. Angles follow the active angle unit. sin(A) must not be zero. Possible errors: a must be ≥ 0 Invalid a, Invalid angle Division by 0 Invalid result

Example: a = 10 A = 30° B = 60° b ≈ 17.32050808

χ — Euler Characteristic

Purpose: Computes Euler characteristic from vertices, edges, and faces.

Inputs: V, E, F

Parameters: V is vertices. E is edges. F is faces.

Result: $\chi = V - E + F$

Result behavior: Scalar numeric count.

Notes / restrictions: V, E, and F must be nonnegative integers. Possible errors: Invalid input, V, E, F must be ≥ 0

Example: V=8 E = 12 F=6 $\chi = 2$

GENUS — Genus (Holes)

Purpose: Computes genus for a closed orientable surface using Euler characteristic.

Inputs: V, E, F

Result: $\chi = V - E + F$ $g = (2 - \chi) / 2$

Result behavior: Scalar numeric count.

Notes / restrictions: The result must be a nonnegative integer for a valid closed orientable surface. Possible errors: Invalid input, V, E, F must be ≥ 0, Invalid closed orientable surface Genus must be an integer

Example: $\chi = 0$ g=1

Errors and limitations

Negative dimensions Most length, radius, side, height, area, and volume inputs must be nonnegative. Typical errors: r must be ≥ 0 side must be ≥ 0 w,h must be ≥ 0 l,w,h must be ≥ 0 R,r,h must be ≥ 0 Strictly positive values Some formulas require positive rather than merely nonnegative values. Examples: Triangle sides in HERON and INR must be > 0 Regular polygon side length must be > 0 Regular polygon n must be ≥ 3 Triangle validity Heron's formula and incircle radius require a valid triangle. Possible errors: Invalid triangle, No real area, Square-root restrictions Several functions require a nonnegative radicand. Examples: PYTH mode 1, SCONE, SPYR, SAG Possible errors: No real solution, No real slant height Division by zero Some calculations can fail when a denominator is zero.

Examples: SLOPE with x1 = x2 LOS with sin(A) = 0 Possible error: Division by 0 Degenerate geometry Some coordinate-geometry calculations require distinct or nondegenerate points. Possible errors: Degenerate points, Degenerate line, Invalid points, Angle behavior Angle-input formulas use the active CCalc angle unit. Enter angles according to the current setting. Examples: If angle unit is degrees, enter 90 for a right angle. If angle unit is radians, enter 1.57079632679 for a right angle. If angle unit is gradians, enter 100 for a right angle. If angle unit is turns, enter 0.25 for a right angle. D↔R does not depend on the active angle unit. Ellipse perimeter approximation Ellipse perimeter is approximate. The docklet uses Ramanujan's approximation, not an exact elliptic-integral calculation. Topology assumptions GENUS assumes a closed orientable surface. If the Euler characteristic does not correspond to a valid integer genus, the function returns an error.

Practical usage notes

Use circle tools when the radius and angle are known. Use ARC, SECT, SEGM, and CHORD together when comparing related circular measurements. Use SAG when the radius and chord are known and the arc height is needed. Use A△ when base and height are known. Use HERON when only the three sides are known. Use INR when the incircle radius is needed from side lengths. Use PYTH for right-triangle side recovery. Mode 0 finds the hypotenuse. Mode 1 finds a missing leg. Use LOC and LOS for non-right triangles. Make sure the active angle unit matches the angle values being entered. Use DIST, DIST3, MID, SLOPE, ∠3PT, and PTLINE for coordinate geometry. Use V●, S●, V□, S□, VBOX, SBOX, VCYL, SCYL, VCONE, SCONE, VPYR, SPYR, FRUST, TORUS, and CAP for 3D measurement problems. Use χ and GENUS only when the vertices, edges, and faces describe the intended topological structure.

Appendix G.9 — Linear Algebra

Overview

The Linear Algebra docklet provides vector and matrix tools for 3D vectors, 2×2 matrices, and 3×3 matrices. It covers: Vector operations, Matrix arithmetic Matrix transformations Matrix properties, Row reduction LU and QR decomposition Linear systems Characteristic polynomials Eigenvalues and eigenvectors Singular value decomposition The docklet is intended for users who need compact technical calculations rather than full symbolic algebra. Most matrix operations are numerical and return rounded decimal results according to the app's Decimal Places setting. The docklet exposes 32 functions: $a \cdot b$, $a \times b$, $\|v\|$, $\hat{v}$, $\angle(u, v)$, $d(u, v)$, $\text{proj}_v u$, $a \cdot (b \times c)$, $u+v$, $u-v$, kv, $A \pm B$, kA, AB Av, A^n, A^T, [A, B], tr(A), det(A), A^{-1}, rank(A), κ(A), RREF, A=LU, A=QR, ORTHO Ax=b, P(λ), λ, v_λ, SVD

Opening and using the docklet

The Linear Algebra docklet appears as a parked or expandable panel. When parked, the header shows: Linear Algebra When expanded, the user can scroll through the available functions. Selecting a function opens a prompt dialog. The dialog may ask directly for vector components, or it may first ask for the matrix size. Matrix operations support: 2×2 matrices 3×3 matrices For many matrix functions, the workflow is two-step:

1. Enter the matrix size, usually 2 or 3.
2. Tap Compute.
3. The dialog expands and shows the matrix-entry fields.
4. Enter the matrix values.

5. Tap Compute again to calculate the result.

The prompt is firm. Tapping outside the prompt does not dismiss it; it gives warning feedback. Use Cancel to leave the prompt without computing.

Current operand behavior

Most Linear Algebra functions require manual input. However, some vector fields support: blank = operand When a field says blank = operand, leaving that field empty uses the current calculator value. This behavior applies to the first component of certain vector-style inputs: ‖v‖ v̂ kv, Av, Ax=b Examples: If the current calculator value is 10, then in a vector field: x: blank = operand y: 0 z: 0 leaving x blank makes the vector: (10; 0; 0) This only applies where the prompt explicitly says blank = operand. All other fields must be entered manually.

Prompt dialogs

Prompt dialogs are required for every function in this docklet. Required fields All visible fields must contain valid input before the Compute button can run. The only exception is a field whose placeholder explicitly says: blank = operand In that case, the empty field is allowed and the current calculator operand is used. Numeric input The docklet accepts decimal numeric input. Examples: 2, -3, 4.5, 0,75 Both dot and comma decimal input are parsed as decimal values. Matrix size field Many matrix functions begin with: Size (2 or 3) Valid values: 23 Invalid values produce: Size must be 2 or 3 Invalid size, A±B mode field The A±B function includes a mode choice: Add, Subtract The user-facing dialog shows this as a segmented choice. Matrix-entry fields For a 2×2 matrix, fields are: A11 A12, A21 A22 For a 3×3 matrix, fields are: A11 A12 A13, A21 A22 A23, A31 A32 A33 For two-matrix operations, the second matrix uses B fields: B11 B12, B21 B22 or: B11 B12 B13, B21 B22 B23, B31 B32 B33, Vector-entry fields 3D vector functions use: xyz or: ux uy uz vx vy vz The docklet treats vector inputs as 3D vectors unless the function specifically uses a 2×2 matrix with a 2D vector. Compute behavior For many matrix functions, the first Compute expands the dialog and does not yet calculate the result. The second Compute performs the calculation after all matrix entries are filled. Cancel behavior Cancel closes the prompt and does not change the current calculator value.

Decimal places and rounding

Numeric results are rounded according to the app's Decimal Places setting. The docklet also snaps very small values to zero and snaps values very close to integers to the nearest integer. This helps avoid visual artifacts such as: 0.0000000001 -0 1.9999999999 Typical cleaned results may display as: 0 2 -3 The docklet normalizes negative zero to: 0 Vector and matrix results are formatted component by component using the same decimal formatting.

Locale and separators

The Linear Algebra docklet accepts decimal input with either dot or comma decimal separators because entered values are normalized before parsing. Examples: 1.5, 1,5 Both are interpreted as: 1.5 Matrix and vector output uses semicolons inside rows and between vector components. Vector output: (1; 2; 3) 2×2 matrix output: [[1; 2]; [3; 4]] 3×3 matrix output: [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]] The semicolon is part of the matrix/vector display format. It is not a list-entry separator that the user types into one field.

Angle-unit behavior

The Linear Algebra docklet does not use the app's angle-unit setting. The ∠(u,v) function always returns the angle in radians. Its result label includes: rad

Example: $\angle_3(u,v)$ rad Degrees, gradians, and turns do not affect this docklet.

Result behavior

The docklet has two broad result types. Scalar numeric results Scalar results return one number and can continue in ordinary calculator expressions. Examples: a·b, ‖v‖, ∠(u, v), d(u, v), a·(b×c), tr(A), det(A), rank(A), κ(A), λ Label-style or structured results Vector, matrix, decomposition, system, eigenvector, and SVD results are structured outputs. They are displayed as text and should be read as final descriptive results. Examples: a×b, v̂, $\text{proj}_v u$, u+v, u−v, kv, A±B, kA, AB, Av, A^n, A^T, A^{-1}, [A, B], RREF, A=LU A=QR, ORTHO, Ax=b, P(λ), v_λ, SVD These results are not ordinary single-number outputs, even when they contain many numbers.

Result tags

The Linear Algebra docklet uses the standard CCalc result tags. [NUM] [NUM] means the result is numeric and exact in form within the docklet's numerical method. In this docklet, rank(A) uses [NUM].

Example: rank(A_2) [NUM] [APPROX] [APPROX] means the result is numerical and should be treated as approximate. Most scalar linear algebra results use [APPROX], including: Dot product, Norm, Angle, Distance, Trace, Determinant, Condition number Eigenvalue

Example: det(A_2) [APPROX] [LABEL] [LABEL] means the result is structured, descriptive, vector-valued, matrix-valued, symbolic, or non-scalar. Examples: $\text{CROSS}_3(u,v)$ = (0; 0; 1) [LABEL] A_2^{-1} = [[...]] [LABEL] RREF(A_3) = [[...]] • rank=2 [LABEL] SVD(A_2) = U=... • Σ=... • V^T=... [LABEL]

Function groups

The docklet is grouped into five logical sections.

Group 1 — Vector Operations: a·b, a×b, ‖v‖, v̂, ∠(u, v), d(u, v), $\text{proj}_v u$, a·(b×c), u+v, u−v, kv

Group 2 — Matrix Arithmetic: and Transformations, A±B, kA, AB, Av, A^n, A^T, [A, B]
Group 3 — Matrix Properties: tr(A), det(A), A^{-1}, rank(A), κ(A)
Group 4 — Decompositions: and Forms, RREF, A=LU, A=QR, ORTHO
Group 5 — Systems and: Spectral Theory, Ax=b, P(λ), λ, v_λ, SVD

Function reference

a·b — Dot Product

Purpose: Computes the dot product of two 3D vectors.
Inputs: ux, uy, uz vx, vy, vz
Parameters: u = (ux, uy, uz) and v = (vx, vy, vz).
Result: u · v = ux vx + uy vy + uz vz
Result behavior: Scalar numeric result. Can continue in calculator expressions.
Notes / restrictions: Both vectors must contain valid numeric components.
Example: u = (1; 2; 3) v = (4; 5; 6) u·v = 32

a×b — Cross Product

Purpose: Computes the cross product of two 3D vectors.
Inputs: ux, uy, uz vx, vy, vz
Parameters: u and v are 3D vectors.
Result: u×v
Result behavior: Structured vector result. Label-style output.
Formula: u×v= (uy vz − uz vy; uz vx − ux vz; ux vy − uy vx)
Notes / restrictions: Only 3D vectors are supported.
Example: u = (1; 0; 0) v = (0; 1; 0) u×v = (0; 0; 1)

‖v‖ — Norm / Magnitude

Purpose: Computes the Euclidean norm of a 3D vector.
Inputs: xyz Default behavior: x: blank = operand y: 0 z: 0
Parameters: v = (x, y, z).
Result: ‖v‖ = sqrt($x^2 + y^2 + z^2$)
Result behavior: Scalar numeric result. Can continue in calculator expressions.
Notes / restrictions: If x is left blank, the current calculator operand is used.
Example: v = (3; 4; 0) ‖v‖ = 5

v̂ — Normalize

Purpose: Returns the unit vector in the direction of v.
Inputs: xyz Default behavior: x: blank = operand y: 0 z: 0
Parameters: v = (x, y, z).
Result: v̂ = v / ‖v‖
Result behavior: Structured vector result. Label-style output.
Notes / restrictions: The zero vector cannot be normalized. Possible error: Cannot normalize
Example: v = (3; 4; 0) v̂ = (0.6; 0.8; 0)

∠(u,v) — Angle Between

Purpose: Computes the angle between two 3D vectors.
Inputs: ux, uy, uz vx, vy, vz
Parameters: u and v are 3D vectors.
Result: Angle in radians.
Formula: θ = arccos((u·v) / (‖u‖ ‖v‖))
Result behavior: Scalar numeric result. Can continue in calculator expressions.
Notes / restrictions: The angle is always returned in radians. The app’s angle-unit setting does not affect this function.
Possible errors: Zero vector, Invalid result
Example: u = (1; 0; 0) v = (0; 1; 0) θ = 1.57079633 rad

d(u,v) — Vector Distance

Purpose: Computes Euclidean distance between two 3D vectors.
Inputs: ux, uy, uz vx, vy, vz
Parameters: u and v are points or vectors in 3D space.
Result: d(u,v) = ‖u − v‖
Result behavior: Scalar numeric result. Can continue in calculator expressions.
Example: u = (1; 2; 3) v = (1; 2; 6) d(u,v) = 3

$\text{proj}_v u$ — Projection
Purpose: Projects vector u onto vector v.
Inputs: ux, uy, uz vx, vy, vz
Parameters: u is the vector being projected. v is the direction vector.
Result: $\text{proj}_v u = ((u \cdot v) / (v \cdot v))\, v$
Result behavior: Structured vector result. Label-style output.
Notes / restrictions: v must not be the zero vector. Possible error: Division by 0
Example: u = (3; 4; 0) v = (1; 0; 0) $\text{proj}_v u$ = (3; 0; 0)

a·(b×c) — Scalar Triple Product
Purpose: Computes the scalar triple product of three 3D vectors.
Inputs: ax, ay, az bx, by, bz cx, cy, cz
Parameters: a, b, and c are 3D vectors.
Result: $a \cdot (b \times c)$
Result behavior: Scalar numeric result. Can continue in calculator expressions.
Notes / restrictions: The result is the signed volume of the parallelepiped formed by the three vectors.
Example: a = (1; 0; 0) b = (0; 1; 0) c = (0; 0; 1) a·(b×c) = 1

u+v — Vector Add
Purpose: Adds two 3D vectors.
Inputs: ux, uy, uz vx, vy, vz
Result: u+v
Result behavior: Structured vector result. Label-style output.
Example: u = (1; 2; 3) v = (4; 5; 6) u+v = (5; 7; 9)

u−v — Vector Subtract
Purpose: Subtracts one 3D vector from another.
Inputs: ux, uy, uz vx, vy, vz
Result: u−v
Result behavior: Structured vector result. Label-style output.
Example: u = (4; 5; 6) v = (1; 2; 3) u−v = (3; 3; 3)

kv — Scalar Multiply (vector)
Purpose: Multiplies a 3D vector by a scalar.
Inputs: kxyz Default behavior: x: blank = operand y: 0 z: 0
Parameters: k is the scalar. v = (x, y, z).
Result: kv = (kx; ky; kz)
Result behavior: Structured vector result. Label-style output.
Example: k=3 v = (1; 2; 3) kv = (3; 6; 9)

A±B — Matrix Add/Sub
Purpose: Adds or subtracts two matrices of the same size.
Inputs: Size (2 or 3) Mode, A entries, B entries Mode: Add, Subtract
Parameters: A and B are both 2×2 or both 3×3.
Result: A+B or: A−B
Result behavior: Structured matrix result. Label-style output.
Notes / restrictions: Only 2×2 and 3×3 matrices are supported.
Example: A = [[1; 2]; [3; 4]] B = [[5; 6]; [7; 8]] A+B = [[6; 8]; [10; 12]]

kA — Scalar Multiply (matrix)
Purpose: Multiplies a matrix by a scalar.
Inputs: Size (2 or 3) k, A entries
Parameters: k is the scalar. A is a 2×2 or 3×3 matrix.
Result: kA
Result behavior: Structured matrix result. Label-style output.
Example: k=2 A = [[1; 2]; [3; 4]] kA = [[2; 4]; [6; 8]]

