numlib — library for number theory

Table of contents

The library `numlib` contains function related to number theory.

`numlib::contfrac` – the domain of continued fractions

`numlib::contfrac(a, n)` creates a continued fraction approximation for `a`, using the first `n` digits of its floating-point evaluation.

Creating Elements:

- `numlib::contfrac(a <,n>)`

Parameters:

`n` — positive integer
`a` — numerical expression

Details:

- If `n` is not given, the value of the variable `DIGITS` is used.

Mathematical Methods

Method `approx`: rational approximation

`approx(`*`dom`*` cf, `*`positive integer`*` n)`

- This method returns two rational numbers a and b such that the interval of all rational numbers whose first `n` coefficients coincide with those of `cf` equals $[a, b)$.

Method `unapprox`: find simple continued fraction in interval

`unapprox(`*`numerical expression`*` a, `*`numerical expression`*` b)`

- This method returns a rational number in the interval $[a, b]$, represented as a finite continued fraction, such that the number of coefficients of the continued fraction is minimal; among several such continued fractions (which must agree in all coefficients except for the last), that one with minimal last coefficient is chosen.

Method `_plus`: sum of two continued fractions

`_plus(`*`dom`*` a, `*`dom`*` b)`

- This method converts `a` and `b` into rationals, adds them, and returns the sum converted back into a continued fraction.
- It overloads the function `_plus` of the system kernel.

Method _mult: product of two continued fractions

`_mult(dom a, dom b)`

- This method converts `a` and `b` into rationals, multiplies them, and returns the product converted back into a continued fraction.
- It overloads the function `_mult` of the system kernel.

Method _invert: Inverse of a continued fraction

`_invert(dom a)`

- This method computes $1/a$.
- Inverting a continued fraction means a shift of the coefficients by one to the left or to the right.
- This method overloads the function `_invert` of the system kernel.

Method _power: Integer power of a continued fraction

`_power(dom a, integer n)`

- This method multiplies `a` by itself `n` times, or, if `n` is negative, the inverse of `a` by itself `-n` times.

Example 1. `numlib::contfrac` can also compute continued fraction expansions of irrational numbers:

```
>> a:= numlib::contfrac(PI, 5):
   b:= numlib::contfrac(sqrt(7), 2):
   a, b

                1                                1
      ---------------------- + 3, ------------------- + 2
              1                              1
      ------------------ + 7      --------------- + 1
            1                          1
      ------------- + 15          ----------- + 1
          1                          1
      --------- + 1               ------- + 1
       1                           1
      --- + 292                   --- + 4
      ...                         ...
```

All basic arithmetical operations are available:

```
>> a + b, a*b, a^3
```

```
       1                1               1
-------------- + 5, ------- + 8, ------------ + 31
     1               1              1
----------- + 1     --- + 3     ------- + 159
   1                ...           1
------- + 3                     --- + 3
 1                              ...
--- + 1
...
```

Changes:

- numlib::contfrac used to be contfrac.

---

numlib::decimal – infinite representation of rational numbers

numlib::decimal(q) computes the decimal expansion of a rational number q.

Call(s):

- numlib::decimal(q)

Parameters:

q — nonnegative rational number

Return Value: numlib::decimal(q) returns an expression sequence consisting of nonnegative integers or an expression sequence consisting of nonnegative integers and terminated by a list of nonnegative integers.

Details:

- If q is a nonnegative rational number whose decimal expansion is finite, then numlib::decimal(q) returns the expression sequence starting with the integral part of q and followed by the digits after the decimal point.
- If q is a nonnegative rational number whose decimal expansion is infinite, then numlib::decimal(q) returns the expression sequence starting with the integral part of q, followed by the digits of the pre-period and terminated with a list, containing the digits of a minimal period.
- numlib::decimal returns an error if the argument is a number but not a rational number ≥ 0.

