SpeedCrunch version 0.10 math

Philosophy of Input

When you first launch SpeedCrunch you might be tempted to use it like an ordinary pocket calculator, that is punch with your mouse on the keypad to produce results.

But SpeedCrunch has been developed with fast keyboard usage in mind. Of course, at your option, you may stick to mouse and graphical elements for input, but the recommended way is typing in your expressions.

So, right after you launched of SpeedCrunch, an **input field** is readily awaiting your keystrokes. The text you enter there is always an arithmetic formula describing your desired calculation, as simple as

1.234 + 2.987

or as complex as

*sqrt(2*pi)*exp(ln x * x – 0.5).*

Expressions like these look very much like the formulas in mathematics textbooks, and this form of input is known as **algebraic** notation.

Once you have entered a formula, you press the ENTER key for evaluation, and the result will be displayed, along with your formula, in the window above the input field, called **result window**.

The result window does not only show your last operation, it saves a history of what you entered before. This gives you an overview over all your calculations, and you may recall any of your former input by, well, clicking on the corresponding position in the result window, or, faster, by using the UP- and DOWN-ARROW keys.

By now, you should already be able to use SpeedCrunch effectively in an intuitive way. Maybe, you want to enter some expressions of your own now, and learn how SpeedCrunch reacts on your input.

Algebraic notation

What now follows, is a short introduction to algebraic notation. This is not about teaching you math again, we just want to introduce some terms for later reference, and give you an overview over the input syntax.

A simple expression like

1 + 2

has three elements: Two **operands**, namely 1 and 2, and a **binary operator** +. Binary, because the operator takes two operands, in contrast to **unary operators**, that take one operand only. One such operator is the minus sign, or negation:

- pi.

Because it prepends its operand, it is called a **prefix** operator. Unary operators following their operand

are consequently called **postfix** operators, like the exclamation mark in

8!

When there is more than one operator present in a formula, conflict situations arise. How shall a formula like

*1 + 2 * 3*

be executed? Addition first, or multiplication?

Maybe you remember from school, that there comes a **precedence order** with operators. That way, the rules of algebraic formulas clarify some of these conflicts, by assigning a higher precedence to the multiplication than to the addition, for example.

SpeedCrunch knows this precedence order and applies it to your input. So the result of the above formula is 7, not 9, as would be with strict left-to-right evaluation.

Another way to establish an evaluation order is the use of **parenthesis**. In case of doubt about the precedence order, the use of parenthesis is recommended.

So the formula 1 + 2 * 3 may be restated more clearly as

*1 + (2 * 3).*

When operators are different, precedence rules can tell how to execute a formula. But they fail to resolve conflicts with series of operators of the same kind, or precedence, as in

125 – 13 – 77.

So a second set of rules, called the **associativity** of an operator, determines what to evaluate first then. The evaluation order is almost always from **left-to-right**. So the above example is equal to

(125 – 13) – 77.

We have just one big exception, the power operator ^ (or **):

2 ^ 3 ^ 4

means 2 ^ (3 ^ 4). Its associativity is **right-to-left**.

SpeedCrunch implements a set of the most common mathematical functions like log, sin and so on. Functions take **parameters**, sometimes just a single one, or up to arbitrarily many. You start a function call by typing the **function name**, followed by the parameter(s), the **parameter list**, enclosed in parenthesis:

ln(2) / ln(10).

If you submit more than one parameter, separate them by a semicolon:

nCr(17; 3).

A function like average takes arbitrarily many parameters:

average(1; 2; 3; 4; 5; 6)

Each parameter in turn can be an expression:

geomean(sqrt(3); tan 4; 7+5)

As an abbreviation, if a function takes just one parameter, you sometimes may omit the parenthesis. For instance, we may rewrite one of the examples above as

ln 2 / ln 10.

But if you want to evaluate

ln (2 / 3),

you need the braces. Functions eat up as much of your following input as is necessary to complete the needed parameter, but not more. So, without braces, the above statement is interpreted as

(ln 2) / 3,

because the number 2 is already recognized as a valid parameter. Again, if you are in doubt of how SpeedCrunch interprets your input, use parenthesis.

Here is an example of a more complex parameter involving operators and functions:

ln - ln ln ln 3,

which is equivalent to ln(-ln(ln(ln 3))).

SpeedCrunch offers the usage of variables. You **assign** a value to a variable by using the '=' character:

x = sqrt 2.

If x existed before, its former value is now overwritten by the square root of 2. If there was no such variable, it is created and assigned the given value.

Once you have defined a variable, you may use it in any location, where a number constant is valid. So, with the above definition of x,

*x * x*

should yield the result 2.

It is allowed to use the same variable on the left and right hand side of '=':

*x = 2 * x*

doubles the contents of x. The expression to the right of the '=' is evaluated first, using the old value of x, before the assignment takes place and overwrites the value of x.

Literal number constants

The most basic sort of input to a calculator is that of a stating a number value. SpeedCrunch follows the usual input rules well known from other programs, so your intuition will be almost always right.

However, there are a few corner cases, that should be mentioned here. We start with the **decimal dot**, a character, that is not fixed in SpeedCrunch. There are countries, that use a comma instead of a dot, so you may customize SpeedCrunch to your particular needs. The **Settings** menu provides a dialogue window, where you can pick your favourite character. Throughout this manual, we will assume you chose a dot.

A preceding minus is never part of a number literal; it is treated as a unary operator applied to following unsigned literal constant. This is sometimes visible, for example in -0.5!, which is executed as -(0.5)! . And, unlike common usage, unary operators take precedence over the power (^ or **) operator. This is likely to change in a future release.

fixpoint format

Integers are given as a sequence of digits,

1234

for example.