AB — Matrix Multiply
Purpose: Multiplies two matrices.
Inputs: Size (2 or 3) A entries, B entries
Parameters: A and B are both 2×2 or both 3×3.
Result: AB

Result behavior: Structured matrix result. Label-style output.
Notes / restrictions: Only square 2×2 and 3×3 matrix multiplication is supported.
Example: A = [[1; 2]; [3; 4]] B = [[5; 6]; [7; 8]] AB = [[19; 22]; [43; 50]]

Av — Matrix × Vector

Purpose: Multiplies a matrix by a vector.
Inputs: Size (2 or 3) A entries, v entries For 2×2: vx vy For 3×3: vx vy vz Default behavior: vx: blank = operand
Parameters: A is a 2×2 or 3×3 matrix. v is a compatible vector.
Result: Av
Result behavior: Structured vector result. Label-style output.
Example: A = [[1; 2]; [3; 4]] v = (5; 6) Av = (17; 39)

A^n — Matrix Power

Purpose: Raises a square matrix to a nonnegative integer power.
Inputs: Size (2 or 3) n, A entries
Parameters: n must be an integer greater than or equal to zero.
Result: A^n
Result behavior: Structured matrix result. Label-style output.
Notes / restrictions: n≥0 If n = 0, the result is the identity matrix of the selected size.
Example: A = [[1; 1]; [0; 1]] n=2 A^2 = [[1; 2]; [0; 1]]

A^T — Transpose

Purpose: Computes the transpose of a matrix.
Inputs: Size (2 or 3) A entries
Result: A^T
Result behavior: Structured matrix result. Label-style output.
Example: A = [[1; 2]; [3; 4]] A^T = [[1; 3]; [2; 4]]

[A,B] — Commutator

Purpose: Computes the matrix commutator.
Inputs: Size (2 or 3) A entries, B entries
Result: [A,B] = AB − BA
Result behavior: Structured matrix result. Label-style output.
Notes / restrictions: Only 2×2 and 3×3 square matrices are supported.
Example: [A,B] = AB − BA If the result is the zero matrix, the matrices commute for that operation.

tr(A) — Trace

Purpose: Computes the trace of a square matrix.
Inputs: Size (2 or 3) A entries
Result: tr(A) = sum of diagonal entries For 2×2: tr(A) = A11 + A22 For 3×3: tr(A) = A11 + A22 + A33
Result behavior: Scalar numeric result. Can continue in calculator expressions.
Example: A = [[1; 2]; [3; 4]] tr(A) = 5

det(A) — Determinant

Purpose: Computes the determinant of a 2×2 or 3×3 matrix.
Inputs: Size (2 or 3) A entries
Result: det(A)
Result behavior: Scalar numeric result. Can continue in calculator expressions. For 2×2: det([[a; b]; [c; d]]) = ad − bc
Notes / restrictions: The result is numerical and rounded according to Decimal Places.
Example: A = [[1; 2]; [3; 4]] det(A) = -2

A^{-1} — Inverse

Purpose: Computes the inverse of a nonsingular 2×2 or 3×3 matrix.
Inputs: Size (2 or 3) A entries
Result: A^{-1}
Result behavior: Structured matrix result. Label-style output.
Notes / restrictions: The matrix must be nonsingular. Possible error: Singular matrix
Example: A = [[1; 2]; [3; 4]] A^{-1} = [[-2; 1]; [1.5; -0.5]]

rank(A) — Rank

Purpose: Computes matrix rank using row reduction.
Inputs: Size (2 or 3) A entries
Result: rank(A)

Result behavior: Scalar numeric result. Can continue in calculator expressions.

Notes / restrictions: The rank is computed numerically using a tolerance. Very nearly dependent rows or columns may be affected by numerical precision.

Example: A = [[1; 2]; [2; 4]] rank(A) = 1

κ(A) — Condition Number

Purpose: Computes a numerical condition number based on singular values.

Inputs: Size (2 or 3) A entries

Result: κ(A) = largest singular value / smallest singular value

Result behavior: Scalar numeric result when finite. Label-style result when infinite.

Notes / restrictions: If the smallest singular value is effectively zero, the result is: ∞ Possible errors: Condition number failed, Invalid result

Example: A = identity matrix κ(A) = 1

RREF — Reduced Row Echelon

Purpose: Computes the reduced row echelon form of a matrix and reports its rank.

Inputs: Size (2 or 3) A entries

Result: RREF(A), rank=<rank>

Result behavior: Structured matrix result. Label-style output.

Notes / restrictions: Uses numerical row reduction with tolerance. Near-singular or nearly dependent matrices may be sensitive to rounding.

Example: A = [[1; 2]; [2; 4]] RREF(A) = [[1; 2]; [0; 0]] rank=1 A=LU — LU Decomposition

Purpose: Computes an LU decomposition.

Inputs: Size (2 or 3) A entries

Result: L, U where: A = LU

Result behavior: Structured matrix result. Label-style output.

Notes / restrictions: This LU calculation does not perform pivoting. It can fail when a pivot is zero or too small, even if a pivoted decomposition would exist. Possible error: LU failed (pivot needed or singular)

Example: A = [[2; 1]; [4; 3]] L = [[1; 0]; [2; 1]] U = [[2; 1]; [0; 1]] A=QR — QR Decomposition

Purpose: Computes a QR decomposition using Gram–Schmidt-style orthogonalization.

Inputs: Size (2 or 3) A entries

Result: Q, R where: A = QR

Result behavior: Structured matrix result. Label-style output.

Notes / restrictions: The columns must be suitable for QR decomposition. Dependent or near-dependent columns may fail. Possible error: QR failed

Example: A=QR Q=[[...]] R=[[...]]

ORTHO — Gram–Schmidt (3D)

Purpose: Applies Gram–Schmidt orthonormalization to three 3D vectors.

Inputs: v1x, v1y, v1z v2x, v2y, v2z v3x, v3y, v3z

Parameters: Three 3D vectors.

Result: e1 e2 e3 where e1, e2, and e3 are orthonormal vectors.

Result behavior: Structured vector result. Label-style output.

Notes / restrictions: The input vectors must be independent enough to form an orthonormal basis. Possible errors: v1 is zero v2 dependent v3 dependent

Example: v1 = (1; 0; 0) v2 = (1; 1; 0) v3 = (1; 1; 1)

Result: e1=(1; 0; 0) e2=(0; 1; 0) e3=(0; 0; 1) Ax=b — Solve System

Purpose: Solves a linear system using matrix inverse.

Inputs: Size (2 or 3) A entries, b entries For 2×2: bx by For 3×3: bx by bz Default behavior: bx: blank = operand

Result: $x = A^{-1}b$

Result behavior: Structured vector result. Label-style output.

Notes / restrictions: The matrix must be nonsingular. Possible errors: Invalid A/b, Singular matrix

Example: A = [[2; 0]; [0; 4]] b = (8; 12) x = (4; 3)

P(λ) — Char. Polynomial

Purpose: Computes the characteristic polynomial of a 2×2 or 3×3 matrix.

Inputs: Size (2 or 3) A entries

Result: Symbolic polynomial text. For 2×2: $\lambda^2 - (\mathrm{tr}(A))\lambda + \det(A)$ For 3×3: $\lambda^3 - (\mathrm{tr}(A))\lambda^2 + c_2\lambda - \det(A)$ where: $c_2 = 1/2((\mathrm{tr}(A))^2 - \mathrm{tr}(A^2))$

Result behavior: Label-style result.

Notes / restrictions: The output is a formatted polynomial using rounded numeric coefficients.

Example: A = [[1; 0]; [0; 2]] $P_2(\lambda) = \lambda^2 - (3)\lambda + (2)$

λ — Eigenvalues

Purpose: Computes a real eigenvalue of a 2×2 or 3×3 matrix.

Inputs: Size (2 or 3) A entries

Result: The first real eigenvalue returned by the numerical method.

Result behavior: Scalar numeric result. Can continue in calculator expressions.

Notes / restrictions: For 2×2 matrices, complex eigenvalues are not supported. For 3×3 matrices, real eigenvalues are approximated using QR iteration. Complex eigenvalues are not supported. Possible errors: Complex eigenvalues not supported Complex eigenvalues not yet supported

Example: A = [[2; 0]; [0; 3]] $\lambda_1 = 3$ depending on ordering returned by the calculation.

v_λ — Eigenvectors

Purpose: Computes eigenvector information associated with real eigenvalues.

Inputs: Size (2 or 3) A entries

Result: Eigenvalue and eigenvector information.

Result behavior: Label-style result, with a scalar eigenvalue also shown. For 2×2, the output includes: $\lambda_1(A_2)$=... v_2=(...; ...) λ_2=... available For 3×3, the output lists eigenvalue/eigenvector pairs where stable.

Notes / restrictions: Complex eigenvalues are not supported. Eigenvectors can be unstable for repeated, nearly repeated, or ill-conditioned eigenvalues. Possible errors: Complex eigenvalues not supported Complex eigenvalues not supported (yet) Eigenvector unstable

Example: A = [[2; 0]; [0; 3]] $\lambda_1(A_2)$=3 v_2=(0; 1) λ_2=2 available

SVD — Singular Value Dec.

Purpose: Computes a singular value decomposition.

Inputs: Size (2 or 3) A entries

Result: U, Σ, V^T where: $A = U\Sigma V^T$

Result behavior: Structured matrix result. Label-style output.

Notes / restrictions: The result is numerical. SVD can fail if the numerical eigenvalue or orthogonalization steps fail. Possible error: SVD failed

Example: SVD(A_2), U=[[...]], Σ=[[σ1; 0]; [0; σ2]], V^T=[[...]]

Errors and limitations

Supported dimensions The docklet supports: 3D vectors 2×2 matrices 3×3 matrices It does not support arbitrary matrix sizes. Invalid size values produce: Size must be 2 or 3 Invalid size, Required matrix expansion step For many matrix functions, the first Compute after selecting the size expands the dialog. This is expected behavior. The calculation happens only after the matrix fields have appeared and been filled. Invalid numeric input Invalid fields produce errors such as: Invalid input, Invalid vectors, Invalid A, Invalid A/B, Invalid A/v, Invalid A/b Invalid matrices, Singular matrices Inverse-based operations require a nonsingular matrix. Affected functions include: A^{-1}, Ax=b Possible error: Singular matrix, Zero vectors Vector normalization and angle calculations require nonzero vectors. Possible errors: Cannot normalize, Zero vector Projection division by zero Projection onto the zero vector is invalid. Possible error: Division by 0 LU decomposition limitations LU decomposition does not use pivoting. It may fail when pivoting would be required. Possible error: LU failed (pivot needed or singular) QR decomposition limitations QR decomposition can fail for zero, dependent, or nearly dependent columns. Possible error: QR failed, Eigenvalue limitations Complex eigenvalues are not supported. Possible errors: Complex eigenvalues not supported Complex eigenvalues not yet supported 3×3 eigenvalues use a numerical QR iteration. If the method does not converge to usable real diagonal values, the function fails. Eigenvector limitations Eigenvectors can be unstable for repeated eigenvalues, nearly repeated eigenvalues, or ill-conditioned matrices. Possible error: Eigenvector unstable, SVD limitations SVD is numerical. It depends on singular-value and orthogonalization calculations. It can fail for difficult or unstable cases. Possible error: SVD failed, Condition number limitations Condition number is based on singular values. If the smallest singular value is effectively zero, the result is: ∞ Possible error: Condition number failed Approximation behavior Many Linear Algebra results are approximate numerical results. This includes determinant, eigenvalues, SVD, QR, condition number, and angle calculations. Rounded output should not be interpreted as exact symbolic algebra.

Practical usage notes

Use vector operations when working with 3D quantities such as forces, directions, displacement vectors, and geometric vectors. Use a·b for alignment or projection-related calculations. Use a×b when the perpendicular vector or oriented area is needed. Use ‖v‖ before v̂ if you need to check whether a vector can be normalized. Use A±B, kA, AB, Av, and A^n for direct matrix arithmetic. Use tr(A) and det(A) for quick scalar matrix properties. Use A^{-1} only when you need the inverse itself. Use Ax=b when solving a system. Use rank(A) and RREF to examine linear dependence. Use κ(A) to judge numerical conditioning. A large condition number indicates sensitivity to small input changes. Use A=LU when no pivoting is needed. If LU fails, the matrix may still be decomposable with pivoting, but this docklet does not perform pivoted LU. Use A=QR and ORTHO for orthogonalization workflows. Use λ, v_λ, P(λ), and SVD as numerical tools. Their outputs should be interpreted with the stated limitations, especially for complex eigenvalues or unstable matrices.

Appendix G.10 — Number Theory

Overview

The Number Theory docklet provides integer-focused tools for divisibility, primes, modular arithmetic, number sequences, combinatorics, congruences, and special number tests. It is designed for calculations where the input values are integers rather than decimal measurements. Most functions require one or more integer values entered into a prompt dialog. Some functions return a single numeric result. Others return a descriptive result, such as a factorization, a divisor list, a congruence solution, or a classification such as “prime”, “composite”, “perfect”, “abundant”, or “palindrome”. The docklet contains 45 exposed functions: even?, □?, ⌊√n⌋, Σd, dr, #d, rev, τ(n), σ(n), s(n), div, φ, λ, μ, prime?, mr? factor, pfact, p±, p_n, π(x), gcd/l, egcd, mod, pow_m, inv_m, log_m, ord, $root_m$, ≡ Fib, n!, nCr, Cat, Tri, Pent, CRT, (a|p), (a|n), [a_0;...], p(n), perf?, a/d?, pal? arm? The calculations are integer-based. Decimal input is not accepted.

Opening and using the docklet

The Number Theory docklet appears as a parked or expandable panel. When parked, the header shows: Number Theory When expanded, the user can scroll through the function list. Selecting a function opens a compact prompt dialog. Each prompt dialog shows: the selected function symbol and name one or more integer input fields a Compute button, a Cancel button an error area, if validation fails The docklet parks itself after a function is selected, while the prompt remains active above the calculator. Tapping outside the prompt does not close it. The dialog is firm: an outside tap gives warning feedback instead of dismissing the prompt. After a successful computation, the prompt closes and the selected function is visually marked as completed. If the user cancels, the selected function is visually marked as cancelled.

Current operand behavior

The Number Theory docklet does not use the current calculator operand as an automatic input. All prompt fields must be entered manually. There is no blank-field behavior where an empty field means “use the current operand.” After a successful calculation, the result may become the current calculator value, depending on the function result type. For single numeric results, the returned value can continue in ordinary calculator expressions. For descriptive results, the calculator may still hold a representative numeric value, but the displayed meaning is the descriptive result.

Prompt dialogs

All functions in this docklet use prompt dialogs. Required fields All fields are required. The Compute button remains disabled until each field contains a parseable integer. Integer-only input The docklet accepts integer text only. Decimal values are not valid. Signed integers are accepted where the field allows them. Examples: -12 0 15 Unsigned/natural-number behavior is applied where a function requires a nonnegative or positive value. Field names The most common field names are: n x a b e p q r a1 n1 a2 n2 These names are mathematical parameter names. Compute button behavior The Compute button:

1. validates the fields
2. runs the selected calculation
3. displays a result or an error
4. closes the dialog if the calculation succeeds

If a domain rule fails, the prompt remains open and shows an error such as: n must be > 0 No inverse, No solution p must be an odd prime Cancel behavior Cancel closes the prompt without computing a result. The current calculator value is not changed. Outside-tap behavior Tapping outside the prompt does not dismiss it. The app gives warning feedback. Keyboard behavior The prompt fields use a numeric keyboard. Return advances to the next field when another field is available. Longer prompts can scroll so the active field remains visible.