Example 1. Computing the decimal expansion of 1999:

```
>> numlib::decimal(1999)
```

```
1999
```

Example 2. Computing the (finite) decimal expansion of 52187/78125:

```
>> numlib::decimal(52187/78125)
```

```
0, 6, 6, 7, 9, 9, 3, 6
```

Example 3. Computing the (infinite) decimal expansion of 111/7:

```
>> numlib::decimal(111/7)
```

```
15, [8, 5, 7, 1, 4, 2]
```

Example 4. Computing the (infinite) decimal expansion of 37/28:

```
>> numlib::decimal(37/28)
```

```
1, 3, 2, [1, 4, 2, 8, 5, 7]
```

Changes:

- No changes.

`numlib::divisors` – divisors of an integer

`numlib::divisors(n)` returns the list of positive divisors of `n`.

Call(s):

- `numlib::divisors(n)`

Parameters:

`n` — integer

Return Value: `numlib::divisors` returns a list of nonnegative integers.

Related Functions: `ifactor`, `numlib::numdivisors`, `numlib::tau`, `numlib::primedivisors`, `numlib::numprimedivisors`, `polylib::divisors`

Details:

- If `a` is a non-zero integer then `numlib::divisors(a)` returns the sorted list of all positive divisors of `a`.
- `numlib::divisors(0)` returns `[0]`.
- `numlib::divisors` returns an error if the argument evaluates to a number of wrong type.

Example 1. We compute the list of all positive divisors of 72:

```
>> numlib::divisors(72)

            [1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72]
```

Example 2. We compute the list of all positive divisors of -63:

```
>> numlib::divisors(-63)

                      [1, 3, 7, 9, 21, 63]
```

Background:

- Internally, `ifactor` is used for factoring `n`.

`numlib::ecm` – factor an integer using the elliptic curve method

`numlib::ecm(n)` tries to factor the positive integer `n` using the elliptic curve method.

`numlib::ecm(n, BaseBound, s, Step2Bound)` and `numlib::ecm(n, Base, s, Step2Bound)` do the same, with some internal parameters of the algorithm specified – see the "Details" section. The last, or the last two, parameter(s) may be omitted.

Call(s):

- `numlib::ecm(n)`
- `numlib::ecm(n, BaseBound)`
- `numlib::ecm(n, Base)`
- `numlib::ecm(n, BaseBound, s)`
- `numlib::ecm(n, Base, s)`
- `numlib::ecm(n, BaseBound, s, Step2Bound)`
- `numlib::ecm(n, Base, s, Step2Bound)`

Parameters:

`n`	— positive integer
`BaseBound`	— positive integer
`Base`	— list of primes
`s`	— integer
`Step2Bound`	— positive integer

Return Value: `numlib::ecm` returns an integer that divides `n`; the return value may equal 1 or `n`.

Related Functions: `ifactor`

Details:

- Basically, `numlib::ecm` takes an elliptic curve modulo `n` and a point on that curve and computes some multiple of that point. This multiplication may fail; in this case, a proper factor of `n` can be found. Otherwise, the point computed is likely to have small order; it is used in a post-processing step.
- The starting point is computed from the parameter `s`. It is chosen at random if `s` is not given.
- The starting point chosen is multiplied either by all primes in `Base`, or all primes below `BaseBound`, or — if neither of both is given — by all primes below 1000.
- The post-processing step consists of a certain number of iterations, determined by the parameter `Step2Bound`. By default, 100 times `BaseBound` (or the maximum of `Base`, respectively) is chosen.

Example 1. We factor an integer using the default parameters.

```
>> numlib::ecm(10000019070000133)

                                10000019
```

Example 2. If too few multiplications on the elliptic curve are carried out, the algorithm is likely to fail.

```
>> numlib::ecm(10000019070000133, 50)
```

```
                         1
```

Background:

- A description of the algorithm can be found in "Speeding the Pollard and Elliptic Curve Methods of Factorization", by Peter Montgomery, Math. of Comp. 48 (177), pages 243-264, January 1987.

Changes:

- No changes.

`numlib::fibonacci` – Fibonacci numbers

`numlib::fibonacci(n)` returns the `n`-th Fibonacci number.

Call(s):

- `numlib::fibonacci(n)`

Parameters:

`n` — a nonnegative integer

Return Value: a nonnegative integer, or the function call with its arguments evaluated.