The first digit need not be non-zero, as in

0023.

SpeedCrunch is capable of holding integers up to 78 digits, without a loss of a digit. If your input has more than 78 significant digits, this is silently converted into an approximate value always of the right scale, but with a truncated digit sequence.

Real numbers are characterized by a non-zero fractional part.

12.345

is an example of such a number. You may append as many trailing zeros to a **fraction** as you like, or put leading zeros to the front of the **integral part**, as in

0012.3450000

If the integral part of a number is zero, you may omit the zero altogether and begin your number with the decimal dot:

.005

is the same as 0.005.

Real numbers can hold up to 78 significant digits; if you state more, an approximate value of the right scale, but with a truncated digit sequence is stored. Apart from this truncation, your literal constants enter calculations without conversion or rounding errors, because SpeedCrunch uses internally a decimal encoding.

scientific format

The most general type of a number literal is given by the **scientific format**. Here, the **scale** (or magnitude) of a value is separated from its sign and digit sequence, the **significand**. This format allows simple input of huge or tiny numbers, whose first significant digit lies way too far away from the decimal dot for common number representation. You will find this format on scientific pocket calculators as well, so maybe, you are already familiar with this format.

The scale of a number (sometimes called its exponent) begins with the scale character 'e' or 'E' followed by a signed integer. So 'e+10', 'e-4', 'E-0' are all valid scale expressions. If the sign is '+', you may simply omit it: 'e0', 'E10'.

The meaning of a scale expression is ' * 10^{exp} ', where exp is the signed integer value following the scale character. As you might see from this, a standalone scale expression is meaningless, because you need another factor to make up a complete product. This other factor is called the significand (or mantissa), and it always precedes the scale. A valid significand is any ordinary (non-scaled fix point) real number or integer, given as a number literal according to the previous section. Here's a list of valid number literals in scientific format, meaning $1.2 * 10^{3}$, $7 * 10^{-4}$, $0.3 * 10^{12}$, $14.8 * 10^{-9}$, resp.

1.2e+3

7e-4

.3e12

14.8e-9

Usually, significands are normalized, i.e. they fulfil 1 <= significand < 10, but the listed examples show, that this is not required by SpeedCrunch.

If the scale value is zero, the scale is effectively redundant, because ' * 10^0 ' reduces to ' * 1'. So adding a scale 'e0', 'E-0' to a number does not make any difference.

0 can be combined with any scale value without affecting the result, so 0e0, 0e100000, 0e-10000 will all yield 0.

By giving huge scale values, you can produce numbers near the overflow or underflow limit of SpeedCrunch (1e+/-536870912). Overflow or Underflow is not determined by the scale value alone; it is the product of scale and significand that determines the result. So in

0.000001e536870916

the scale part overflows; but it is valid input, though, because the final result, 1e536870910, is below the limit.

You can specify just a single scale in a literal number constant. 1e10e5 is not grammatical.

input to bases 2, 8 and 16

The handling of binary encoded numbers is still not well integrated into SpeedCrunch; there is work in progress to relieve this situation. But the necessary changes to the code base of SpeedCrunch proved to be too massive to be ready for version 0.10.

For the time being some preliminary support is provided.

You can enter unsigned integers up to 2^{256}-1 directly by adding a **radix prefix** to the front of the digit sequence. Valid prefixes are

- '0x', '0X', '0h', '0H', '#' for hexagesimal digit sequences;
- '0o', '0O' for octal digit sequences. Note: it does not suffice to start a digit sequence with 0 to mark it as octal;
- '0b', '0B' for binary digit sequences;
- '0d', '0D' for decimal sequences. This is redundant, SpeedCrunch will assume this by default.

If you enter a hexagesimal digit sequence, the digits '10' to '15' are, as usual, represented by the letters 'A' to 'F'. You may use their lower case form as well.

So valid input is:

0xFA67

0b1001000100100111

0o7212627

#8a

As mentioned, all these values are unsigned integers. You cannot specify -1 by 0xFFFFFFFF, for

instance. You can create negative numbers by

- use of negation: the following yields -10

  -0xA

- use of the function **unmask** to convert an unsigned integer to its signed counterpart: the following evaluates to -1

  unmask(0xFFFF; 16)

SpeedCrunch can switch output mode to other radices than 10 as well, but this affects output only; for input you still need the radix prefixes. So even in 'hex' output mode, you must not forget the '0x' (or #, or..) tag on hexagesimal input.

You may freely mix operands to different bases in a calculation:

*17 * 0xFA – 0b10010001*

or even

sqrt(0xff),

although the real valued result is fully shown in decimal output mode only.

Math engine

All calculations are done in a program module called the 'math engine'. We provide you with some internals of this module here, should this knowledge be necessary to understand SpeedCrunch's behaviour in corner cases. Usually, you can safely skip this paragraph.

SpeedCrunch stores and evaluates all values in a normalized scientific format. The significand keeps the sign and 78 decimal digits plus three guard digits. 78 digits have been chosen to allow keeping a 256 bit unsigned integer without loss of digits.

The three guard digits are private to the engine; you must not rely on their values. They are meant to collect all round off errors, and prevent them from polluting the first 78 digits. Their existence is occasionally visible, for example, when you subtract two almost equal numbers. The guard digits never leave the engine; on export the rounded result to at most 78 digits are written.