Decimal places and rounding

The Number Theory docklet works primarily with integers. However, returned values still pass through the app’s decimal-place setting before display. The decimal places value is clamped to: 0...18 Since the docklet returns integer results, rounding normally has no visible effect. There is no special snapping of near-zero or near-integer values in this docklet. Results are exact within the integer methods and caps used by the function. Very large exact values are sometimes displayed as descriptive text while a smaller representative numeric value is kept as the calculator value. This applies to large Fibonacci numbers, factorials, binomial coefficients, Catalan numbers, and partition numbers.

Locale and separators

The Number Theory docklet does not use decimal separators or list separators for input. All inputs are integers. The docklet may display lists, such as divisors or factorization components, using comma separators or multiplication symbols. These are output formatting conventions, not locale-sensitive input rules. Examples: divisors=1, 2, 3, 6 $2^3 \times 3^2$ solutions: 2, 5, 8

Result behavior

Number Theory functions return either scalar numeric results or descriptive terminal results. Scalar numeric results Scalar results are ordinary numeric values. They can continue in calculator expressions. Examples: ⌊√n⌋, Σd, dr, #d, τ(n), σ(n), s(n), φ, λ, μ, p_n, π(x), mod, pow_m, inv_m, log_m ord, $root_m$, Fib, nCr, Cat, Tri, Pent, (a|p), (a|n) Descriptive or terminal results Some functions return a classification, list, symbolic expression, or multi-part result. These are best read as final descriptive outputs. Examples: even?, □?, div, prime?, mr?, factor, pfact, p±, gcd/l, egcd, CRT, [a_0;…], p(n) perf?, a/d?, pal?, arm? Representative numeric values Some descriptive results also place a numeric value in the calculator display. Examples: even? returns 1 for even and 0 for odd, while the result label says even or odd. prime? returns 1 for prime and 0 for composite, while the result label says prime or composite. factor displays the original number numerically, while the result label shows the factorization. div displays the divisor count numerically, while the result label shows the divisor list. large n!, Fib, nCr, Cat, and p(n) may display or store a reduced numeric representative while the result label shows the full exact value. When a result is descriptive, the label is the meaning the user should read.

Result tags

The Number Theory docklet uses result tags: [NUM] [APPROX] [LABEL], [NUM] [NUM] means the result is intended as a numeric value. Examples: Σd(n) [NUM], φ(n) [NUM], pow_m(a, e;n) [NUM], [LABEL] [LABEL] means the result should be read as a descriptive, symbolic, list-style, classification, or non-scalar result. Examples: prime [LABEL], $2^3 \times 3^2$ [LABEL], gcd=6, lcm=60 [LABEL], solutions: 2, 5 8 [LABEL], [APPROX] [APPROX] is defined by the docklet's result system, but the supplied Number Theory functions do not use it for normal computations. The docklet is primarily exact integer arithmetic, subject to the caps and overflow limits stated below.

Function groups

The docklet is organized into seven logical groups.

Group 1 — Basic and digit: properties, even?, □?, ⌊√n⌋, Σd, dr, #d, rev

Group 2 — Divisors and: totients, τ(n), σ(n), s(n), div, φ, λ, μ

Group 3 — Primes and: factorization, prime?, mr?, factor, pfact, p±, p_n, π(x)

Group 4 — Modular: arithmetic, gcd/l, egcd, mod, pow_m, inv_m, log_m, ord, $root_m$, ≡

Group 5 — Sequences and: combinatorics, Fib, n!, nCr, Cat, Tri, Pent

Group 6 — Advanced / cryptographic: CRT, (a|p), (a|n), [a_0;…], p(n)

Group 7 — Special number: tests, perf?, a/d?, pal?, arm?

Function reference

even? — Is Even/Odd

Purpose: Tests whether an integer is even or odd.

Inputs: n

Parameters: n is a signed integer.

Result: Returns 1 if n is even, 0 if n is odd.

Result behavior: Terminal-label result. The label shows either: even odd

Notes / restrictions: Valid for signed integers.

Example: n = 12 Result = 1 Label = even

□? — Is Perfect Square

Purpose: Tests whether n is a perfect square.

Inputs: n

Parameters: n must be an unsigned integer.

Result: Returns 1 if n is a perfect square, otherwise 0.

Result behavior: Terminal-label result. If n is a square, the label shows: √n=<root> If not, the label shows: ⌊√n⌋=<integer root floor>

Notes / restrictions: Negative input is invalid because the function parses n as unsigned.

Example: n = 49 Result = 1 Label = √n=7

⌊√n⌋ — Integer √n

Purpose: Computes the integer floor of the square root.

Inputs: n

Parameters: n must be an unsigned integer.

Result: Largest integer r such that: $r^2 \le n$

Result behavior: Scalar numeric result.

Notes / restrictions: Negative input is invalid.

Example: n = 50 Result = 7

Σd — Digit Sum

Purpose: Computes the sum of decimal digits.

Inputs: n
Parameters: n is a signed integer. The absolute value is used for digit extraction.
Result: Sum of the decimal digits.
Result behavior: Scalar numeric result.
Example: n = -1234 Result = 10

dr — Digital Root

Purpose: Computes the digital root of an integer.
Inputs: n
Parameters: n is a signed integer. The absolute value is used.
Result: Digital root.
Result behavior: Scalar numeric result.
Formula: if n = 0: dr = 0 otherwise: dr = 1 + (|n| - 1) mod 9
Example: n = 9875 Result = 2

#d — Digit Count

Purpose: Counts decimal digits.
Inputs: n
Parameters: n is a signed integer. The absolute value is counted.
Result: Number of decimal digits.
Result behavior: Scalar numeric result.
Notes / restrictions: 0 has digit count 1.
Example: n = -2048 Result = 4

rev — Reverse Digits

Purpose: Reverses the decimal digits of an integer.
Inputs: n
Parameters: n is a signed integer. The sign is preserved.
Result: Integer with digits reversed.
Result behavior: Scalar numeric result.
Notes / restrictions: If internal reversal would overflow, the function returns 0.
Example: n = -1204 Result = -4021

τ(n) — Divisor Count

Purpose: Computes the number of positive divisors of n.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: Divisor count.
Result behavior: Scalar numeric result.
Formula: If, n = p1^a1 × p2^a2 × ... × pk^ak then: τ(n) = (a1 + 1)(a2 + 1)...(ak + 1)
Notes / restrictions: n must be > 0 For n ≥ 10,000,000, the calculation may run as a heavier background-style computation.
Example: n = 12 Divisors = 1, 2, 3, 4, 6, 12 Result = 6

σ(n) — Divisor Sum

Purpose: Computes the sum of positive divisors of n.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: Sum of divisors.
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 0 If the divisor-sum multiplication overflows internally, the result may fail through overflow handling in related functions.
Example: n = 12 Divisors = 1, 2, 3, 4, 6, 12 Result = 28

s(n) — Aliquot Sum

Purpose: Computes the aliquot sum of n, meaning the sum of proper divisors.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: Aliquot sum.
Result behavior: Scalar numeric result.
Formula: s(n) = σ(n) - n
Notes / restrictions: n must be > 0 Possible error: Overflow

Example: n = 12 σ(12) = 28 s(12) = 16

div — List Divisors

Purpose: Lists the positive divisors of n.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: The display value is the divisor count. The result label shows the full divisor list.
Result behavior: Terminal-label result. Label format: divisors=1, 2, 3, ...
Notes / restrictions: n must be > 0
Example: n = 12 Display result = 6 Label = divisors=1, 2, 3, 4, 6, 12

φ — Euler's φ(n)

Purpose: Computes Euler's totient function.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: Number of integers in 1...n that are coprime to n.
Result behavior: Scalar numeric result.
Formula: φ(n) = n × Π(1 - 1/p) where p ranges over distinct prime factors of n.
Notes / restrictions: n must be > 0
Example: n=9 Result = 6

λ — Carmichael λ(n)

Purpose: Computes the Carmichael function.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: Least universal exponent for the multiplicative group modulo n.
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 0 The function computes λ for prime powers and combines them with least common multiple. Possible error: Overflow (λ too large)
Example: n=8 Result = 2

μ — Möbius μ(n)

Purpose: Computes the Möbius function.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: One of: 1 0 -1
Result behavior: Scalar numeric result. Rules: μ(1) = 1 μ(n) = 0 if n has a repeated prime factor μ(n) = 1 if n is square-free with an even number of prime factors μ(n) = -1 if n is square-free with an odd number of prime factors
Notes / restrictions: n must be > 0
Example: n = 30 30 = 2 × 3 × 5 Result = -1

prime? — Prime Test

Purpose: Tests whether n is prime.
Inputs: n
Parameters: n must be an unsigned integer.
Result: Returns 1 for prime and 0 for composite.
Result behavior: Terminal-label result. Label: prime composite
Notes / restrictions: Uses deterministic Miller-Rabin-style testing for 64-bit unsigned integers with fixed bases.
Example: n = 97 Result = 1 Label = prime

mr? — Miller-Rabin Test

Purpose: Runs the same primality test pathway exposed as a Miller-Rabin-style test.
Inputs: n
Parameters: n must be an unsigned integer.
Result: Returns 1 for prime and 0 for composite.
Result behavior: Terminal-label result. Label: prime composite
Notes / restrictions: The implementation uses deterministic bases for 64-bit input, not a random probabilistic witness set.
Example: n = 91 Result = 0 Label = composite

factor — Factorization

Purpose: Computes the prime factorization of n.
Inputs: n
Parameters: n must be a positive unsigned integer.

Result: The display value is n. The label shows the factorization.
Result behavior: Terminal-label result. Label format: $2^3 \times 3^2 \times 5$
Notes / restrictions: n must be > 0 Uses Pollard Rho factorization and deterministic primality testing. For n ≥ 10,000,000, the factorization may run as a heavier computation.
Example: n = 360 Label = $2^3 \times 3^2 \times 5$

pfact — Prime Factors

Purpose: Lists the distinct prime factors of n.
Inputs: n
Parameters: n must be a positive unsigned integer.
Result: The display value is the number of distinct prime factors. The label shows the set of prime factors.
Result behavior: Terminal-label result. Label format: {2, 3, 5}
Notes / restrictions: n must be > 0
Example: n = 360 Display result = 3 Label = {2, 3, 5}

p± — Next/Prev Prime

Purpose: Finds the previous and next primes around n.
Inputs: n
Parameters: n must be an unsigned integer.
Result: The display value is the next prime. The label shows both previous and next prime.
Result behavior: Terminal-label result. Label format: prev=<previous>, next=<next>
Notes / restrictions: n too large (cap 5000000) If no previous prime exists, previous is shown as: none
Example: n = 20 Display result = 23 Label = prev=17, next=23

p_n — nth Prime

Purpose: Computes the nth prime.
Inputs: n
Parameters: n must be a positive integer.
Result: The nth prime.
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 0 n too large (cap 200000)
Example: n = 10 Result = 29

π(x) — Prime Count π(x)

Purpose: Counts primes less than or equal to x.
Inputs: x
Parameters: x must be an unsigned integer.
Result: Number of primes ≤ x.
Result behavior: Scalar numeric result.
Notes / restrictions: x too large (cap 5000000) Uses a sieve up to x.
Example: x = 10 Primes = 2, 3, 5, 7 Result = 4

gcd/l — GCD / LCM

Purpose: Computes greatest common divisor and least common multiple.
Inputs: ab
Parameters: a and b are signed integers.
Result: The display value is the GCD. The label shows both GCD and LCM.
Result behavior: Terminal-label result. Label format: gcd=<g>, lcm=<l> If LCM overflows: gcd=<g>, lcm=overflow
Example: a = 12 b = 18 Display result = 6 Label = gcd=6, lcm=36

egcd — Extended GCD

Purpose: Computes the greatest common divisor and Bézout coefficients.
Inputs: ab
Parameters: a and b are signed integers.
Result: The display value is the GCD. The label shows: g=<g>, x=<x>, y=<y> such that: ax + by = g
Result behavior: Terminal-label result.
Example: a = 30 b = 12 Label = g=6, x=1, y=-2 because: 30(1) + 12(-2) = 6

mod — a mod n

Purpose: Computes normalized modulo.
Inputs: an
Parameters: • a: signed integer • n: modulus
Result: Normalized residue in the range: 0...|n|-1

Result behavior: Scalar numeric result.
Notes / restrictions: n must be ≠ 0
Example: a = -3 n = 10 Result = 7

pow_m — Modular Power

Purpose: Computes modular exponentiation.
Inputs: aen
Parameters: • a: base • e: exponent • n: modulus
Result: a^e mod n
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 0 e must be ≥ 0 Possible error: Invalid pow
Example: a=2 e = 10 n = 1000 Result = 24

inv_m — Modular Inverse

Purpose: Computes the modular inverse of a modulo n.
Inputs: an
Parameters: • a: integer • n: positive modulus
Result: Integer x such that: ax ≡ 1 (mod n)
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 0 No inverse An inverse exists only when: gcd(a,n) = 1
Example: a=3 n = 11 Result = 4 because: 3 × 4 ≡ 1 (mod 11)

log_m — Discrete Log

Purpose: Solves a discrete logarithm problem.
Inputs: abn
Parameters: • a: base • b: target • n: modulus
Result: A nonnegative integer x such that: a^x ≡ b (mod n)
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 1 gcd(a,n) must be 1 n too large (cap 2000000) No solution Uses a baby-step giant-step method.
Example: a=2 b=8 n = 13 Result = 3 because: $2^3 \equiv 8 \pmod{13}$

ord — Order of a mod n

Purpose: Computes the multiplicative order of a modulo n.
Inputs: an
Parameters: • a: integer • n: modulus
Result: Smallest positive integer k such that: a^k ≡ 1 (mod n)
Result behavior: Scalar numeric result.
Notes / restrictions: n must be > 1 No order (need gcd(a,n)=1)
Example: a=2 n=7 Result = 3 because: $2^3 \equiv 1 \pmod{7}$

$root_m$ — Primitive Root

Purpose: Finds a primitive root modulo an odd prime p.
Inputs: p
Parameters: p must be an odd prime.
Result: A primitive root modulo p.
Result behavior: Scalar numeric result.
Notes / restrictions: p must be an odd prime p too large (cap 20000000) No primitive root (need prime p) The function searches for the first valid primitive root.
Example: p=7 Result = 3

≡ — Solve ax≡b (mod n)

Purpose: Solves a linear congruence.
Inputs: abn
Parameters: • a: coefficient • b: target • n: modulus
Result: A solution to: ax ≡ b (mod n)
Result behavior: Scalar result if there is one residue class. Terminal-label result if multiple solutions are produced.
Notes / restrictions: n must be > 0 No solution, No inverse If gcd(a,n) divides b, solutions exist. If multiple solutions exist, the label shows all listed solutions.
Example: a=2 b=4 n=6 Label = solutions: 2, 5

Fib — Fibonacci F_n

Purpose: Computes the nth Fibonacci number.

Inputs: n

Parameters: n must be a nonnegative integer.

Result: Fibonacci number F_n.

Result behavior: Scalar-listed result. For large values, the label shows the full exact integer.

Notes / restrictions: n must be ≥ 0 n too large (cap 50000) For large results, the displayed numeric value may be a representative part of the big integer while the label contains the full exact value.

Example: n = 10 Result label = 55

n! — Factorial n!

Purpose: Computes factorial.

Inputs: n

Parameters: n must be a nonnegative integer.

Result: n! = 1 × 2 × ... × n

Result behavior: Scalar for small exact values; terminal-label result for large exact values.

Notes / restrictions: n must be ≥ 0 n too large (cap 10000) For n ≤ 18, the full result fits in standard integer flow. For larger n, the full value is shown as a label.

Example: n=5 Result = 120

nCr — Binomial ^nC_r

Purpose: Computes a binomial coefficient.

Inputs: nr

Parameters: • n: total count • r: selected count

Result: C(n,r)

Result behavior: Scalar result for manageable values; label carries full exact value for larger results.

Notes / restrictions: n must be ≥ 0 r must be 0..n n too large (cap 5000)

Example: n = 10 r=3 Result = 120

Cat — Catalan C_n

Purpose: Computes the nth Catalan number.