Details:

- If `n` is a nonnegative integer then `numlib::fibonacci(n)` returns the `n`-th Fibonacci number.
- `numlib::fibonacci` returns an error if the argument evaluates to a number of wrong type. `numlib::fibonacci` returns the unevaluated function call if `n` does not evaluate to a number.

Example 1. We compute the 201-the Fibonnacci number:

```
>> numlib::fibonacci(201)
```

```
        453973694165307953197296969697410619233826
```

Background:

- The n-th Fibonacci number F_n is defined by the recursion formula $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_n + F_{n+1}$.
- `numlib::fibonacci` uses quadratic recursion formulas.

`numlib::fromAscii` – decoding of ASCII codes

If `L` is a list of ASCII codes then `numlib::fromAscii(L)` returns the string coded by `L`.

Call(s):

- `numlib::fromAscii(listOfCodes)`

Parameters:

`listOfCodes` — a list of ASCII codes

Return Value: a string

Related Functions: `numlib::toAscii`

Details:

- ASCII codes of (in MuPAD) non-printable characters, i. e., codes between `0` and `8` and between `11` and `31`, are ignored.
- `numlib::fromAscii` returns an error if its argument is not a list of integers between `0` and `127`, i. e., not a list of legal ASCII codes.

Error: Unexpected 'identifier' [col 3]

Example 1. Non-printable characters are ignored, but tabulator and newline characters are decoded.

```
>> L := [0,1,2,3,9,10,31,10,9,32,45,32,101,105,110,32,
               84,101,115,116,32,61,32,97,32,116,101,115,116]:
```

```
>> numlib::fromAscii(L)

                 "\t\n\n\t - ein Test = a test"
```

Changes:

- No changes.

`numlib::g_adic` – g-adic representation of a nonnegative integer

If `a` is a natural number and `g` is an integer such that $|g| > 1$ `numlib::g_adic(a,g)` returns the `g`-adic representation of `a` as a list $[a_0, \ldots, a_r]$ such that

$$\mathtt{a} = a_0 + a_1 * \mathrm{g} + a_2 * \mathrm{g}^2 + \cdots + a_r * \mathrm{g}^r$$

and $0 \le a_i < |g|$ für $i = 0, \ldots, r-1$ and $0 < a_r < |g|$.

Call(s):

- `numlib::g_adic(par1,par2)`

Parameters:

`par1` — an nonnegative integer
`par2` — an integer whose absolute value is greater then 1

Return Value: a list of nonnegative integers, or the function call with evaluated arguments if one of the arguments is not a number.

Related Functions: `genpoly`, `int2text`, `text2int`

Details:

- `numlib::g_adic(0,g)` returns `[0]`.
- `numlib::g_adic` returns an error if the arguments evaluate to numbers which are not both of the correct type.

Example 1. Computing the dyadic representation of 1994:

```
>> numlib::g_adic(1994,2)

                [0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1]
```

Example 2. Computing the hexadecimal representation of 2001:

```
>> numlib::g_adic(2001,16)

                [1, 13, 7]
```

Changes:

- No changes.

`numlib::ichrem` – **Chinese remainder theorem for integers**

`numlib::ichrem(a,m)` returns the least nonnegative integer x such that $x \equiv \mathtt{a[i]} \pmod{\mathtt{m[i]}}$ for $i = 1, \ldots, \mathtt{nops(m)}$ if such a number exists; otherwise `numlib::ichrem(a,m)` returns `FAIL`.

Call(s):

- `numlib::ichrem(a, m)`

Parameters:

`a` — a list of integers
`m` — a list of natural numbers of the same length as `a`

Return Value: either a nonnegative integer or `FAIL`.

Related Functions: `numlib::lincongruence`

Details:

- The entries in `m` need not be pairwise coprime.
- `numlib::ichrem(a,m)` returns an error if `a` is not a list of integers or `m` is not a list of natural numbers or `a` and `m` are not lists of the same length.

Example 1. Here the moduli are pairwise coprime. In this case, a solution always exists:

```
>> numlib::ichrem([2,3,2],[3,5,7])

                        23
```

Example 2. Here the moduli are not pairwise coprime, and a solution does not exist:

```
>> numlib::ichrem([5,6,8],[20,21,22])

                       FAIL
```

Example 3. Also here the moduli are not pairwise coprime, but a solution nevertheless exists:

```
>> numlib::ichrem([5,6,7],[20,21,22])

                                   4605
```

Changes:

- No changes.