The significand of an internal value is either 0, or its absolute value is from 1 <= significand < 10. So the scale describes a power of 10 and is a plain signed integer of 30 bits width, so it can assume values between -536870912 and 536870911. These are the bounds of the valid data range of the engine. Any bigger or smaller scale produces an overflow or underflow error. An intermediate result is subject to this bound checking as well, so you may receive an overflow or underflow error, even if the final result would be representable in the engine's value range.

A special encoding is reserved for a so-called 'Not a Number' (or short: NaN). This is the value returned on error. The engine returns a NaN along with an error code describing the kind of error in more detail, but the current SpeedCrunch implementation discards the error code and simply reports “NaN”.

The engine recognizes the following types of error:

- NaN operand: You tried to perform an operation on a NaN. This NaN is most probably the result of a former failed operation that now propagates through the rest of your computation, but it may indicate a not initialized value as well.

- Overflow, Underflow: A (final or intermediate) result violates the bounds of valid numbers
- Too Expensive: Some operations can be potentially very expensive, like the modulo operation. In order to prevent SpeedCrunch from freezing, this error is issued, when the engine detects a situation that would require too much time.
- Unstable algorithm: If you try to evaluate a function very close to a pole, the result might be dependent on the guard digits only. Instead of returning an almost random result, the engine issues this error. The same holds, if you try to evaluate a periodic function in a number range where its periodicity is not recognized any more, because any change of the 78th digit of the argument covers already more than one cycle of the function.
- Out of Domain: some functions like ln or sqrt cannot be applied to all real numbers. If you pass an invalid parameter to such a function, you get this error.
- Divide by Zero: You tried to divide by zero, evaluated a function at a pole, or used 0 as a modulo.

The internal decimal encoded scientific format is not appropriate for logic operations. You need binary encoded operands for this. A conversion unit in the math engine takes care of that. The conversion is limited to integers ranging from -2^{255} to $+2^{256} - 1$. Negative numbers are converted to a 256 bit two's complement representation. Positive numbers yield a bit pattern representing their value in binary encoding. Note that the conversion is sometimes ambiguous: -2^{255} and $+2^{255}$ have the same bit pattern. Although logic operations take integers $>= 2^{255}$ as operands, their result are always signed integers from -2^{255} to $+2^{255} - 1$.

Unary operators

Here comes a list of all unary operators:

unary prefix operator +

is effectively a “no operation”, and therefore always redundant.

+++2.77

is the same as 2.77.

The + operator is higher in precedence than any binary operator, but lower than postfix operators.

A + preceding a literal number constant is not recognized as part of that constant, but treated as an unary operator instead (albeit this difference does not matter).

A single + preceding a scale integer is a redundant part of that constant:

1e+33

is OK, but “1e++33” is not, because inside a literal constant no computation takes place, and this operator is not applied to remove any of the +.

This operator is applicable to all real numbers.

unary prefix operator -

negates a number.

--2.77

is the same as 2.77.

The - operator is higher in precedence than any binary operator, but lower than postfix operators.

A - preceding a literal number constant is not recognized as part of that constant, but treated as an unary operator instead. This is visible in expressions like -0.5!, which is evaluated as -(0.5!), because the postfix operator ! takes precedence over the unary operator -.

A single - preceding a scale integer is part of that constant:

1e-33

is OK, but "1e+-33" is not, because inside a literal constant no computation takes place, and this operator is not applied to find the sign of the scale.

This operator is applicable to all real numbers.

unary postfix operator !

finds the factorial of the preceding value:

10!

returns 10*9*8*7*6*5*4*3*2*1 == 3628800.

Maybe this comes as a surprise to many users, but mathematicians have interpolated the factorials in a "decent" way. So

(-0.5)!

turns out to be the square root of pi.

SpeedCrunch knows of this extension and returns factorials even of non-integers.

The factorial postfix operator has higher precedence than the unary operators + or -, so

-0.5!

is evaluated to -(0.5!) == -sqrt(pi) / 2.

The factorial of negative integers cannot be computed.

The factorial is growing tremendously with its argument, and overflows near 72306960.008... On the negative axis it underflows near -72306961.008... Of course, the poles at negative integers return errors, too.

Binary operators

Here's a list of all binary operators.

binary operator +

performs an ordinary addition.

12.73 + -465.44

The precedence of this operation is lowest.

The addition is an inherently unstable operation, when you add two operands of almost the same value, but opposite signs. This means, the result depends on less significant digits of both operands,

and is typically not valid to all 78 digits.

When adding values near the overflow or underflow limit of SpeedCrunch, this operation may overflow or underflow.

binary operator -

performs an ordinary subtraction.

-12.73 – -465.44

The precedence of this operation is lowest.

The subtraction is an inherently unstable operation, when you subtract two operands of almost the same value. This means, the result depends on less significant digits of both operands, and is typically not valid to all 78 digits.

When subtracting values near the overflow or underflow limit of SpeedCrunch, this operation may overflow or underflow.

binary operator *

performs an ordinary multiplication.

*-12.73 * -465.44*

The precedence of this operation is higher than + or -, but lower than a unary -.

For very huge or very tiny operands, this operation may overflow or underflow.

binary operator /

performs an ordinary division.

-12.73 / -465.44

The precedence of this operation is higher than + or -, but lower than a unary -.

For very huge or very tiny operands, this operation may overflow or underflow.

You cannot divide by zero.

binary operator ^

binary operator **

Both operators are synonyms for the power operator.

12.4 ^ 3

raises 12.4 to the 3rd power (== 12.4 * 12.4 * 12.4 == 1906.624).

The precedence of this operator is higher than unary + or -, but lower than function calls.

Using logarithms, the raising of positive numbers to any real-valued power is well-defined:

12.4 ^ -3.7

returns a result near 0.00009.