Inputs: n

Parameters: n must be a nonnegative integer.

Result: $C_n = (1 / (n + 1)) \times C(2n,n)$

Result behavior: Scalar result for manageable values; label carries full exact value for larger results.

Notes / restrictions: n must be ≥ 0 n too large (cap 2000)

Example: n=4 Result = 14

Tri — Triangular T_n

Purpose: Computes the nth triangular number.

Inputs: n

Parameters: n must be a nonnegative integer.

Result: $T_n = n(n + 1) / 2$

Result behavior: Scalar numeric result.

Notes / restrictions: n must be ≥ 0

Example: n = 10 Result = 55

Pent — Pentagonal P_n

Purpose: Computes the nth pentagonal number.

Inputs: n

Parameters: n must be a nonnegative integer.

Result: $P_n = n(3n - 1) / 2$

Result behavior: Scalar numeric result.

Notes / restrictions: n must be ≥ 0

Example: n=5 Result = 35

CRT — Chinese Remainder

Purpose: Solves a two-congruence Chinese Remainder problem.

Inputs: a1 n1 a2 n2

Parameters: x ≡ a1 (mod n1) x ≡ a2 (mod n2)

Result: A solution residue and combined modulus.

Result behavior: Terminal-label result. Label format: x≡<value> (mod <modulus>)

Notes / restrictions: moduli must be > 0 No solution, No inverse The congruences must be compatible: a2 - a1 must be divisible by gcd(n1,n2)

Example: x ≡ 2 (mod 3) x ≡ 3 (mod 5), Label = x≡8 (mod 15)

(a|p) — Legendre Symbol

Purpose: Computes the Legendre symbol.

Inputs: ap

Parameters: • a: integer • p: odd prime

Result: -1, 0, or 1

Result behavior: Scalar numeric result.

Notes / restrictions: p must be an odd prime

Example: a=2 p=7 Result = 1

(a|n) — Jacobi Symbol

Purpose: Computes the Jacobi symbol.

Inputs: an

Parameters: • a: integer • n: odd positive integer

Result: -1, 0, or 1

Result behavior: Scalar numeric result.

Notes / restrictions: n must be odd and > 0 The Jacobi symbol does not by itself prove that a is a quadratic residue when n is composite.

Example: a=5 n = 21 Result = 1

[a₀;…] — Cont. Fraction

Purpose: Computes the continued fraction expansion of a rational number.

Inputs: pq

Parameters: p/q

Result: The display value is the first continued-fraction term. The label shows the continued fraction list.

Result behavior: Terminal-label result. Label format: [a0; a1; a2; ...]

Notes / restrictions: q must be ≠ 0 The expansion is capped internally after more than 64 terms.

Example: p = 415 q = 93 Label = [4; 2; 6; 7]

p(n) — Partition p(n)

Purpose: Computes the integer partition number.

Inputs: n

Parameters: n must be a nonnegative integer.

Result: Number of ways to write n as a sum of positive integers, ignoring order.

Result behavior: Terminal-label result for larger values; scalar-style for small values.

Notes / restrictions: n must be ≥ 0 n too large (cap 1500) For n ≤ 30, the full result is small enough to use directly. For larger values, the full exact value is shown in the label.

Example: n=5 Result = 7

perf? — Is Perfect

Purpose: Tests whether n is a perfect number.

Inputs: n

Parameters: n must be a positive unsigned integer.

Result: Returns 1 if perfect, otherwise 0.

Result behavior: Terminal-label result. Label: perfect not perfect

Formula: n is perfect if s(n) = n

Notes / restrictions: n must be > 0 Overflow

Example: n = 28 Result = 1 Label = perfect

a/d? — Is Abundant/Def.

Purpose: Classifies a number as abundant, deficient, or perfect.

Inputs: n

Parameters: n must be a positive unsigned integer.

Result: The display value is the aliquot sum s(n). The label shows the classification.

Result behavior: Terminal-label result. Label format: s(n)=<value> → abundant s(n)=<value> → deficient s(n)=<value> → perfect

Notes / restrictions: n must be > 0 Overflow

Example: n = 12 s(12) = 16 Label = s(n)=16 → abundant

pal? — Is Palindrome

Purpose: Tests whether the decimal digits of n form a palindrome.

Inputs: n

Parameters: n is a signed integer. The absolute value is tested.

Result: Returns 1 if palindrome, otherwise 0.

Result behavior: Terminal-label result. Label: palindrome not palindrome

Example: n = -1221 Result = 1 Label = palindrome

arm? — Is Armstrong

Purpose: Tests whether n is an Armstrong number in base 10.

Inputs: n

Parameters: n is a signed integer. The absolute value is tested.

Result: Returns 1 if the number equals the sum of its digits raised to the number of digits, otherwise 0.

Result behavior: Terminal-label result.

Formula: For a number with k digits: n = Σ d^k Label format: Σ d^k = <sum>

Example: n = 153 $1^3 + 5^3 + 3^3 = 153$ Result = 1

Errors and limitations

Integer-only input The Number Theory docklet accepts integers only. Invalid examples: 3.5 2,4 1/2 abc Typical error: Invalid n, Invalid a, b, Invalid input, Unsigned input restrictions Some functions parse n, x, or p as unsigned values. Negative input is invalid for these functions. Examples include: □?, ⌊√n⌋, τ(n) σ(n) s(n) div φ, λ, μ prime?, factor, pfact p±, π(x), $root_m$ Positive-value restrictions Many functions require positive input: n must be > 0 x too large p must be an odd prime moduli must be > 0 Zero modulus restrictions Modulo-related functions reject zero modulus where applicable: n must be ≠ 0 n must be > 0 n must be > 1 Modular inverse restrictions A modular inverse exists only when the value and modulus are coprime. Typical error: No inverse Multiplicative order restrictions The order of a mod n requires: n>1 gcd(a,n) = 1 Otherwise: No order (need gcd(a,n)=1) Discrete logarithm restrictions Discrete logarithm has these restrictions: n must be > 1 gcd(a,n) must be 1 n too large (cap 2000000) No solution It uses baby-step giant-step and is capped for performance. Prime and factorization limits Factorization uses Pollard Rho and deterministic 64-bit primality testing. Very large inputs can be expensive. Some operations are explicitly capped: p±: n ≤ 5,000,000 p_n: n ≤ 200,000 π(x): x ≤ 5,000,000 $root_m$: p ≤ 20,000,000 Big integer limits The docklet uses a custom big-natural-number system for large exact values in: Fib, n!, nCr, Cat, p(n) Caps: Fib: n ≤ 50,000 n!: n ≤ 10,000 nCr: n ≤ 5,000 Cat: n ≤ 2,000 p(n): n ≤ 1,500 For large exact results, the result label should be treated as the authoritative full output. Overflow risks Some functions use fixed-width integer arithmetic. Overflow is guarded in some places and not fully avoidable in all conceptual cases. Possible overflow-related messages include: Overflow, Overflow (λ too large), lcm=overflow, Continued fraction limit Continued fraction expansion is capped after more than 64 terms. Classification functions return 1 or 0 Boolean tests return numeric values: 1 = true 0 = false The label gives the readable meaning. Examples: prime, composite, perfect, not perfect, palindrome, not palindrome

Practical usage notes

Use prime? or mr? for primality testing. In this docklet, both use the same deterministic 64-bit testing behavior. Use factor when the full factorization is needed. Use pfact when only the distinct prime factors are needed. Use τ(n), σ(n), and s(n) for divisor arithmetic. Use div only when the actual divisor list is useful. Use φ, λ, and μ for multiplicative number-theory functions. These depend on factorization and may be heavier for large inputs. Use mod, pow_m, inv_m, ord, and ≡ for modular arithmetic. Check the modulus restrictions carefully. Use CRT for two congruences only. It does not accept a general list of congruences. Use Fib, n!, nCr, Cat, and p(n) for exact integer sequence values. For large results, read the label as the full result. Use perf?, a/d?, pal?, and arm? for number classification. For descriptive results, do not rely only on the numeric display. Read the result label.

Appendix G.11 — Statistics

Overview

The Statistics docklet provides descriptive statistics, probability tools, confidence intervals, hypothesis-test helpers, and simple regression utilities. Unlike the Trigonometry docklet, Statistics is a prompt-based docklet. Selecting a function opens a compact dialog where the user enters the required data, parameters, or test values. The docklet then computes the requested statistic and returns a result to the calculator. The Statistics docklet contains 31 exposed functions: x̄, MED, MODE, MINMAX, RNG, Σx, SD, VAR, CV, Q1–Q3, IQR, P, Φ, $Φ^{-1}$, Z, T, t-CDF BINOM, POIS, CI-μ, CI-p, TTEST, T2, ZTEST, CHI2, PVAL, r, LINREG, R2, n, 5NUM The docklet is intended for users who need quick statistical calculations without building a spreadsheet or entering full formulas manually.

Opening and using the docklet

The Statistics docklet appears as a parked or expandable panel. When parked, the header shows: Statistics When expanded, the user selects one of the listed statistics functions. Tapping a function opens a prompt dialog. The prompt dialog contains: a title showing the selected symbol and label one or more input fields a Compute button, a Cancel button an error message area, when validation fails The dialog is firm. Tapping outside the dialog does not dismiss it. Instead, the app plays a warning sound and gives light haptic feedback. After a function is computed successfully:

1. the result is rounded according to the decimal places setting
2. the result is committed into the calculator flow when appropriate
3. the display and ticker are updated

4. the prompt closes
5. the selected function symbol is visually marked as computed
6. the docklet remains in its parked/expanded state according to the surrounding docklet behavior

If the user cancels:

1. the prompt closes
2. the selected function is visually marked as cancelled
3. no statistic is computed
4. no result is committed

Current operand behavior

The Statistics docklet does not use the current calculator operand as a default input. All required values must be entered manually into the prompt fields. There is no blank = operand behavior in this docklet. The current operand is only affected after a successful computation. When a scalar result is produced, the computed value becomes the calculator’s current numeric result through the normal result-commit pathway.

Prompt dialogs

The Statistics docklet uses prompt dialogs for every exposed function. Required fields Every prompt field is required unless explicitly handled by a mode rule. In this code, the Statistics docklet does not define optional blank fields. The Compute button is disabled until all fields contain parseable input. Field validation The docklet validates fields according to field title: Field type Expected input Data, X, Y list of numeric values n, df, trials, k, mode fields integer other numeric fields decimal number Compute button behavior The Compute button:

1. checks that the input is valid
2. plays the menu sound
3. runs the selected calculation
4. commits the result if successful
5. leaves the dialog open if validation fails

If validation fails during computation, the prompt displays an error message and plays the warning sound. Cancel behavior Cancel: marks the selected function as cancelled clears the active prompt state closes the dialog does not compute anything does not change the calculator result Outside-tap behavior The prompt is firm. Tapping outside the prompt: • does not dismiss the dialog • plays a warning sound • triggers light haptic feedback Keyboard behavior The prompt fields use numeric keyboard entry. The Return key advances to the next field where applicable. In longer dialogs, the active field is scrolled into view. Dragging inside the prompt’s scrollable field area dismisses the keyboard.

Decimal places and rounding

The Statistics docklet uses the app’s decimalPlaces setting for result rounding and display formatting. The decimal places value is clamped to: 0...18 Rounding uses plain rounding through NSDecimalRound. The active rounding process is:

1. compute the statistic as Double
2. convert the finite result to Decimal
3. round using decimalPlaces
4. normalize -0 to 0
5. commit the rounded value to the calculator

There is no special near-integer snapping in this docklet. If a computed value cannot be converted into a finite decimal result, the docklet reports: Invalid result

Locale and separators

The Statistics docklet is locale-sensitive. In comma-decimal mode: • decimal separator is comma • list separator is semicolon In dot-decimal mode: • decimal separator is dot • list separator is comma Dot-decimal locale list entry In dot-decimal locales, a data list is entered with commas: 1.2,2.3,3.4 Comma-decimal locale list entry In comma-decimal locales, a data list is entered with semicolons: 1,2;2,3;3,4 This distinction is important. In comma-decimal locales, comma cannot safely separate list items because comma is already used inside numbers. X and Y lists Regression and correlation tools use the same list rules. Dot-decimal example: X = 1,2,3,4 Y = 2,4,6,8 Comma-decimal example: X = 1;2;3;4 Y = 2;4;6;8 For decimal values in comma-decimal mode: X = 1,5;2,5;3,5 Y = 2,0;4,0;6,0 Ticker localization Ticker text is also localized. Decimal separators appearing between digits are adjusted to match the active decimal separator convention.

Result behavior

Statistics functions can return either scalar results or terminal label-style results. Scalar flow results Scalar results are intended to continue in calculator expressions. The following functions are marked as scalar flow results: x̄, MED, RNG, Σx, SD, VAR, CV, IQR, P, Φ, Φ^{-1}, Z, T, t-CDF, BINOM, POIS, TTEST, ZTEST, CHI2, PVAL, r, R2, n These results behave like ordinary numeric calculator results. Terminal-label results Terminal-label results contain descriptive or multi-value output. They are not simple scalar-only answers, even though the docklet may still commit one representative numeric value internally. The following functions are marked as terminal-label results: MODE, MINMAX, Q1–Q3, CI-μ, CI-p, T2, LINREG, 5NUM Examples: MIN=1, MAX=9, Q1=2, Q3=8, CI=[4.12, 5.88], y = 2x + 1, Ticker result The ticker normally shows the function label, such as: x̄(n=5) When a function produces a richer result, the ticker may show an override, such as: MIN=1, MAX=9 or: CI=[4.12, 5.88] Most results use [NUM] by default. The two-sample t-test explicitly uses [APPROX].

Result tags

The Statistics docklet defines: [NUM] [APPROX] [LABEL], [NUM] [NUM] means the result is a numeric statistical result. Most Statistics functions use [NUM], even when the underlying calculation is floating-point. Examples: x̄(n=5) [NUM], SD(n=5), Φ(x;μ,σ), [APPROX] [APPROX] means the result is explicitly approximate. In this code, T2 uses [APPROX] because the two-sample t-test computes an approximate Welch-style degrees of freedom value.

Example: df≈17.42 [APPROX] [LABEL] [LABEL] is defined, but the supplied Statistics code does not actively use it as the default tag for errors or descriptive results. Errors appear as prompt error messages rather than ticker label results. Descriptive outputs such as confidence intervals and regression equations are represented through ticker overrides, while their tags are determined by the commitValue call.

Function groups

The docklet's logical grouping follows the visible list and separator indices.

Group 1 — Basic descriptive: statistics, x̄, MED, MODE, MINMAX, RNG, Σx

Group 2 — Dispersion and percentiles:, SD, VAR, CV, Q1–Q3, IQR, P

Group 3 — Probability: distributions and scores, Φ, Φ^{-1}, Z, T, t-CDF, BINOM, POIS

Group 4 — Confidence: intervals and tests, CI-μ, CI-p, TTEST, T2, ZTEST, CHI2, PVAL

Group 5 — Correlation and regression:, r, LINREG, R2

Group 6 — Count and summary:, n, 5NUM

Function reference

x̄ — Mean

Purpose: Computes the arithmetic mean of a numeric data list.

Inputs: Data

Parameters: Data is a list of finite numeric values.

Result: Mean of the values.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: x̄ = Σx / n

Notes / restrictions: Data must contain at least one valid number.

Example: Data = 2,4,6,8 Result = 5 Ticker = x̄(n=4)

MED — Median

Purpose: Computes the median of a numeric data list.

Inputs: Data

Parameters: Data is sorted internally before the median is computed.

Result: Middle value for odd-length lists; average of two middle values for even-length lists.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: odd n: median = sorted[n/2] even n: median = (sorted[n/2 - 1] + sorted[n/2]) / 2

Notes / restrictions: Data must contain at least one valid number.

Example: Data = 1,4,9 Result = 4 Ticker = MED(n=3) [NUM]

MODE — Mode

Purpose: Computes the mode of a numeric data list.

Inputs: Data

Parameters: Values are quantized internally at a scale of 1,000,000,000,000 for counting.

Result: Most frequent value.