`numlib::igcdmult` – the extended Euclidean algorithm for integers

For integers `a_1,a_2, ...,a_n` `numlib::igcdmult(a_1,a_2, ...,a_n)` returns a list `[d,v_1, ...,v_n]` of integers such that `d` is the nonnegative greatest common divisor of `a_1,a_2, ...,a_n` and `d = a_1*v_1 + a_2*v_2 + ... + a_n*v_n`.

Call(s):

- `numlib::igcdmult(par1,par2, ...)`

Parameters:

`par1` — integer
`par2,...` — integers

Return Value: a list of integers, or the function call with evaluated arguments if some argument is not a number.

Related Functions: `igcd`, `igcdex`

Details:

- `numlib::igcdmult` is an extension of the kernel function `igcdex`.
- `numlib::igcdmult` returns an error if the arguments evaluate to numbers which are not all of the correct type.

Example 1. Computing the greatest common divisor $d \geq 0$ of $455, 385, 165, 273$ and integers v_1, v_2, v_3, v_4 such that $d = 455v_1 + 385v_2 + 165v_3 + 273v_4$:

```
>> numlib::igcdmult(455,385,165,273)

                      [1, -7630, 9156, -327, 2]
```

Changes:

- No changes.

`numlib::invphi` – the inverse of the Euler φ function

`numlib::invphi(n)` computes all positive integers `i` with $\varphi(i) = n$.

Call(s):

- `numlib::invphi(n)`

Parameters:

`n` — a positive integer

Return Value: a list of positive integer numbers.

Related Functions: `numlib::phi`

Example 1. We compute all numbers `i` with $\varphi(i) = 500$:

```
>> s := numlib::invphi(500)

                  [625, 753, 1004, 1250, 1506]
```

Test for correctness:

```
>> map(s, numlib::phi)

                    [500, 500, 500, 500, 500]
```

Changes:

- `numlib::invphi` is a new function.

`numlib::ispower` – test for perfect powers

`numlib::ispower(n)` tests whether `n` is of the form a^k for some positive integers a, k with $a, k \geq 2$.

`numlib::ispower` returns `FALSE` if `n` is not a perfect power.

Call(s):

- `numlib::ispower(n)`

Parameters:

`n` — an integer

Return Value: `numlib::ispower` returns a sequence of two positive integers ≥ 2, or `FALSE` if `n` is not a perfect power.

Related Functions: `_power`, `ifactor`, `isqrt`

Details:

- Among several pairs (a, k) for which $n = a^k$, that one with minimal a is returned.

Example 1. This number is a perfect power:

```
>> numlib::ispower(1977326743)

                                        7, 11
```

This number is not a perfect power:

```
>> numlib::ispower(1977326744)

                                        FALSE
```

Changes:

- No changes.

`numlib::isquadres` – test for quadratic residues

If the integer number `a` is a quadratic residue modulo the natural number `m` `numlib::isquadres(a,m)` returns `TRUE` if the integer number `a` is a quadratic residue modulo the natural number `m`, and `FALSE` otherwise.

Call(s):

- `numlib::isquadres(a, m)`

Parameters:

`a` — an integer
`m` — a natural number coprime to `a`

Return Value: `numlib::isquadres` returns `TRUE`, `FALSE`, or the function call with its arguments evaluated.

Related Functions: `numlib::legendre`, `numlib::jacobi`, `numlib::msqrts`

Details:

- If the integer number `a` is a quadratic residue modulo the natural number `m` `numlib::isquadres(a,m)` returns `TRUE`, and if `a` is a quadratic non-residue modulo `m` `numlib::isquadres(a,m)` returns `FALSE`.
- If `a` and `m` are not coprime `numlib::isquadres(a,m)` returns an error.
- `numlib::isquadres` returns an error if the arguments evaluate to numbers which are not both of the correct type.
- `numlib::isquadres` returns the function call with its arguments evaluated if the arguments do not evaluate to numbers.