You can draw the n-th root of a number by raising it to the (1/n)-th power:

*12.4 ** (1/5)*

draws the 5-th root of 12.4 (== 1.654..).

A power of zero is zero, if the exponent is positive, otherwise it is an error.

If the exponent has no fractional part (is an integer), SpeedCrunch allows negative numbers as bases:

(-2.6) ^ -5

is OK.

However, in general, raising a negative number to an arbitrary non-integer exponent is not possible for SpeedCrunch, because

- there always exist at least two, usually infinite many results;
- these results are, in rare cases with just one exception, true complex values, especially the principal value is always complex.

The rare cases are when the exponent is rational, and its denominator is odd.

For example, the 3rd root (or (1/3)-rd power) of (-1) has three solutions in the complex plane, namely (sqrt(3)/2 + 1/2*i), (-1) and (sqrt(3)/2 – 1/2*i). Of these, the first is denoted by mathematicians as the principal value (and not (-1), that is so nicely real-valued).

Without delving too deep into theory here, one can see, that it is impossible to spot the few exceptions, where a single negative real-valued result (among many others) exists. Since the exponent is passed approximately only (1/3 has been converted to 0.3333... before) to the power routine, it cannot safely be identified as of rational origin any more.

So SpeedCrunch always returns an error, when it is requested to raise a negative number to a non-integer power.

We explained the reasons here in some detail, because people are sometimes confused when they learn that cbrt(-2) yields a result, but (-2)^(1/3) does not.

Especially when exponents are very large, the power operation overflows or underflows easily.

Statistical Functions

probability distributions

SpeedCrunch supports common probability distributions. Since many distributions define the same set of functions and characteristic values, we will describe their common meaning here.

A single probabilistic **test** or **trial** (like throwing a dice) has a random outcome. There is some freedom about how we define outcomes. We can be very specific, e.g. enumerating all possible results of a dice throw, or we can subsume several of them, e.g. see whether a “six” is shown, or not. Such possibly subsumed outcomes are called **events**. In addition, we will assume here, that you can describe events sufficiently by a single number. If the possible results reduce to a set of two mutually exclusive events, we talk about a **yes/no test**.

All (numerical) outcomes are either

- from a fixed finite set of possible outcomes (like the faces of the dice);
- from a continuous range (like the length of an object).

If the set of outcomes is finite, you can assign each of the outcomes a value, its **probability** p, that describes how likely this outcome is. It is always a real value between 0 and 1, bounds inclusive (Multiplied by 100, this value gives the chance of the specific outcome in percent). E.g., if you use unbiased dices, the probability of the outcome "six" is exactly 1/6, as it is for all other possible results. The function mapping the possible (and mutually exclusive) events to probability values is called the **probability mass function (pmf)**. Since the set of outcomes is limited, we speak of a discrete probability distribution.

If you deal with results from a continuous range (like real values), you must somehow derive a **probability density function** f(x), that allows you to compute how likely an outcome lies between two given bounds. If the bounds are very close to each other, the probability is roughly given by f(x)*(upper limit – lower limit), but for arbitrary limits you need to know the integral function P(x) of f(x). The probability of a test result to lie between any two limits is then given by P(upper limit) – P(lower limit). A very famous and fundamental example is that of the bell curve describing a **normal distribution**. This function, in its purest form, is analytically given by the function

f(x) = exp(-x*x)/sqrt(pi),

and its integral function (actually twice its integral) is implemented as error function **erf**(x) in SpeedCrunch.

Since we assume outcomes to be numeric, questions like "how likely is it that the outcome is below a bound" (note that the probabilistic mass function only answers "how likely is it that the outcome is exactly a given value"), and as an extension, "how likely is it, that a result lies between two given bounds" arise. In the continuous case, we saw that the integral of a probability function can tell the answer. In the discrete case, we can derive a similar function from a distribution function, called the **cumulative distribution function (cdf)**. And again, a simple difference tells the desired answer: cdf(upper limit) – cdf(lower limit) is the probability, that a test result is less or equal to upper limit, but greater than the lower limit.

distributions of series of tests

Up to now, we talked about a single test or probabilistic experiment only. In practice, we often have to deal with a series of tests. If the outcome of one test does not influence the following test, the tests are said to be **independent**. Casting a dice is an example of this, but drawing a card from a deck is not, because the removed card modifies the probabilities of the following test.

The outcome of a series of tests is naturally not a single value any more, but a tuple of the size of the test count. In the discrete case, again, we are able to assign any such tuple a probability, now based on the probabilities of each test. Since this value can be computed, statisticians have developed a theory around test series and derived probability distributions on tuples rather than on single outcomes, and clarified how these distributions depend on that of the tests. Some of the simpler distributions have been implemented in SpeedCrunch.

Associated with distributions are some characteristic values. One of these is the **mean** or **expected value**. This value is the (expected) average outcome in a very long series of tests. The second such value is the **variance** of a distribution (roughly: the average of the squares of distances to the mean), that says something about the spread of the distribution.

The **binominal distribution** is based on a series of independent and equal yes/no tests. The likeliness of a certain event in each test is the probability p. The concrete nature of the test is of no importance, just p and the assumed independence. The binominal distribution tells how likely it is that exactly r events occur in a series of n tests. The associated cdf function returns how likely it is that at most r

events occur.

The **hypergeometric distribution** handles cases where you know the number of a specific event in a series of tests in advance. It answers the question, how likely you will have a certain amount of these events in any given subset of these tests. The associated cdf function tells how likely there will be up to a given count of events in any subset of the series.