Result behavior: Terminal-label result. Tagged [NUM] by the commit function.

Notes / restrictions: If no value occurs more than once, the docklet reports: No mode If multiple values share the highest frequency, the smallest quantized value is selected.

Example: Data = 1,2,2,3 Result = 2 Ticker = MODE(n=4) [NUM]

MINMAX — Min / Max
Purpose: Finds the minimum and maximum of a numeric data list.
Inputs: Data
Parameters: Data is sorted internally.
Result: The committed numeric value is the maximum. The ticker shows both minimum and maximum.
Result behavior: Terminal-label result. Tagged [NUM]. Ticker override format: MIN=<minimum>, MAX=<maximum>
Example: Data = 5,2,9 Display result = 9 Ticker = MIN=2, MAX=9 [NUM]

RNG — Range
Purpose: Computes the range of a numeric data list.
Inputs: Data
Parameters: Range is maximum minus minimum.
Result: Scalar numeric value.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: range = max(x) - min(x)
Example: Data = 5,2,9 Result = 7 Ticker = RNG(n=3) [NUM]

Σx — Sum
Purpose: Computes the sum of a numeric data list.
Inputs: Data
Parameters: All values are added.
Result: Scalar numeric value.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: Σx = x1 + x2 + ... + xn
Example: Data = 2,4,6 Result = 12 Ticker = Σx(n=3) [NUM]

SD — Std. Dev.
Purpose: Computes the sample standard deviation.
Inputs: Data
Parameters: Uses sample variance with denominator n - 1.
Result: Sample standard deviation.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: s = sqrt(Σ(x - x̄)^2 / (n - 1))
Notes / restrictions: Requires at least 2 values. Otherwise: Need n ≥ 2
Example: Data = 2,4,6 Result ≈ 2 Ticker = SD(n=3) [NUM]

VAR — Variance
Purpose: Computes the sample variance.
Inputs: Data
Parameters: Uses denominator n - 1.
Result: Sample variance.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: s^2 = Σ(x - x̄)^2 / (n - 1)
Notes / restrictions: Requires at least 2 values. Otherwise: Need n ≥ 2
Example: Data = 2,4,6 Result = 4 Ticker = VAR(n=3) [NUM]

CV — Coeff. Var.
Purpose: Computes the coefficient of variation.
Inputs: Data
Parameters: Uses sample standard deviation divided by the absolute mean.
Result: Scalar numeric value.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: CV = s / |x̄|
Notes / restrictions: Requires at least 2 values. Mean must not be zero. Possible errors: Need n ≥ 2 Mean is 0
Example: Data = 8,10,12 Mean = 10 SD = 2 CV = 0.2

Q1–Q3 — Quartiles
Purpose: Computes the first and third quartiles.
Inputs: Data
Parameters: Uses the docklet's linear percentile method.
Result: The committed numeric value is Q3. The ticker shows both Q1 and Q3.
Result behavior: Terminal-label result. Tagged [NUM].

Formula: Q1 = percentile(Data, 0.25) Q3 = percentile(Data, 0.75) Ticker override format: Q1=<value>, Q3=<value>
Example: Data = 1,2,3,4,5 Ticker = Q1=2, Q3=4 [NUM]

IQR — IQR

Purpose: Computes the interquartile range.
Inputs: Data
Parameters: Uses Q3 minus Q1, with quartiles computed by the docklet's linear percentile method.
Result: Scalar numeric value.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: IQR = Q3 - Q1
Example: Data = 1,2,3,4,5 Result = 2 Ticker = IQR(n=5) [NUM]

P — Percentile

Purpose: Computes a percentile from a numeric data list.
Inputs: Data, p
Parameters: p may be entered as either: 0..1 or: 0..100 If p > 1, the docklet divides it by 100.
Result: Percentile value.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: h = (n - 1) × p The docklet linearly interpolates between adjacent sorted values.
Notes / restrictions: After conversion, p must satisfy: 0≤p≤1 Otherwise: p must be 0..1 or 0..100 Default field value: p = 0.5
Example: Data = 1,2,3,4,5 p = 0.5 Result = 3 Ticker = P(0.5) [NUM]

Φ — Normal CDF

Purpose: Computes the normal cumulative distribution function.
Inputs: xμσ
Parameters: • x: value to evaluate • μ: mean • σ: standard deviation
Result: Probability P(X ≤ x) for a normal distribution.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: z = (x - μ) / σ Φ = normalCDF(z)
Notes / restrictions: σ must be greater than zero. Error: σ must be > 0 The normal CDF uses an internal error-function approximation.
Example: x=0 μ=0 σ=1 Result ≈ 0.5 Ticker = Φ(x;μ,σ) [NUM]

Φ^{-1} — Normal Inverse

Purpose: Computes the inverse standard normal CDF.
Inputs: p
Parameters: p is a probability.
Result: The z-value whose standard normal cumulative probability is p.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: $z = \Phi^{-1}(p)$
Notes / restrictions: Input must satisfy: 0<p<1 Error: p must be 0<p<1 Default field value: p = 0.5 The implementation uses a rational approximation.
Example: p = 0.5 Result ≈ 0 Ticker = $\Phi^{-1}(p)$ [NUM]

Z — z-Score

Purpose: Computes a z-score.
Inputs: xμσ
Parameters: • x: observed value • μ: mean • σ: standard deviation
Result: Scalar z-score.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: z = (x - μ) / σ
Notes / restrictions: σ must be greater than zero. Error: σ must be > 0
Example: x = 110 μ = 100 σ = 15 Result ≈ 0.66666667 Ticker = Z(x;μ,σ) [NUM]

T — t-Score

Purpose: Computes a one-sample t statistic.
Inputs: $\bar{x}$ μ s n
Parameters: $\bar{x}$: sample mean, μ: hypothesized mean, s: sample standard deviation n: sample size
Result: Scalar t statistic.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: T = ($\bar{x}$ - μ) / (s / sqrt(n))

Notes / restrictions: n must be ≥ 2 s must be > 0

Example: x̄ = 105 μ = 100 s = 10 n = 25 Result = 2.5 Ticker = T [NUM]

t-CDF — t-Distribution

Purpose: Computes the cumulative probability of a t statistic.

Inputs: t df

Parameters: • t: t statistic • df: degrees of freedom

Result: Probability P(T ≤ t).

Result behavior: Scalar flow result. Tagged [NUM].

Formula: Uses a regularized incomplete beta function internally.

Notes / restrictions: df must be greater than zero. Error: df must be > 0 If the numerical result is not finite: Invalid result

Example: t=0 df = 10 Result ≈ 0.5 Ticker = t-CDF [NUM]

BINOM — Binomial

Purpose: Computes a binomial probability mass value.

Inputs: nkp

Parameters: • n: number of trials • k: number of successes • p: success probability

Result: Probability of exactly k successes in n trials.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: P(X = k) = C(n,k) p^k (1-p)^(n-k) Implementation note: The docklet computes this using log-gamma terms for numerical stability.

Notes / restrictions: n must be ≥ 0 k must be 0..n p must be 0..1

Example: n = 10 k=3 p = 0.5 Result ≈ 0.1171875 Ticker = BINOM [NUM]

POIS — Poisson

Purpose: Computes a Poisson probability mass value.

Inputs: k, λ

Parameters: • k: event count • λ: rate parameter

Result: Probability of exactly k events.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: P(X = k) = e^(-λ) λ^k / k! Implementation note: The docklet computes this using log-gamma terms.

Notes / restrictions: k must be ≥ 0 λ must be ≥ 0

Example: k=2 λ=3 Result ≈ 0.22404181 Ticker = POIS [NUM]

CI-μ — CI for Mean

Purpose: Computes a confidence interval for a mean.

Inputs: x̄ s n, Confidence

Parameters: x̄: sample mean, s: sample standard deviation, n: sample size Confidence: confidence level

Result: The committed numeric value is the upper interval bound. The ticker shows the interval.

Result behavior: Terminal-label result. Tagged [NUM]. Formula used: alpha = 1 - Confidence z = Φ^{-1}(1 - alpha/2) SE = s / sqrt(n) CI = [x̄ - zSE, x̄ + zSE]

Notes / restrictions: n must be ≥ 2 s must be > 0 Confidence must be (0,1) Default confidence: 0.95 Implementation note: This function uses a normal critical value, not a t critical value.

Example: x̄ = 100 s = 10 n = 25 Confidence = 0.95 Ticker = CI=[96.080..., 103.919...] [NUM]

CI-p — CI for Proportion

Purpose: Computes a confidence interval for a sample proportion.

Inputs: p̂ n, Confidence

Parameters: • p̂: sample proportion • n: sample size • Confidence: confidence level

Result: The committed numeric value is the upper interval bound. The ticker shows the interval.

Result behavior: Terminal-label result. Tagged [NUM]. Formula used: alpha = 1 - Confidence z = Φ^{-1}(1 - alpha/2) SE = sqrt(p̂(1-p̂) / n) CI = [p̂ - zSE, p̂ + zSE] The interval is clipped to [0,1].

Notes / restrictions: n must be > 0 p̂ must be 0..1 Confidence must be (0,1) Default confidence: 0.95

Example: p̂ = 0.4 n = 100 Confidence = 0.95 Ticker = CI=[0.3039..., 0.4960...] [NUM]

TTEST — t-Test (1-sample)

Purpose: Computes the one-sample t-test statistic.

Inputs: x̄ μ0 s n

Parameters: x̄: sample mean, μ0: null mean, s: sample standard deviation, n: sample size

Result: Scalar t statistic.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: t = (x̄ - μ0) / (s / sqrt(n)) df = n - 1 Ticker label includes degrees of freedom: TTEST(df=<n-1>)

Notes / restrictions: n must be ≥ 2 s must be > 0

Example: x̄ = 105 μ0 = 100 s = 10 n = 25 Result = 2.5 Ticker = TTEST(df=24) [NUM]

T2 — t-Test (2-sample)

Purpose: Computes a two-sample t statistic and approximate degrees of freedom.

Inputs: x̄1 s1 n1 x̄2 s2 n2

Parameters: x̄1: first sample mean, s1: first sample standard deviation n1: first sample size, x̄2: second sample mean s2: second sample standard deviation, n2: second sample size

Result: The committed numeric value is the t statistic. The ticker may show approximate degrees of freedom.

Result behavior: Terminal-label result. Tagged [APPROX].

Formula: v1 = s1² / n1 v2 = s2² / n2 t = (x̄1 - x̄2) / sqrt(v1 + v2) df ≈ (v1 + v2)² / (v1²/(n1-1) + v2²/(n2-1))

Notes / restrictions: n1,n2 must be ≥ 2 s1,s2 must be > 0

Example: x̄1 = 10 s1 = 2 n1 = 20 x̄2 = 8 s2 = 3 n2 = 18 Ticker = df≈... [APPROX]

ZTEST — z-Test

Purpose: Computes a z-test statistic for a sample mean when population standard deviation is supplied.

Inputs: x̄ μ0 σ n

Parameters: x̄: sample mean, μ0: null mean, σ: population standard deviation, n: sample size

Result: Scalar z statistic.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: z = (x̄ - μ0) / (σ / sqrt(n))

Notes / restrictions: n must be > 0 σ must be > 0

Example: x̄ = 105 μ0 = 100 σ = 15 n = 36 Result = 2 Ticker = ZTEST [NUM]

CHI2 — χ² Test

Purpose: Computes an approximate chi-square cumulative probability.

Inputs: χ² df

Parameters: • χ²: chi-square statistic • df: degrees of freedom

Result: Approximate chi-square CDF value.

Result behavior: Scalar flow result. Tagged [NUM]. Formula / method: Uses a Wilson-Hilferty normal approximation: z = ((x / df)^(1/3) - (1 - 2/(9df))) / sqrt(2/(9df)) CDF ≈ Φ(z)

Notes / restrictions: df must be > 0 χ² must be ≥ 0 If the result is not finite: Invalid result

Example: χ² = 5 df = 2 Ticker = CHI2-CDF [NUM]

PVAL — p-Value

Purpose: Computes a p-value from a selected test statistic mode.

Inputs: Mode (0 z, 1 t, 2 χ²), Statistic, df

Parameters: Mode = 0: z statistic Mode = 1: t statistic Mode = 2: chi-square statistic Statistic: test statistic df: required for t and chi-square modes

Result: Scalar p-value.

Result behavior: Scalar flow result. Tagged [NUM]. Mode behavior: 0: two-sided z p-value 1: two-sided t p-value 2: upper-tail chi-square p-value Formulas: z mode: p = 2 × (1 - Φ(|z|)) t mode: p = 2 × min(tCDF(t), 1 - tCDF(t)) χ² mode: p = 1 - chiSquareCDF(χ²)

Notes / restrictions: • mode must be 0, 1, or 2 • df is required for mode 1 and mode 2 • df must be greater than zero for t and chi-square modes Possible errors: Mode must be 0, 1, 2, df required df must be > 0 Invalid input Default mode: 0

Example: Mode = 0 Statistic = 1.96 Result ≈ 0.05 Ticker = PVAL(z) [NUM]

r — Correlation

Purpose: Computes Pearson correlation between paired numeric lists.

Inputs: X, Y

Parameters: • X: list of x values • Y: list of y values

Result: Pearson correlation coefficient.

Result behavior: Scalar flow result. Tagged [NUM].

Formula: r = Σ((x - x̄)(y - ȳ)) / sqrt(Σ(x - x̄)² Σ(y - ȳ)²)

Notes / restrictions: X and Y must have same length Need at least 2 pairs If either list has zero variance: Invalid r

Example: X = 1,2,3 Y = 2,4,6 Result = 1 Ticker = r(n=3) [NUM]

LINREG — Linear Reg.

Purpose: Computes simple linear regression.

Inputs: X, Y

Parameters: • X: list of x values • Y: list of y values

Result: The committed numeric value is the slope. The ticker shows the regression equation.

Result behavior: Terminal-label result. Tagged [NUM].

Formula: slope = Σ((x - x̄)(y - ȳ)) / Σ(x - x̄)² intercept = ȳ - slope × x̄ Ticker override format: y = <slope>x + <intercept>
Notes / restrictions: X and Y must have same length Need at least 2 pairs If all X values are equal: Invalid regression
Example: X = 1,2,3 Y = 2,4,6 Display result = 2 Ticker = y = 2x + 0 [NUM]

R2 — R²

Purpose: Computes the coefficient of determination from Pearson correlation.
Inputs: X, Y
Parameters: • X: list of x values • Y: list of y values
Result: r²
Result behavior: Scalar flow result. Tagged [NUM].
Formula: R² = r²
Notes / restrictions: X and Y must have same length Need at least 2 pairs If correlation is invalid: Invalid r
Example: X = 1,2,3 Y = 2,4,6 r=1 R² = 1 Ticker = R²(n=3) [NUM]

n — n (Count)

Purpose: Counts the number of values in a data list.
Inputs: Data
Parameters: Data is a list of numeric values.
Result: Number of values.
Result behavior: Scalar flow result. Tagged [NUM].
Formula: n = count(Data)
Example: Data = 4,8,12 Result = 3 Ticker = n [NUM]

5NUM — 5-Num Summary

Purpose: Computes a five-number summary.
Inputs: Data
Parameters: Data is sorted internally.
Result: The committed numeric value is the maximum. The ticker shows the complete summary.
Result behavior: Terminal-label result. Tagged [NUM].
Formula: MIN = minimum Q1 = percentile(0.25) MED = median Q3 = percentile(0.75) MAX = maximum Ticker override format: MIN=<min>, Q1=<q1>, MED=<median>, Q3=<q3>, MAX=<max>
Example: Data = 1,2,3,4,5 Ticker = MIN=1, Q1=2, MED=3, Q3=4, MAX=5 [NUM]

Errors and limitations

Invalid data The docklet reports: Invalid data when a data list cannot be parsed or is empty. Typical causes: wrong list separator, decimal separator mismatch letters or unsupported characters empty input, malformed numbers, Locale separator errors In dot-decimal mode, use comma as the list separator: 1.2,2.3,3.4 In comma-decimal mode, use semicolon as the list separator: 1,2;2,3;3,4 Using commas for both decimals and list separation in comma-decimal mode will not parse as intended. Sample-size restrictions Several functions require minimum sample sizes: Function Requirement, SD n ≥ 2, VAR n ≥ 2, CV n ≥ 2, mean not zero, Tn≥2 TTEST n ≥ 2, T2 n1 ≥ 2, n2 ≥ 2, r at least 2 paired values LINREG at least 2 paired values, R2 at least 2 paired values Positive scale restrictions Several functions require positive scale parameters: Function Requirement, Φσ>0, Zσ>0, Ts>0, TTEST s > 0, T2 s1 > 0, s2 > 0 ZTEST σ > 0, t-CDF df > 0, CHI2 df > 0, Probability restrictions Function Requirement, Φ⁻¹ 0 < p < 1, BINOM 0 ≤ p ≤ 1, CI-p 0 ≤ p̂ ≤ 1 Confidence fields 0 < Confidence < 1, Count restrictions, Function Requirement BINOM n ≥ 0, 0 ≤ k ≤ n, POIS k ≥ 0, λ ≥ 0, ZTEST n > 0, CI-p n > 0 Approximation methods Several probability functions use approximations: Normal CDF uses an error-function approximation. Normal inverse uses a rational approximation. t-CDF uses a regularized incomplete beta implementation. Chi-square CDF uses a Wilson-Hilferty normal approximation. Binomial and Poisson probabilities use log-gamma calculations. These are numerical utilities, not symbolic statistics proofs. No raw p-values for all tests TTEST, T2, and ZTEST compute test statistics. They do not automatically compute p-values. To compute a p-value, use PVAL with the appropriate mode. Confidence interval critical values CI-μ uses a normal critical value, not a t critical value. This is important for small-sample interpretation. Regression limitations LINREG is simple linear regression only. It does not provide: residuals, standard errors confidence intervals for slope/intercept multiple regression, nonlinear regression If all X values are identical, regression fails because the slope denominator is zero. Correlation limitations Correlation requires nonzero variance in both X and Y. If one list is constant, r is invalid.

Practical usage notes

Use x̄, MED, MODE, MINMAX, RNG, and Σx for basic one-variable summaries. Use SD, VAR, and CV when measuring spread. Remember that SD and VAR use sample formulas with denominator n - 1. Use Q1–Q3, IQR, P, and 5NUM when the position of values inside a sorted distribution matters. Use Φ, Φ⁻¹, and Z for normal-distribution work. Use T, TTEST, and t-CDF for t-statistic workflows. Use PVAL after computing or entering a statistic if a p-value is needed. Use BINOM and POIS for probability mass calculations, not cumulative probabilities. Use r, LINREG, and R2 only when X and Y lists are paired in the same order. When entering lists, check the active decimal separator. Most input errors in this docklet will come from using the wrong list separator for the selected locale.

Appendix G.12 — Trigonometry

Overview

The Trigonometry docklet provides direct trigonometric, inverse trigonometric, hyperbolic, and inverse hyperbolic functions for the current calculator operand. Unlike prompt-based docklets, the Trigonometry docklet does not ask the user to fill in fields. Each function acts immediately on the value currently shown in the calculator. The result is returned to the calculator as a scalar numeric result and can continue participating in normal calculator expressions. The docklet is intended for users who need fast access to trigonometric and hyperbolic functions without leaving the main calculator workflow. The Trigonometry docklet contains 24 exposed functions: 6 standard trigonometric functions 6 inverse trigonometric functions 6 hyperbolic functions, 6 inverse hyperbolic functions All successful results are tagged as approximate results because the docklet uses floating-point mathematical functions internally.

Opening and using the docklet

The Trigonometry docklet appears as a parked or expandable panel attached to the calculator interface. When parked, the header shows: Trigonometry When expanded, the header shows the current angle unit label: deg rad grad rev The user opens the docklet, selects a function, and the function is applied immediately to the current calculator operand. There is no Compute button and no prompt dialog. Selecting a row is the computation. After a function is selected:

1. The current operand is read.
2. The selected function is applied.
3. The result is rounded according to the decimal places setting.
4. The calculator display is updated.
5. The result is committed back into the calculator expression flow.
6. The ticker is updated with the function label and result tag.
7. The docklet parks itself again if it was open.
8. The selected function symbol is visually marked as the last used function.

The docklet auto-parks after approximately two minutes of inactivity.

Current operand behavior

The Trigonometry docklet always uses the current calculator operand as its input. There are no manual input fields. For example, if the calculator display currently shows: 30 and the active angle unit is degrees, tapping: sin computes: sin(30 deg) If the calculator display currently shows: 2 and the user taps: cosh the docklet computes: cosh(2) The current operand must be convertible to a finite Double. If it is not finite, the docklet returns an invalid input state. The docklet uses the current operand differently depending on the function group: Function type How the operand is interpreted Standard trig functions Operand is interpreted as an angle in the active angle unit Inverse trig functions Operand is interpreted as a raw ratio/input value Hyperbolic functions Operand is interpreted as a raw numeric value Inverse hyperbolic functions Operand is interpreted as a raw numeric value

Prompt dialogs

The Trigonometry docklet does not use prompt dialogs. There are therefore no: required fields, optional fields, blank-field behaviors, mode fields choice menus, Compute button actions, Cancel button actions outside-tap warning states, list-entry fields Every function is immediate. The current calculator operand is the only input.

Decimal places and rounding

The Trigonometry docklet uses the app’s decimalPlaces setting. After a result is computed:

1. The raw floating-point result is converted to Decimal.
2. The decimal result is rounded using the active decimal places setting.
3. A displayed value of -0 is normalized to 0.
4. The rounded value becomes the new current operand.
5. The rounded value is committed back into the calculator expression system.

The docklet does not use the unused helper snapCommon in its active computation path. Near-zero snapping is only applied when the rounded decimal string would otherwise become -0.

Example: sin(π rad) may produce a floating-point value extremely close to zero. After rounding and normalization, the display may show: 0, Angle unit behavior The Trigonometry docklet uses the app’s global angle unit setting: angleUnit The supported angle units are: Stored angle unit Header / ticker label, Degrees deg, Radians rad, Gradians grad Turns rev, Standard trig functions The following functions treat the current operand as an angle: sin, cos, tan, csc, sec, cot Before computation, the operand is converted internally to radians. Active angle unit Conversion to radians Degrees x × π / 180 Radians x, Gradians x × π / 200, Turns x × 2π

Example: sin(90 deg) is internally computed as: sin(π/2) Inverse trig functions The following functions return an angle: arcsin, arccos, arctan, arccsc, arcsec, arccot They compute the inverse function internally in radians, then convert the result back to the active angle unit. For example, with degrees active: arcsin(1) returns: 90 deg With radians active, the same computation returns: 1.57079633 rad depending on the decimal places setting. Hyperbolic functions The hyperbolic and inverse hyperbolic functions do not use the active angle unit. They use the current operand as a raw numeric input. Examples: sinh(2), cosh(2), artanh(0.5) No angle conversion is applied to these functions.

Result behavior

Successful Trigonometry results are scalar numeric results. A successful result: updates the calculator display updates the current operand clears the expression entry calls commitUnaryResult can continue in the calculator expression flow updates the ticker receives an [APPROX] tag The ticker label is built from the function name and argument.

Example: sin(30 deg) [APPROX] For inverse trigonometric functions, the displayed result includes the active angle unit label.

Example: 30 deg The committed numeric value is still the numeric result. The unit label is display text, not a separate unit object.

Result tags

The Trigonometry docklet defines three result tags: [NUM] [APPROX] [LABEL] In this docklet, successful computed results use: [APPROX] Error and warning states use: [LABEL] Although [NUM] is defined in the docklet, it is not used for successful Trigonometry function results in the supplied code. [APPROX] [APPROX] means the result is numerical and approximate. This is appropriate because trigonometric and hyperbolic functions are evaluated through floating-point functions such as: sin cos tan sinh cosh tanh asin acos atan asinh acosh atanh The result may be very accurate for ordinary use, but it should not be interpreted as an exact symbolic value. [LABEL] [LABEL] is used for messages that are not normal numeric results. In this docklet, [LABEL] appears for errors such as: Division by 0 Out of range Invalid input, Invalid result, Invalid function When an error occurs, the display shows the error message and the ticker records the function that produced it.

Example: csc(0 rad) = Division by 0 [LABEL]

Function groups

The docklet groups functions visually in sets of three rows, with a separator after every third function. The logical function groups are: Standard trigonometric functions sin cos tan csc sec cot Inverse trigonometric functions $\sin^{-1}$ $\cos^{-1}$ $\tan^{-1}$ $\csc^{-1}$ $\sec^{-1}$ $\cot^{-1}$ Hyperbolic functions sinh cosh tanh csch sech coth Inverse hyperbolic functions $\sinh^{-1}$ $\cosh^{-1}$ $\tanh^{-1}$ csch^{-1} sech^{-1} $\coth^{-1}$

Function reference

sin — sin(x)

Purpose: Computes the sine of the current operand.

Inputs: Current calculator operand.

Parameters: x is interpreted as an angle in the active angle unit.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: sin(x) where x is converted to radians internally.

Notes / restrictions: No special domain restriction. Any finite numeric angle is accepted.

Example: Input: 90, Angle unit: deg, Function: sin, Result: 1 Ticker: sin(90 deg) [APPROX]

cos — cos(x)

Purpose: Computes the cosine of the current operand.

Inputs: Current calculator operand.

Parameters: x is interpreted as an angle in the active angle unit.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: cos(x) where x is converted to radians internally.

Notes / restrictions: No special domain restriction. Any finite numeric angle is accepted.

Example: Input: 180, Angle unit: deg, Function: cos, Result: -1 Ticker: cos(180 deg) [APPROX]

tan — tan(x)

Purpose: Computes the tangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is interpreted as an angle in the active angle unit.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: tan(x) where x is converted to radians internally.

Notes / restrictions: The code does not explicitly block tangent at vertical asymptotes. If the floating-point result is finite, it is returned. If the result is not finite, the docklet reports Invalid result.

Example: Input: 45, Angle unit: deg, Function: tan, Result: 1 Ticker: tan(45 deg) [APPROX]

csc — csc(x)

Purpose: Computes the cosecant of the current operand.

Inputs: Current calculator operand.

Parameters: x is interpreted as an angle in the active angle unit.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: csc(x) = 1 / sin(x)

Notes / restrictions: If sin(x) is effectively zero, the docklet reports: Division by 0 The zero threshold is: abs(value) < 1e-14

Example: Input: 90, Angle unit: deg, Function: csc, Result: 1 Ticker: csc(90 deg) [APPROX] Error example: Input: 0, Angle unit: rad, Function: csc Display: Division by 0 Ticker: csc(0 rad) = Division by 0 [LABEL]

sec — sec(x)

Purpose: Computes the secant of the current operand.

Inputs: Current calculator operand.

Parameters: x is interpreted as an angle in the active angle unit.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: sec(x) = 1 / cos(x)

Notes / restrictions: If cos(x) is effectively zero, the docklet reports: Division by 0 The zero threshold is: abs(value) < 1e-14

Example: Input: 60, Angle unit: deg, Function: sec, Result: 2 Ticker: sec(60 deg) [APPROX]

cot — cot(x)

Purpose: Computes the cotangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is interpreted as an angle in the active angle unit.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: cot(x) = 1 / tan(x)

Notes / restrictions: If tan(x) is effectively zero, the docklet reports: Division by 0 The zero threshold is: abs(value) < 1e-14

Example: Input: 45, Angle unit: deg, Function: cot, Result: 1 Ticker: cot(45 deg) [APPROX]

$\sin^{-1}$ — arcsin(x)

Purpose: Computes the inverse sine of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric ratio, not an angle.

Result: Angle result in the active angle unit.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX]. The display includes the active angle unit label.

Formula: arcsin(x)

Notes / restrictions: Input must satisfy: $-1 \le x \le 1$ Otherwise, the docklet reports: Out of range (−1…1)

Example: Input: 1, Angle unit: deg, Function: $\sin^{-1}$, Result: 90 deg Ticker: arcsin(1) [APPROX]

$\cos^{-1}$ — arccos(x)

Purpose: Computes the inverse cosine of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric ratio, not an angle.

Result: Angle result in the active angle unit.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX]. The display includes the active angle unit label.

Formula: arccos(x)

Notes / restrictions: Input must satisfy: $-1 \le x \le 1$ Otherwise, the docklet reports: Out of range (−1…1)

Example: Input: 0, Angle unit: deg, Function: $\cos^{-1}$, Result: 90 deg Ticker: arccos(0) [APPROX]

$\tan^{-1}$ — arctan(x)

Purpose: Computes the inverse tangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value, not an angle.

Result: Angle result in the active angle unit.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX]. The display includes the active angle unit label.

Formula: arctan(x)

Notes / restrictions: Any finite numeric input is accepted.

Example: Input: 1, Angle unit: deg, Function: tan⁻¹, Result: 45 deg Ticker: arctan(1) [APPROX]

csc⁻¹ — arccsc(x)

Purpose: Computes the inverse cosecant of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Angle result in the active angle unit.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX]. The display includes the active angle unit label.

Formula: arccsc(x) = arcsin(1 / x)

Notes / restrictions: Input must satisfy: |x| ≥ 1 Input must also not be effectively zero. Possible errors: Division by 0 Out of range (|x| ≥ 1)

Example: Input: 2, Angle unit: deg, Function: csc⁻¹, Result: 30 deg Ticker: arccsc(2) [APPROX]

sec⁻¹ — arcsec(x)

Purpose: Computes the inverse secant of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Angle result in the active angle unit.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX]. The display includes the active angle unit label.

Formula: arcsec(x) = arccos(1 / x)

Notes / restrictions: Input must satisfy: |x| ≥ 1 Input must also not be effectively zero. Possible errors: Division by 0 Out of range (|x| ≥ 1)

Example: Input: 2, Angle unit: deg, Function: sec⁻¹, Result: 60 deg Ticker: arcsec(2) [APPROX]

cot⁻¹ — arccot(x)

Purpose: Computes the inverse cotangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Angle result in the active angle unit.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX]. The display includes the active angle unit label.

Formula: arccot(x) = arctan(1 / x)

Notes / restrictions: The code rejects inputs that are effectively zero. Possible error: Division by 0 Important implementation note: mathematically, some definitions of arccot allow x = 0. This implementation does not. In this docklet, arccot(0) returns a division-by-zero warning.

Example: Input: 1, Angle unit: deg, Function: cot⁻¹, Result: 45 deg Ticker: arccot(1) [APPROX]

sinh — sinh(x)

Purpose: Computes the hyperbolic sine of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: sinh(x)

Notes / restrictions: No angle unit conversion is applied. The active angle unit is irrelevant.

Example: Input: 2, Function: sinh, Result: 3.62686041, Ticker: sinh(2) [APPROX]

cosh — cosh(x)

Purpose: Computes the hyperbolic cosine of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: cosh(x)

Notes / restrictions: No angle unit conversion is applied. The active angle unit is irrelevant.