The **Poisson distribution** handles the case where you know the average occurrence of an (independent) event in a given interval (often a time interval), and assume it has the same variance , and you now want to know, how likely the occurrence of exactly n events in this interval is. The associated cdf function tells how likely there will be up to a n events in this interval.

function list

Here's an alphabetical list of implemented functions:

abs (x)

returns |x|, turning negative numbers into positive ones, and leaving positive ones unchanged.

abs (-1)

and

abs (1)

both yield 1.

acos (x)

returns the arc cos (x), the inverse function to cos y.

The return value is dependent on the degree setting; if “degrees” is enabled, the return value is from the range 0 <= result <= 180, else from 0 <= result <= π.

The arc cos(x) is the principal solution to the equation cos (y) = x. Other solutions are (radians)

- arc cos (x) + 2*n*π for all integers n;
- -arc cos (x) + 2*n*π for all integers n.

or (degrees):

- arc cos (x) + n*360 for all integers n;
- -arc cos (x) + n*360 for all integers n.

The valid range of the parameter is -1 <= x <= 1.

Example:

acos (0.5)

Related to this function is the arc sec (x), found by

acos (1/x)

and (x1; x2; ...)

Performs a bitwise logical AND on the submitted parameter(s) (one or more).

All parameters have to be integers from the range -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers). The result is from -2^{255} to $+2^{255} - 1$ (signed integer).

Note that and(x) is not the identity, because the unsigned $+2^{255}$ is mapped to the signed -2^{255}, for example.

An error is returned, if a parameter is not from the valid range.

Example:

hex (and (29; 0x33; 0b11111))

arcosh (x)

computes the area hyperbolic cosine of x, the inverse function to cosh y.

arcosh (x) is the positive solution to cosh (y) = x.

Except for 1, there exists a second solution to this equation: -arcosh (x).

The parameter x has to be >= 1.

Example:

arcosh (4.7)

Related to this function is arsech, the area hyperbolic secant, which can be computed by

arcosh (1/x)

arsinh (x)

computes the area hyperbolic sine of x, the inverse function to sinh y.

arsinh (x) is the (only) solution to sinh (y) = x.

Example:

arsinh (-0.47)

Related to this function is arcsch, the area hyperbolic cosecant, which can be computed by

arsinh (1/x)

artanh (x)

computes the area hyperbolic tangens of x, the inverse function to tanh y.

artanh (x) is the only solution to tanh (y) = x.

The parameter x has to fulfil -1 < x < 1.

Example:

artanh (-0.7)

Related to this function is arcoth, the area hyperbolic cotangent, which can be computed by

artanh (1/x)

asin (x)

returns the arc sin (x). the inverse function to sin.

The return value is dependent on the degree setting; if “degrees” is enabled, the return value is from the range -90 <= result <= 90, else from -π/2 <= result <= π/2.

The arc sin(x) is the principal solution to the equation sin (y) = x. Other solutions are (radians)

- arc sin (x) + 2*n*π for all integers n;
- -arc sin (x) + (2*n+1)*π for all integers n.

or (degrees):

- arc sin (x) + n*360 for all integers n;
- -arc sin (x) + n*360 + 180 for all integers n.

The valid range of the parameter is -1 <= x <= 1.

Example:

asin (0.5)

Related to this function is the arc csc (x), found by

asin (1/x)

atan (x)

returns the arc tan (x), the inverse function to tan y.

The return value is dependent on the degree setting; if “degrees” is enabled, the return value is from the range -90 < result < 90, else from -π/2 < result < π/2.

The arc tan(x) is the principal solution to the equation tan (y) = x. Other solutions are

(radians): arc tan (x) + n*π for all integers n;

or

(degrees): arc tan (x) + n*180 for all integers n;

Example:

atan (500)

Related to this function is the arc cot (x), found by

atan (1/x)

average (x1; x2; ...)

computes the average, or mean value of the submitted parameter(s) (one or more).

Example:

*average (13.5; 3*3; 0; -4)*

bin (n)

Displays n in binary format, without globally changing the output format. A fractional part of n is cut

off before conversion, and the remaining integral part has to be from the range -2^{255} to $+2^{256} - 1$.

Example:

bin (1000/13)

returns 0b1001100, the binary encoding of 76.

binomcdf (max; trials; p)

trials is a non-negative integer, max a positive integer, and 0 <= p <= 1.

This function determines the probability, that in a series of 'trials' independent probabilistic tests, each resulting in an event with probability p, at most 'max' events occur.

binommean (trials; p)

trials is a non-negative integer, 0 <= p <= 1.

If you have 'trials' independent tests, each of them resulting in a given event with probability p, this number tells you, how many events you will have on the average.

binompmf (hits; trials; p)

trials is a non-negative integer, hits a positive integer, and 0 <= p <= 1.

This function determines the probability, that in a series of 'trials' independent probabilistic tests, each resulting in an event with probability p, exactly 'hits' events occur.

binomvar (trials; p)

trials is a non-negative integer, 0 <= p <= 1.

The variance of the binominal distribution function, based on 'trials' independent tests, each resulting in an event with probability p. In other words, it tells how 'good' the result of binommean is.

cbrt (x)

computes the third (cubic) root of x. This function operates on negative numbers as well. The inverse function is y ^ 3.

Example:

cbrt (-27)

is -3.

ceil (x)

Finds the smallest integer greater or equal to x, the ceiling of x.

ceil (-2.4)

is -2,

ceil (2.4)

is 3, and

ceil (-2)

is -2.

cos (x)

evaluates the cosine of x. The result is dependent on the degree setting. If “degrees” is active, the parameter x is assumed to be an arc from a 360° circle, otherwise a full circle is represented by 2*π.