Example: Input: 2, Function: cosh, Result: 3.76219569, Ticker: cosh(2) [APPROX]

tanh — tanh(x)

Purpose: Computes the hyperbolic tangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.
Result behavior: Returns to the calculator expression flow and is tagged [APPROX].
Formula: tanh(x)
Notes / restrictions: No angle unit conversion is applied. The active angle unit is irrelevant.
Example: Input: 2, Function: tanh, Result: 0.96402758, Ticker: tanh(2) [APPROX]

csch — csch(x)

Purpose: Computes the hyperbolic cosecant of the current operand.
Inputs: Current calculator operand.
Parameters: x is a raw numeric value.
Result: Scalar numeric result.
Result behavior: Returns to the calculator expression flow and is tagged [APPROX].
Formula: csch(x) = 1 / sinh(x)
Notes / restrictions: If sinh(x) is effectively zero, the docklet reports: Division by 0 The zero threshold is: abs(value) < 1e-14
Example: Input: 2, Function: csch, Result: 0.27572056, Ticker: csch(2) [APPROX] Error example: Input: 0, Function: csch Display: Division by 0 Ticker: csch(0) = Division by 0 [LABEL]

sech — sech(x)

Purpose: Computes the hyperbolic secant of the current operand.
Inputs: Current calculator operand.
Parameters: x is a raw numeric value.
Result: Scalar numeric result.
Result behavior: Returns to the calculator expression flow and is tagged [APPROX].
Formula: sech(x) = 1 / cosh(x)
Notes / restrictions: If cosh(x) is effectively zero, the docklet reports: Division by 0 For ordinary real inputs, cosh(x) is not zero, so this error is unlikely unless a non-finite or numerically invalid condition occurs.
Example: Input: 2, Function: sech, Result: 0.26580223, Ticker: sech(2) [APPROX]

coth — coth(x)

Purpose: Computes the hyperbolic cotangent of the current operand.
Inputs: Current calculator operand.
Parameters: x is a raw numeric value.
Result: Scalar numeric result.
Result behavior: Returns to the calculator expression flow and is tagged [APPROX].
Formula: coth(x) = 1 / tanh(x)
Notes / restrictions: If tanh(x) is effectively zero, the docklet reports: Division by 0 The zero threshold is: abs(value) < 1e-14
Example: Input: 2, Function: coth, Result: 1.03731472, Ticker: coth(2) [APPROX]

$\sinh^{-1}$ — arsinh(x)

Purpose: Computes the inverse hyperbolic sine of the current operand.
Inputs: Current calculator operand.
Parameters: x is a raw numeric value.
Result: Scalar numeric result.
Result behavior: Returns to the calculator expression flow and is tagged [APPROX].
Formula: arsinh(x)
Notes / restrictions: Any finite numeric input is accepted.
Example: Input: 2, Function: $\sinh^{-1}$, Result: 1.44363548, Ticker: arsinh(2) [APPROX]

$\cosh^{-1}$ — arcosh(x)

Purpose: Computes the inverse hyperbolic cosine of the current operand.
Inputs: Current calculator operand.
Parameters: x is a raw numeric value.
Result: Scalar numeric result.
Result behavior: Returns to the calculator expression flow and is tagged [APPROX].
Formula: arcosh(x)
Notes / restrictions: Input must satisfy: $x \geq 1$ Otherwise, the docklet reports: Out of range (≥ 1)
Example: Input: 2, Function: $\cosh^{-1}$, Result: 1.3169579, Ticker: arcosh(2) [APPROX]

$\tanh^{-1}$ — artanh(x)

Purpose: Computes the inverse hyperbolic tangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: artanh(x)

Notes / restrictions: Input must satisfy: -1 < x < 1 The endpoints are excluded. Otherwise, the docklet reports: Out of range (−1…1 excl.)

Example: Input: 0.5, Function: tanh⁻¹, Result: 0.54930614, Ticker: artanh(0.5) [APPROX]

csch⁻¹ — arcsch(x)

Purpose: Computes the inverse hyperbolic cosecant of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: arcsch(x) = arsinh(1 / x)

Notes / restrictions: Input must not be effectively zero. Possible error: Division by 0 The zero threshold is: abs(value) < 1e-14

Example: Input: 2, Function: csch⁻¹, Result: 0.48121183, Ticker: arcsch(2) [APPROX]

sech⁻¹ — arsech(x)

Purpose: Computes the inverse hyperbolic secant of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: arsech(x) = arcosh(1 / x)

Notes / restrictions: Input must satisfy: 0<x≤1 Otherwise, the docklet reports: Out of range (0 < x ≤ 1)

Example: Input: 0.5, Function: sech⁻¹, Result: 1.3169579, Ticker: arsech(0.5) [APPROX]

coth⁻¹ — arcoth(x)

Purpose: Computes the inverse hyperbolic cotangent of the current operand.

Inputs: Current calculator operand.

Parameters: x is a raw numeric value.

Result: Scalar numeric result.

Result behavior: Returns to the calculator expression flow and is tagged [APPROX].

Formula: arcoth(x) = artanh(1 / x)

Notes / restrictions: Input must satisfy: |x| > 1 Input must also not be effectively zero. Possible errors: Division by 0 Out of range (|x| > 1)

Example: Input: 2, Function: coth⁻¹, Result: 0.54930614, Ticker: arcoth(2) [APPROX]

Errors and limitations

Invalid input If the current operand cannot be converted to a finite Double, the docklet reports: Invalid input The ticker records the attempted function and the error.

Example: sin(?) = Invalid input [LABEL] Invalid result If a computation produces a non-finite value, the docklet reports: Invalid result This can occur if a floating-point function overflows or returns a non-finite value. Division by zero The reciprocal functions check whether their denominator is effectively zero. The threshold is: abs(value) < 1e-14 This applies to: csc(x) = 1 / sin(x) sec(x) = 1 / cos(x) cot(x) = 1 / tan(x) arccsc(x) = arcsin(1 / x) arcsec(x) = arccos(1 / x) arccot(x) = arctan(1 / x) csch(x) = 1 / sinh(x) sech(x) = 1 / cosh(x) coth(x) = 1 / tanh(x) arcsch(x) = arsinh(1 / x) arcoth(x) = artanh(1 / x) Out-of-range inverse functions Several inverse functions enforce explicit domains: Function Required input, arcsin(x) -1 ≤ x ≤ 1, arccos(x) -1 ≤ x ≤ 1 arccsc(x) |x| ≥ 1, arcsec(x) |x| ≥ 1, arcosh(x) x ≥ 1, artanh(x) -1 < x < 1 arsech(x) 0 < x ≤ 1, arcoth(x) |x| > 1 Approximation behavior All successful Trigonometry results are tagged [APPROX]. Reasons: the docklet converts Decimal input to Double calculations are performed using floating-point math trigonometric and hyperbolic functions are numerical values near singularities may be sensitive values near zero may be affected by finite precision Tangent near asymptotes The code does not explicitly block tan(x) at asymptotes. It computes the tangent and returns the value if finite. For angles such as: 90 deg, π/2 rad, 100 grad 0.25 rev the result may be a very large finite floating-point number rather than a division-by-zero error. The reciprocal functions perform explicit near-zero checks, but tan(x) itself does not. Arccot convention The implementation computes: arccot(x) = arctan(1 / x) and rejects x = 0 as division by zero. Some mathematical conventions define arccot differently. This appendix describes the implementation used by CCalc. Hyperbolic angle independence Hyperbolic functions do not use the active angle unit. They always use the current operand as a raw number. For example: sinh(30) does not mean sinh(30 deg). It means the hyperbolic sine of the number 30. No symbolic exactness The docklet does not return symbolic exact values. For example, it does not return: √2 / 2 π/6 It returns rounded numeric approximations. No prompt

correction Because this docklet has no prompt dialog, there is no opportunity to correct fields before computing. The user changes the calculator operand first, then taps the desired function.

Practical usage notes

Use the standard trigonometric functions when the current operand represents an angle. Use the inverse trigonometric functions when the current operand represents a ratio or numeric value and the desired result is an angle. Use the hyperbolic functions when the current operand is a raw numerical argument, not an angle. Use reciprocal functions carefully near zero or near trigonometric singularities. The docklet checks reciprocal division using a near-zero threshold, but floating-point behavior can still produce very large values near asymptotes. Before using standard trigonometric functions, verify the active angle unit in the docklet header. The same displayed operand produces different results depending on whether the active angle unit is degrees, radians, gradians, or turns. Examples: sin(90 deg) = 1, sin(90 rad) ≈ 0.89399666, sin(90 grad) ≈ 0.98768834 sin(0.25 rev) = 1 For inverse trig results, the displayed output uses the active angle unit. This means: arcsin(1) may display as: 90 deg 1.57079633 rad 100 grad 0.25 rev depending on the active angle unit. How to Read Formula Entries Each function entry in Appendix G follows a consistent structure. A complete entry may include: Function, Symbol, Category, Purpose, Inputs Formula or method Output Output unit or meaning

Result behavior

Approximation notes Error conditions, Example Simple functions may use shorter entries. Complex functions, such as IRR, SVD, QR decomposition, regression, numerical integration, modular arithmetic, or matrix solving, should use more detailed entries.

Example: Function: Circle Area, Symbol: A◯, Category: Geometry Purpose: Computes the area of a circle from its radius. Inputs: r,

Formula: $A = \pi r^2$, Output: area Output unit or meaning: square length unit Result behavior: numeric scalar Approximation notes: Uses numerical π and the active decimal-place setting. Error conditions: r must be ≥ 0. Example: r = 10 → A = 314.15926536 This structure keeps the catalogue readable even when a docklet contains many functions. Input Requirements Input requirements describe what a function needs before it can compute. Inputs may come from several sources: prompt fields the current calculator operand selected modes, selected units, selected angle unit, selected base currency stored settings lists of values matrices or vectors expression templates, constants, graphing variables, previous results A function may require a single input, such as: sin(x) or several inputs, such as: PV, FV, rate, NPER, payment timing or a structured set of inputs, such as: A11, A12, A21, A22 or: x1, y1, x2, y2 or: cashflows: CF0, CF1, CF2, ... Some fields are mandatory. Other fields may have default behavior. When a prompt says: blank = operand it means that the field may be left empty, and CCalc will use the current calculator operand as that input.

Example: Current operand: 10 Prompt field: r/s, blank = operand Leaving the field empty means: r = 10 This behavior is intended to make docklets faster when the value has already been entered on the main calculator. Required fields must not be left empty. If a required field is empty, malformed, or outside the accepted range, CCalc reports an invalid input state or disables computation until the input is corrected. Parameter Meaning The same letter may mean different things in different docklets. For example: r may mean radius in Geometry r may mean rate in Financial r may mean correlation in Statistics n may mean number of periods, sample size, matrix size, integer input, or sequence index k may mean payment number, exponent, event count, group size, or iteration index A may mean area, matrix A, amplitude, amount, or coefficient For this reason, Appendix G defines parameters locally inside each function entry. A symbol is not enough by itself. The meaning depends on the selected docklet and the selected function. Input Types CCalc functions generally use the following input types. Numeric Scalar A single number. Examples: r = 10, rate = 1.5, x = 2, Integer A whole number, often used for counts, periods, dimensions, factorials, modular arithmetic, sequence indexes, or matrix size.

Examples: n = 12, k = 3, size = 2, Percentage A number interpreted as a percent.

Example: rate = 5 means: 5% not: 5.0 as a raw multiplier List A sequence of values. In dot-decimal mode: 1.2, 2.3, 3.4 In comma-decimal mode: 1, 2; 2, 3; 3, 4, Cash-Flow List An ordered financial list.

Example: -1000, 300, 400, 500 The order matters. The first value is usually the initial cash flow. Vector An ordered tuple.

Example: (3, 4, 5), Matrix A structured grid of values.

Example: [[1, 2], [3, 4]], Angle A number interpreted according to the active angle unit. The active angle unit may be: degrees, radians, gradians, turns, Mode A selector or small integer that changes the function behavior.

Example: 0 = add 1 = subtract or: 0 = deg → rad 1 = rad → deg Output Units Output units describe what the returned value represents. Some CCalc functions are dimensionless. Others return a value with a mathematical, financial, statistical, or physical meaning. Examples: Trigonometric ratios are usually dimensionless. Inverse trigonometric functions return an angle. Circle area returns square length units. Sphere volume returns cubic length units. Financial functions may return money, percent, periods, rates, or ratios. Statistics functions may return a statistic, probability, interval, count, or regression coefficient. Linear algebra functions may return scalars, vectors, matrices, decompositions, or labels. Number theory functions usually return integers or structured integer results. Engineering formulas may return volts, ohms, watts, hertz, seconds, decibels, ratios, or other applied units. Because CCalc usually receives plain numeric values, the user is responsible for keeping input units consistent.

Example: If a radius is entered in centimeters, circle area is returned in square centimeters. If a radius is entered in meters, circle area is returned in square meters. CCalc may label dimensional families in some docklets, such as: [L] [L²] [L³] These labels mean: • [L] = length-like output • [L²] = area-like output • [L³] = volume-like output They do not force a specific physical unit. They describe dimensional meaning. Unit Consistency Many formulas assume consistent units.

Example: $A = \pi r^2$ If r is in meters, A is in square meters.

Example: $V = lwh$ If l, w, and h are in centimeters, V is in cubic centimeters.

Example: P = VI If V is in volts and I is in amperes, P is in watts.

Example: τ = RC If R is in ohms and C is in farads, τ is in seconds. CCalc does not always infer units from the number alone. The user must know what the number represents. The Converter is the main tool for explicit unit conversion. Docklet formulas usually assume the user has entered values in compatible units. Approximation Rules Some functions produce exact-style numeric results. Others produce approximate results. Approximate results are normal and expected in many areas of CCalc. Approximation may occur because the function uses: π e trigonometric functions logarithms, exponentials, square roots floating-point evaluation iterative solving, numerical integration statistical distribution approximations matrix decompositions eigenvalue methods, financial root solving, graph sampling rounded display precision A result tagged: [APPROX] means the value should be understood as a numerical approximation. This does not mean the result is wrong. It means the result is not an exact symbolic expression. Examples: sin(2 rad) is approximate because it is evaluated numerically. Ellipse perimeter may be approximate because closed-form elementary perimeter calculation is not generally used. IRR is approximate because it is solved numerically. Eigenvalues may be approximate because numerical matrix methods are used. Approximate scalar results can usually continue into further arithmetic, but the user should remember that the result carries numerical precision limits. Exact, Approximate, and Label Results Appendix G identifies result type clearly. [NUM] A numeric result intended to behave as an ordinary scalar.

Example: A□ = s² [APPROX] A numeric result produced by a numerical or approximate method.

Example: sin(x) [LABEL] A structured, descriptive, symbolic, matrix-like, list-like, or diagnostic result that should not be treated as a single ordinary scalar.

Example: factor(n) = 2³ × 3 × 5 or: A⁻¹ = [[...]] A label result may still contain numbers, but the full output is not one plain operand. Rounding and Display Precision The active decimal-place setting affects many displayed and committed results. Example, with 8 decimal places: 1 ÷ 3 = 0.33333333 Example, with 4 decimal places: 1 ÷ 3 = 0.3333 Rounding affects how results appear and may also affect the value that continues through the calculator flow after a committed result. This is important when chaining calculations. For high-sensitivity work, users should choose a decimal-place setting appropriate to the task. Error Conditions Error conditions occur when a function cannot produce a valid result from the supplied inputs. Invalid Input The input cannot be parsed or is missing. Examples: Invalid input, Invalid data, Invalid matrix Division by Zero A denominator is zero or a reciprocal is undefined.

Examples: Division by 0, Price cannot be 0, Old cannot be 0 Out of Range The input is outside the mathematical domain. Examples: Out of range p must be 0<p<1 σ must be > 0 r must be ≥ 0 No Real Solution The requested real-valued result does not exist. Examples: No real solution, No real area Complex eigenvalues not supported Singular or Degenerate Case The structure collapses or cannot be inverted, solved, or normalized.

Examples: Singular matrix, Degenerate line, Zero vector, Cannot normalize Unsupported or Limited Case The function is outside the supported implementation range. Examples: Unsupported, n too large Complex eigenvalues not yet supported LU failed Overflow or Precision Limit The number is too large or the result exceeds the safe supported range. Examples: Overflow, Factorial too large, Invalid result Error conditions are protective. They prevent CCalc from returning a misleading value. Technical Examples The examples in Appendix G should use a consistent format. Numeric scalar example

Input: r = 10

Formula: A = πr²

Result: A = 314.15926536 Result type: [NUM] Approximate result example

Input: x = 2 rad

Formula: sin(x)

Result: sin(2 rad) ≈ 0.90929743 Result type: [APPROX] Label-style result example

Input: n = 60

Formula: prime factorization of n

Result: 60 = 2² × 3 × 5 Result type: [LABEL], Matrix-style result example

Input: A = [[1, 2], [3, 4]]

Formula: det(A) = ad − bc

Result: det(A) = -2 Result type: [APPROX] or [NUM], depending on the implementation path Invalid input example

Input: r = -5

Formula: A = πr²

Result: r must be ≥ 0 Result type: Warning / error Recommended Entry Template for Each Docklet Function Each function in the full catalogue may use the following template. Function: Symbol: Category:

Purpose: Inputs: Formula or method:

Output: Output unit or meaning:

Result behavior: Approximation notes: Error conditions:

Example: Simple functions can use short entries. More complex functions should use fuller entries. Detailed entries are especially useful for functions such as: IRR, regression, numerical integration, matrix inverse, determinant, eigenvalues QR decomposition SVD, modular arithmetic, complex logarithms roots of unity probability distributions Closing Note Appendix G should be used as a technical reference, not as a step-by-step tutorial. When using a docklet, the most important questions are:

1. What does this function compute?
2. What inputs does it require?
3. What does each parameter mean?

4. What units or interpretation are assumed?
5. Is the result exact, approximate, numeric, or label-only?
6. Can the result continue into ordinary arithmetic?
7. What input values are invalid?

The docklet-specific sections answer those questions one domain at a time. CCalc’s function catalogue is intentionally broad. It includes elementary formulas, applied formulas, financial formulas, statistical approximations, discrete mathematics, complex numbers, matrix operations, number theory, and specialized engineering workflows. The purpose of Appendix G is to make those tools explicit, predictable, and understandable for users who want to know exactly what CCalc is doing.

Appendix H - Technical Behavior Reference

Overview

This appendix describes the technical behavior that sits behind CCalc’s visible calculator interface.

Most users do not need to read this section. It is intended for users who want to understand why CCalc behaves the way it does when numbers are entered, results are returned, docklets are used, labels appear, warnings are shown, or formatting settings are changed.

CCalc is not only a basic calculator display. It maintains several related states at the same time:

- the visible display
- the current operand
- the expression entry
- the expression token flow
- the ticker
- ticker label overrides
- operand label overrides
- result-return behavior from docklets
- formatting and rounding settings
- locale-sensitive number presentation
- sound and warning feedback

These states work together so that ordinary calculations, docklet results, converter results, grapher values, constants, and labeled outputs can all participate in the same calculator environment.

Display State

The display is the main number or message shown in the calculator’s primary result area.

It usually shows the current numeric value, but it can also temporarily show:

- an error message
- a warning message
- a label-style result
- a formatted value
- a converted value
- a docklet return value
- a result with an angle suffix
- a result affected by notation settings

Examples:

- 12.5
- 0.33333333
- Division by 0
- Out of range
- Invalid input
- 1.57079633 rad

The display is user-facing. It is what the user sees immediately after entering a number, pressing an operator, computing a docklet function, receiving a converter result, or restoring a previous result.

The display should not be confused with the internal expression state. A value may appear in the display while the expression system stores a more structured representation behind it.

Current Operand

The current operand is the active numeric value held by the calculator.

It is the value most operations use as their immediate input.

Examples:

- pressing sin in the Trigonometry docklet applies sin to the current operand
- selecting Circle Area in Geometry may use the current operand when the prompt says blank = operand
- pressing a binary operator uses the current operand as part of the expression flow
- a converter result becomes the current operand after conversion
- a grapher y-value sent back to CCalc becomes the current operand

The current operand is numeric. It is normally stored as a Decimal value.

This matters because some outputs are not purely numeric. For example, a midpoint result may contain both x and y coordinates, and a matrix result may contain several values. In those cases, CCalc may display a label-style result while only one scalar value, or no continued scalar value, is suitable for calculator flow.

In general:

- numeric results can become the current operand
- scalar docklet results can continue into ordinary arithmetic
- label-only results are primarily informational
- structured results may not behave like ordinary operands

Expression Entry

The expression entry is the currently typed numeric text before it is fully committed into the expression.

It is useful to think of it as the active input buffer.

For example, while the user is typing:

123

the expression entry may contain the text being built before it is committed into the expression token stream.

When an operation is selected, CCalc may convert the current expression entry into a numeric token.

Expression entry is cleared when:

- a number is committed into the expression
- a docklet result is committed
- a converter result is received
- a calculation is completed
- a warning state interrupts input
- a label-style result replaces the normal numeric entry path

This prevents stale typed digits from accidentally mixing with a result returned by another part of the app.

Ticker Label Override

The ticker label override is a special label used when the ticker needs to show something more specific than the ordinary operation text.

The ticker normally shows the current expression, operation, formula, or result context. The override allows CCalc to replace or enrich that ticker text.

Typical uses include:

- docklet formula labels
- result tags such as [NUM], [APPROX], and [LABEL]
- converter labels
- multi-part results
- warning messages
- formula-style summaries
- localized ticker strings
- descriptive outputs such as confidence intervals, matrices, or factorization strings

Examples:

sin(2 rad) [APPROX]
A◯[L²](r:10) → 314.15926536 [NUM]
MIN=1, MAX=9 [LABEL]
Division by 0 [LABEL]

The ticker label override is important because the ticker is not always a plain expression. Sometimes it must carry a technical explanation of what was computed.

Operand Label Override

The operand label override is used when the main operand area needs to show a label-style value instead of a normal numeric value.

This is different from the ticker label override.

The ticker label override affects the ticker.

The operand label override affects the operand display.

This is useful for results that are not naturally represented by a single scalar number.

Examples include:

- vectors
- matrices
- intervals
- factorization strings
- multiple roots
- midpoint coordinates
- confidence intervals
- multi-value summaries
- label-only diagnostic messages

For example, a matrix result may be meaningful as:

```
[[1; 2]; [3; 4]]
```

but that is not a single Decimal operand in the ordinary calculator sense.

Operand label override allows CCalc to show the result clearly without pretending that the whole structure is a normal scalar value.

Docklet Return Behavior

Docklets return results in different ways depending on the type of result.

A docklet result may be:

- a normal numeric scalar
- an approximate numeric scalar
- a label-only result
- a mixed result with a scalar plus explanatory text
- an error or warning state

CCalc distinguishes these result types so that the calculator knows whether the result should continue into arithmetic.

The usual return behaviors are:

Flow Scalar

The result is a usable numeric value. It can continue into further arithmetic.

Examples:

```
Circle Area
```

Mean

Determinant

```
sin(x)
```

Terminal Label

The result is mainly descriptive or structured. It should be read as output, not treated as an ordinary arithmetic operand.

Examples:

```
Matrix inverse
```

RREF

Linear regression equation

Factorization

Divisor list

Midpoint coordinates

Approximate Scalar

The result is numeric and can continue, but it should be understood as approximate because it came from floating-point math, an iterative method, a distribution approximation, or a transcendental function.

Examples:

sin(2 rad)

IRR

normal inverse

ellipse perimeter

eigenvalue approximation

Warning or Error

The result should not continue as a normal operand.

Examples:

Division by 0
Invalid input
Out of range
No real solution
Singular matrix

Result Committing Behavior

Result committing is the process by which a computed value is sent back into the calculator.

A committed numeric result normally updates:

- current operand
- display
- expression flow
- ticker
- ticker label override
- result state
- expression entry

The typical numeric commit sequence is:

1. Compute the raw result.
2. Round the result according to the active decimal-place setting.
3. Normalize values such as negative zero.
4. Store the result as the current operand.
5. Update the display.
6. Clear the expression entry.
7. Push the value into the calculator’s expression flow.
8. Update the ticker or ticker label override.
9. Mark the docklet function as computed, where applicable.

This keeps returned values usable. For example, after computing a Geometry area or a Statistics mean, the result can become the starting point for another operation.

Label-only commits are different. They may update the display and ticker without turning the entire output into a normal arithmetic operand.

Auto-Dismiss Behavior

Many CCalc docklets use auto-dismiss behavior.

This means that when a docklet remains open and inactive for a period of time, it parks itself automatically.

The purpose is to keep the interface clear while still leaving the docklet available.

Typical behavior:

- when a docklet is opened, an auto-dismiss timer starts
- interacting with the docklet resets the timer
- scrolling, tapping, dragging, or opening a prompt usually resets the timer
- when the timer expires, the docklet parks
- if a prompt is open, the docklet does not auto-dismiss
- if the docklet is already parked, no further action is needed

Auto-dismiss does not mean the docklet is closed. It means the docklet is minimized into its parked position.

The user can reopen it by tapping or dragging the docklet header.

Formatting Pipeline

The formatting pipeline controls how a numeric value becomes readable text.

A value may pass through several stages before it appears on screen:

1. Internal numeric value
2. Rounded Decimal value
3. Negative-zero normalization
4. Locale-sensitive decimal separator
5. Thousands separator setting
6. Trailing-zero setting
7. Notation mode
8. Display string
9. Ticker string

The internal value and the displayed string are not always identical in form.

For example, the internal value may be:

1234.5

but the displayed value may appear as:

1234.5

or:

1,234.5

or:

1234,5

depending on settings.

Formatting affects presentation. It should not be understood as changing the mathematical meaning of the value unless the value is explicitly rounded and stored after a result commit.

Rounding Pipeline

CCalc uses the active decimal-place setting to round many computed results before display and continued use.

The usual rounding pipeline is:

1. A function computes a raw result.
2. The result is converted into Decimal form where appropriate.
3. The decimal-place setting is clamped to a supported range.
4. The value is rounded using the selected rounding behavior.
5. Negative zero is normalized to zero.
6. The rounded value is stored or displayed.

Example:

```
1 ÷ 3
```

with 8 decimal places may display:

0.33333333

The rounded display is the user-facing result. Depending on the operation path, the rounded value may also become the committed operand used for continued calculation.

This is intentional. It keeps the calculator predictable and consistent with the precision selected by the user.

Locale-Sensitive Formatting

CCalc supports locale-sensitive formatting for decimal separators and list separators.

This matters especially in prompt-based docklets.

In dot-decimal mode, a data list may be written like this:

1.2, 2.3, 3.4

Here, the decimal separator is a dot and the list separator is a comma.

In comma-decimal mode, the same list may be written like this:

1,2; 2,3; 3,4

Here, the decimal separator is a comma and the list separator is a semicolon.

This avoids ambiguity.

For example:

1,2,3,4

could mean several things in comma-decimal mode, so CCalc uses semicolons to separate list items when commas are used as decimal separators.

Locale-sensitive behavior may affect:

- decimal display
- prompt placeholders
- typed numeric input
- list parsing
- ticker text
- formatted results
- statistics data entry
- cash-flow entry
- converter values

The mathematical value is preserved, but the visible string adapts to the selected formatting rules.

Audio Feedback Behavior

CCalc uses audio feedback to make the interface feel responsive and to help distinguish action types.

Different sounds may be used for:

- key presses
- menu selections
- docklet parking
- docklet reopening
- warnings
- confirmation actions
- settings changes

Typical examples:

- pressing a calculator key plays a click sound
- selecting a docklet function plays a click or menu sound
- parking a docklet plays a park sound
- reopening a docklet may play a different sound
- invalid input plays a warning sound

Audio feedback can usually be adjusted or disabled through the Settings menu.

When key click sound is disabled or muted, visual behavior remains the same. Only the sound feedback changes.

Warning Feedback Behavior

Warning feedback appears when CCalc detects a condition that should not proceed as an ordinary calculation.

Warning feedback may include:

- warning sound
- red prompt text
- ticker warning
- display warning
- disabled Compute button
- unchanged result state
- label-style ticker tag
- haptic feedback

Examples of warning states include:

Invalid input
Division by 0

Out of range

No real solution

Singular matrix

n must be > 0

σ must be > 0

Warnings are protective. They prevent CCalc from committing a misleading result or continuing with a value that does not satisfy the selected function’s rules.

Invalid Input States

An invalid input state occurs when the entered value cannot be parsed or does not match the selected function’s requirements.

Common causes include:

- empty required fields
- non-numeric text
- malformed lists
- missing cash-flow values
- invalid matrix size
- invalid mode value
- unsupported negative value
- denominator equal to zero
- invalid probability
- invalid sample size
- invalid angle range
- invalid modulus

Examples:

p must be 0..1

n must be ≥ 2

Size must be 2 or 3

Cashflows CSV is invalid

Invalid input states usually do not replace the current operand with a new valid result. The user must correct the input and compute again.

Division by Zero

Division by zero is treated as a warning state.

It can occur in ordinary arithmetic or inside docklet formulas.

Examples:

- ordinary division by 0
- slope of a vertical line
- reciprocal trig functions at invalid points
- modular inverse when no inverse exists
- financial formulas with invalid denominators
- statistical formulas with invalid variance or sample size
- matrix inverse of a singular matrix

CCalc does not silently continue through division by zero. It reports the issue and prevents the result from being treated as an ordinary valid operand.

Out-of-Range Results

Out-of-range results occur when a function is mathematically undefined for the supplied input.

Examples:

arcsin(2)

is invalid because arcsin expects an input from −1 to 1.

sqrt(-5)

is invalid in ordinary real-number mode.

normal inverse p = 1

is invalid because the inverse normal calculation expects 0 < p < 1.

arcosh(0.5)

is invalid because arcosh expects x ≥ 1.

When a result is out of range, CCalc reports the condition rather than returning a misleading numeric value.

Approximate Results

Some results are tagged or understood as approximate.

Approximation can come from:

- floating-point transcendental functions
- iterative methods
- statistical distribution approximations
- numerical root solving
- eigenvalue approximation
- graph sampling
- geometric approximations
- financial rate solving
- irrational constants such as π

Approximate does not mean unreliable. It means the result is numerical rather than exact symbolic output.

Examples:

sin(2 rad) [APPROX]

IRR% [APPROX]

P□[L](a:5, b:3) [APPROX]

Approximate scalar results may still be used in further calculations.

Numeric Results

Numeric results are ordinary scalar values.

They can generally become the current operand and continue into calculator flow.

Examples:

- 12
- 3.14159265
- 0.25
- 144
- 1.61803399

A result tagged [NUM] indicates that the value is being treated as a normal numeric result.

Numeric results are the safest type for continued arithmetic.

Label-Only Results

Label-only results are outputs whose meaning is not fully captured by one scalar number.

Examples:

- a factorization
- a list of divisors
- a matrix
- a vector
- a regression equation
- a confidence interval
- a midpoint coordinate pair
- a five-number summary

A label-only result may still display a number somewhere, but the whole result should be read as a structured output.

For example:

MIN=1, Q1=2, MED=3, Q3=4, MAX=5

is not one number. It is a summary.

A label-only result is usually not intended to be chained into ordinary arithmetic as if it were a single value.

Display Labels

Display labels are labels shown in or near the main display area to clarify what the value represents.

They help prevent ambiguity when a number alone is insufficient.

For example, a result may be numerically equal to 10, but the meaning could be:

- radius
- area
- rate
- period count
- determinant
- mean
- slope
- tax amount
- converted currency value

Display labels help communicate that context.

Ticker Labels

Ticker labels explain the operation that produced the result.

They may show:

- the function name
- the input values
- the selected mode
- the unit context
- the result tag
- an error message
- a structured result summary

Examples:

A◯[L²](r:10) → 314.15926536 [NUM]
Z(x;μ,σ) [NUM]
SVD(A_2) = U=... Σ=... V^T=... [LABEL]

The ticker is a running explanation layer. It helps the user understand where the visible result came from.

Formatting and Rounding Behavior

Formatting and rounding are related but not identical.

Rounding changes the numeric precision of a result.

Formatting changes how the number is shown.

Examples of rounding:

0.333333333333

rounded to 8 decimal places becomes:

0.33333333

Examples of formatting:

1234.5

may appear as:

1,234.5

or:

1.234,5

depending on separator settings.

CCalc applies formatting consistently across the display, ticker, docklets, converter results, and graph-related returned values where supported.

The guiding rule is:

The internal value must remain usable, while the displayed value must remain readable.

Practical Interpretation Rule

When reading any CCalc result, read the number together with its surrounding context.

A result is fully understood only when the user considers:

- the display value
- the ticker label
- the result tag
- the selected settings
- the docklet or tool that produced it
- whether the result is numeric, approximate, or label-only
- whether the value is intended to continue into arithmetic

This is especially important for advanced docklets, converter results, graphing values, and structured mathematical outputs.

CCalc is designed so that the visible interface remains simple, while the technical state behind it preserves enough context to keep advanced calculations understandable and predictable.