The inverse function is acos(y).

Although the cosine is mathematically defined for all x, for x beyond approximately 1e77, the periodicity of the cosine is not recognized any more, so SpeedCrunch issues an error then. Example:

cos (-1000)

cosh (x)

Finds the hyperbolic cosine of x.

This function overflows for |x| > 1236190959.52...

The inverse function is arcosh.

Example:

cosh (7.22)

cot (x)

evaluates the cotangent of x. The result is dependent on the degree setting. If “degrees” is active, the parameter x is assumed to be an arc from a 360° circle, otherwise a full circle is represented by 2*π.

Although the cotangent is mathematically defined for all x, for x beyond approximately 1e77, the periodicity of the cotangent is not recognized any more, so SpeedCrunch issues an error then. Example:

cot (-1000)

csc (x)

evaluates the cosecant of x. The result is dependent on the degree setting. If “degrees” is active, the parameter x is assumed to be an arc from a 360° circle, otherwise a full circle is represented by 2*π.

Although the cosecant is mathematically defined for all x, for x beyond approximately 1e77, the periodicity of the cosecant is not recognized any more, so SpeedCrunch issues an error then. Example:

csc (-1000)

dec (x)

Displays x in decimal format, without globally changing the output format. This function is useful when you changed the output format to a binary format.

Example:

dec (1000/13)

returns 76.923.., even if you have activated hex output mode for instance.

degrees (x)

converts an angle measured in radian to degrees. This function is especially useful, if you want to override the current angle mode for just one expression. Example:

degrees (acos (0.5))

tan (degrees (1))

degrees (pi)

erf (x)

determines the normalized error function of x, the cumulative distribution function of the normal distribution (bell curve, Gaussian distribution).

Analytically, it is the integral function of 2 * exp(-x*x) / sqrt(π).

Example:

erf (0.5)

erfc (x)

determines the normalized complementary error function of x, related to the cumulative probability function of the normal distribution (bell curve, Gaussian distribution). Erfc(x) + erf(x) is always 1.

Example:

erfc (0.5)

erfc underflows for x > 35159.507..

exp (x)

evaluates the natural exponential function to the base e == 2.71828...

The inverse function is ln(x).

exp (x) overflows/underflows for |x| > 1236190958.83...
Example:

exp (ln 2 + ln 3)

floor (x)

Finds the greatest integer less or equal to x, the floor of x.

floor (-2.4)

is -3,

floor (2.4)

is 2, and

floor (-2)

is -2.

frac (x)

cuts off the integral part of a real number.

frac(-2.3)

returns -0.3.

gamma (x)

Γ(x), equivalent to (x-1)!.

gcd (n1; n2;...)

returns the greatest common divisor of the integers n1, n2 ... (2 or more)

You can use this function to reduce a rational number. If a rational number is given as p / q, its reduced form is (p / gcd(p;q)) / (q / gcd(p;q)).

Related to the gcd is the lcm function (least common multiplier). You can find the lcm by:

lcm(n1; n2) = n1*n2 / gcd(n1; n2), e.g

*720 * 486 / gcd (720; 486)*

is 19440, the lcm of 720 and 486

geomean (x1; x2;...)

finds the geometric mean of the positive numbers x1, x2... (one or more parameter).

Non-positive parameters are not allowed.

Example:

geomean (17; ln 5; 8+2)

hex (n)

Displays n in hexagesimal format, without globally changing the output format. A fractional part of n is cut off before conversion, and the remaining integral part has to be from the range -2^{255} to $+2^{256} - 1$.

Example:

hex (1000/13)

returns 0x4C, the encoding of 76.

hypercdf (max; total; hits; trials)

All parameters are integers, 0 <= max <= hits <= total, 0 <= trials <= total, and 0 < total.

Assume there are 'hits' events in a series of 'total' tests. Then this function evaluates the probability, that you will find up to max events of them in a subseries of 'trials' tests.

hypermean (total; hits; trials)

All parameters are integers, 0 <= hits <= total, 0 <= trials <= total, and 0 < total.

Assume there are 'hits' events in a series of 'total' tests. Then this function evaluates the expected count of events in a subseries of 'trials' tests.

hyperpmf (count; total; hits; trials)

All parameters are integers, 0 <= max <= hits <= total, 0 <= trials <= total, and 0 < total.

Assume there are 'hits' events in a series of 'total' tests. Then this function evaluates the probability, that you will find exactly count events in a subseries of 'trials' tests.

hypermean (total; hits; trials)

All parameters are integers, 0 <= hits <= total, 0 <= trials <= total, and 0 < total.

Assume there are 'hits' events in a series of 'total' tests. Then this function evaluates the variance of the count of events in a subseries of 'trials' tests. Or, in other words, it tells, how 'good' the result of hypermean is matched on the average, if you repeat your subseries of trials very often.

idiv (dividend; divisor)

Evaluates the integer part of dividend / divisor. The result is always exact.

Since SpeedCrunch can hold integers up to 78 digits only, this function overflows even if the ordinary division operator / would yield an approximate (but huge) quotient.

It is possible to apply the idiv function to non-integers as well, but be aware that rounding errors might lead to off-by-one errors. If idiv detects, that a result depends on the validity of the guard digits, it returns a NaN as a warning, e.g. idiv(1/3*3; 1) is NaN, because the first parameter is 0.99..., and idiv cannot safely decide whether to return 0 or 1.

Examples:

idiv (17; -3)

is -5.

idiv (1.23; 0.034)

is 36.

int (x)

returns the (signed) integer part of x.

Example:

int (-2.3)

return -2,

int (0.3)

returns 0.

lg (x)

returns the base 2 logarithm of x.

x has to be positive.

Examples:

lg (16)

is 4.

lg (3.1)

is 1.632..

ln (x)

returns the natural logarithm of x, the base is e = 2.71828...

x has to be positive.

The inverse function is exp (y).

Examples:

ln (0.2)

lngamma (x)

evaluates ln(Γ(x)), or ln((x-1)!), for positive x.

The factorial or Γ(x) is growing tremendously, and overflows for moderate values already. But this function will even return a result, when the ! operator overflows.

Example:

lngamma 1e1000 / ln 10

shows that $(10^{1000})!$ has about $9.996 * 10^{1002}$ decimal digits.

log (x)

returns the base 10 logarithm of x.

x has to be positive.

Examples:

log (10)

is 1.

log (3.1)

is 0.49136...

mask (n; bits)

returns the lower 'bits' bits of the integer n. The result is always unsigned.

n has to be from -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers), bits is an integer ranging from

1 to 255 inclusive.

One application of this function is to find the two's complement representation of a value. So

hex (mask (-1; 16))

returns 0xFFFF, the 16-bit two's complement bit pattern of -1.

max (x1; x2 ..)

takes one or more parameter(s).

returns the maximum value of the submitted parameters.

Example:

max (17; log 1000; exp(3))

min (x1; x2 ..)

takes one or more parameter(s).

returns the minimum value of the submitted parameters.

Example:

min (-0.4; log 0.1; exp(-3))

mod (value; modulo)

evaluates value mod modulo, that is the remainder after finding the integer quotient value / modulo. This function always returns an exact result, if the parameters are exact.

'modulo' has to be non-zero.

If the remainder is non-zero, it takes the sign of value.

You can use this function with non-integers as well, but rounding errors might lead to off-by-one errors.

This function can potentially be very time consuming, so mod is restricted to at most 250 division loops.

The relation x = idiv(x; y) * y + mod(x; y) holds.

Examples:

mod (-17; 3)

is -2, and

mod (1.23; 0.034)

is 0.006 (exact)

nCr (x1; x2)

For integer parameters this function is equal

- to the number of combinations (order is of no importance) of x2 elements from an x1-sized set of elements ;

- to the binominal coefficient x1 over x2.

For arbitrary real-valued x1, x2 the result is 1/((x1+1)*B(x2+1, x1 – x2+1)), where B(a, b) is the complete Beta function.

Examples:

How many combinations of 12 elements out of 17 are there: (6188)

nCr (17; 12)

Find the (binominal) coefficient of x^3 in the series expansion of sqrt(1+x): (0.0625)

nCr (0.5; 3)

Evaluate the complete Beta function for a = 2.3 and b = 7.6: (0.00915..)

1 / (7.6 + 2.3 – 1) / nCr (7.6 + 2.3 – 2; 2.3 – 1)

not (n)

Evaluates the bitwise (one-) complement of n. The result is a signed 256-bit integer.

n has to be an integer from the range -2^{255} to $+2^{256} – 1$ (signed or unsigned 256 bit integers).

Examples:

not (-1)

is 0.

hex (mask (not (0xA77); 32))

returns the 32-bit encoding of not(0xA77) == 0xFFFFF588

nPr (x1; x2)

For integer parameters this function is equal

- to the number of permutations (order is of importance) of x2 elements from an x1-sized set of elements ;
- the falling factorial (or Pochhammer symbol) x1 * (x1 – 1) * .. * (x1 – x2 + 1).

For arbitrary real-valued x1, x2 the result is Γ(x1+1) / Γ(x2), where the poles of the Γ-function are handled correctly.

Examples:

How many permutations of 12 elements out of 17 are there: (2964061900800)

nPr (17; 12)

Find the product 2.5 * 1.5 * 0.5 * ... * (-2.5): (-3.515625)

nPr (2.5; 6)

oct (n)

Displays n in octal format, without globally changing the output format. A fractional part of n is cut off before conversion, and the remaining integral part has to be from the range -2^{255} to $+2^{256} – 1$.

Example:

oct (1000/13)

returns 0o114, the octal encoding of 76.

or (x1; x2; ...)

Performs a bitwise logical OR on the submitted parameter(s) (one or more).

All parameters have to be integers from the range -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers). The result is from -2^{255} to $+2^{255} - 1$ (signed integer).

Note that or(x) is not the identity, because the unsigned $+2^{255}$ is mapped to the signed -2^{255}, for example.

An error is returned, if the parameters are not from the valid range.

Example:

hex (or (29; 0x33; 0b11111))

poicdf (events; avgevnts)

'events' is a non-negative integer, 'avgevents' a positive real number.

If you have on the average 'avgevents' independent events in an interval, this function returns the probability, that you will see up to 'events' events when you test an interval.

poimean (avgevents)

avgevents >= 0

trivial, returns the average number of events in an interval, when this average is 'avgevents'

poipmf (events; avgevnts)

'events' is a non-negative integer, 'avgevents' a positive real number.

If you have on the average 'avgevents' independent events in an interval, this function returns the probability, that you will see exactly 'events' events when you test an interval.

poivar (avgevents)

avgevents >= 0

Trivial, the Poisson distribution assumes, that the variance equals the expected value 'avgevents'.

radians (x)

converts an angle measured in degrees to radians. This function is especially useful, if you want to override the current angle mode for just one expression. Example:

radians (acos (0.5))

tan (radians (1))

radians (180)

round (x; digits)

rounds x to 'digits' decimal places after the decimal point.

'digits' has to be an integer.

The rounding is to nearest.

Examples:

round (pi; 2)

yields 3.14.

round (2/3; 5)

gives 0.66667.

If only a single digit '5' was cut off, then rounding is such that the digit before it is even.

round (1.5; 0)

is 2, but

round (0.5; 0)

is 0.

As an extension, 'digits' may be negative. In this case, rounding takes place in the integer part of x.

This rounds a value to full thousands: (23000)

round (22788.76; -3)

sec (x)

evaluates the secant of x. The result is dependent on the degree setting. If “degrees” is active, the parameter x is assumed to be an arc from a 360° circle, otherwise a full circle is represented by 2*π.

Although the secant is mathematically defined for all x, for x beyond approximately 1e77, the periodicity of the secant is not recognized any more, so SpeedCrunch issues an error then. Example:

sec (-1000)

shl (n, bits)

Shifts the bit pattern of n 'bits' bit to the left, if 'bits' >= 0, otherwise shifts n arithmetically by -'bits' to the right.

-255 <= bits <= 255, and n has to be an integer from the range -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers). The result is from -2^{255} to $+2^{255} - 1$ (signed integer).

On a shift left, zero bits are fed in to the right, and shifted out bits are dropped on the left. On a shift right, bit 255 (the leftmost bit) is copied and shifted in from the left, and shifted out bits on the right are dropped.

An error is returned, if the parameters are not from the valid range.

Example:

shl (-1; 3)

shl (0x33721655617; -1)

The last operation is effectively a shift right.

shr (n, bits)

Shifts the bit pattern of n 'bits' bit to the right, if 'bits' >= 0, otherwise shifts n arithmetically by -'bits' to the left.

-255 <= bits <= 255, and n has to be an integer from the range -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers). The result is from -2^{255} to $+2^{255} - 1$ (signed integer).

On a shift left, zero bits are fed in to the right, and shifted out bits are dropped on the left. On a shift right, bit 255 (the leftmost bit) is copied and shifted in from the left, and shifted out bits on the right are dropped.

An error is returned, if the parameters are not from the valid range.

Example:

shr (2^255; 3)

shr (0x33721655617; -1)

The last operation is effectively a shift left.

sign (x)

returns the sign of x, -1 if negative, 1 if positive, and 0 for 0

Example:

sign (sin 10000)

sin (x)

evaluates the sine of x. The result is dependent on the degree setting. If “degrees” is active, the parameter x is assumed to be an arc from a 360° circle, otherwise a full circle is represented by 2*π.

The inverse function is asin(y).

Although the sine is mathematically defined for all x, for x beyond approximately 1e77, the periodicity of the sine is not recognized any more, so SpeedCrunch issues an error then. Example:

sin (-1000)

sinh (x)

Finds the hyperbolic sine of x.

This function overflows for |x| > 1236190959.52...

The inverse function is arsinh.

Example:

sinh (7.22)

sqrt (x)

computes the square root of x. Not defined for x < 0.

-sqrt(x) is the other root to the equation x = y*y.

The inverse function is y ^ 2.

Example:

sqrt (64)

is 8.

tan (x)

evaluates the tangent of x. The result is dependent on the degree setting. If “degrees” is active, the parameter x is assumed to be an arc from a 360° circle, otherwise a full circle is represented by 2*π.

The inverse function is atan(y).

Although the tangent is mathematically defined for all x, for x beyond approximately1e77, the periodicity of the tangent is not recognized any more, so SpeedCrunch issues an error then. Example:

tan (-1000)

tanh (x)

Finds the hyperbolic sine of x.

The inverse function is artanh.

Example:

tanh (7.22)

trunc (x; digits)

truncates x after 'digits' decimal places behind the decimal point.

'digits' has to be an integer.

Examples:

trunc (pi; 2)

yields 3.14.

trunc (2/3; 5)

gives 0.66666.

As an extension, 'digits' may be negative. In this case, truncation takes place in the integer part of x.

This truncates a value after the thousands: (22000)

trunc (22788.76; -3)

unmask (n; bits)

takes the lower 'bits' bits of n and sign-extends them to full 256 bit, meaning bit number 'bits'-1 is

copied to all upper bits.

n has to be an integer from the range -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers), and 'bits' is an integer with 1 <= bits <= 255.

This function can revert the effect of a mask function in many cases, that's why it is named so.

If you are given a negative integer in two's complement notation, it is not possible to enter it directly as a constant, because binary, hex or octal encoded constants are always recognized as unsigned in SpeedCrunch.

This function allows you to overcome this problem. It can convert an unsigned into a signed integer.

Examples:

unmask (0xFFFF; 16)

yields -1,

unmask (0x1FFF; 16)

yields 0x1FFF, because bit 15 is zero, so the sign-extension simply gives the operand.

The 'bits' parameter need not be a power of two. This gives -2:

unmask (0x3FE; 10),

and if the operand has a bit-length greater than 'bits', the upper bits are masked:

unmask (0x10FF; 8)

yields -1, because all upper bits beyond the eighth are cleared before the operand is extended.

xor (x1; x2; ...)

Performs a bitwise logical XOR on the submitted parameter(s) (one or more).

All parameters have to be integers from the range -2^{255} to $+2^{256} - 1$ (signed or unsigned 256 bit integers). The result is from -2^{255} to $+2^{255} - 1$ (signed integer).

Note that xor(x) is not the identity, because the unsigned $+2^{255}$ is mapped to the signed -2^{255}, for example.

An error is returned, if the parameters are not from the valid range.

Example:

hex (xor (29; 0x33; 0b11111))