Example 1. 132132 is a quadratic residue modulo 3231227:

```
>> numlib::isquadres(132132, 3231227)

                       TRUE
```

Example 2. 222222 is a quadratic non-residue modulo 324899:

```
>> numlib::isquadres(222222,324899)

                       FALSE
```

Example 3. 37 is a quadratic residue modulo 48884:

```
>> numlib::isquadres(37,48884)

                       TRUE
```

Changes:

- No changes.

`numlib::issqr` – test for perfect squares

`numlib::issqr(a)` returns `TRUE` if `a` is the square of an integer, and `FALSE` otherwise.

Call(s):

- `numlib::issqr(a)`

Parameters:

`a` — an integer

Return Value: `numlib::issqr` returns `TRUE`, `FALSE`, or the unevaluated call.

Related Functions: `isqrt`, `numlib::ispower`, `sqrt`

Details:

- `numlib::issqr` returns the function call with evaluated argument if `a` is not a number.

Example 1. 361 is the square of 19:

```
>> numlib::issqr(361)
```

TRUE

Example 2. 362 is not a square:

```
>> numlib::issqr(362)
```

FALSE

Example 3. Negative integers are not squares:

```
>> numlib::issqr(-361)
```

FALSE

Changes:

- No changes.

`numlib::jacobi` – **Jacobi symbol**

`numlib::jacobi(a,m)` returns the Jacobi symbol `(a|m)`.

Call(s):

- `numlib::jacobi(a, m)`

Parameters:

`a` — an integer
`m` — an odd positive integer

Return Value: `numlib::jacobi(a,m)` returns a nonnegative integer, or the function call with evaluated arguments if one of the arguments is not a number.

Related Functions: `numlib::legendre`, `numlib::isquadres`

Details:

- `numlib::jacobi` returns an error if one of its arguments evaluates to a number of wrong type.

Example 1. Computing the Jacobi symbol (222222 | 304679):

```
>> numlib::jacobi(222222, 304679)

                              -1
```

Example 2. Computing the Jacobi-Symbol (222222 | 324889):

```
>> numlib::jacobi(222222, 324899)

                               1
```

Example 3. Computing the Jacobi symbol $(222222 \mid 333333)$:

```
>> numlib::jacobi(222222, 333333)

                                          0
```

Background:

- `numlib::jacobi` doesn't use `ifactor`.
- If a is an integer and m is an odd integer not coprime to a then by definition the Jacobi Symbol $(a \mid m)$ is zero.

Changes:

- No changes.

`numlib::Lambda` – von Mangoldt's function

`numlib::Lambda(m)` returns the value of von Mangoldt's function at `m`.

Call(s):

- `numlib::Lambda(m)`

Parameters:

`m` — arithmetical expression

Return Value: `numlib::Lambda` returns an arithmetical expression

Related Functions: `numlib::ispower`

Details:

- It is an error if `m` is a number but not a natural number.
- If `m` is not a number, `numlib::Lambda` returns the unevaluated function call.

Example 1. `numlib::Lambda` takes on nonzero values only for prime powers:

```
>> numlib::Lambda(49)
```

```
ln(7)
```

```
>> numlib::Lambda(48)
```

```
0
```

Example 2. `numlib::Lambda` returns the function call if its argument is not a number:

```
>> numlib::Lambda(3+n^4)
```

```
                 4
numlib::Lambda(n  + 3)
```

Background:

- The function value of `Lambda` at `m` is defined to be $\log p$ if $m = p^n$ for some prime number p and some positive integer n, and to be zero for positive integers that are not prime powers.

Changes:

- No changes.

`numlib::lambda` – Carmichael function

`numlib::lambda(n)` returns the value of the Carmichael function at `n`.

Call(s):

- `numlib::lambda(n)`

Parameters:

`n` — a natural number

Return Value: `numlib::lambda(n)` returns a natural number, or the function call with its argument evaluated.

Related Functions: `numlib::order`, `numlib::phi`

Details:

- If `m` is a natural number then `numlib::lambda(m)` returns the value of the Carmichael function in `m`, i. e., the maximal order of an element in the group of units modulo `m`.
- `numlib::lambda` returns an error if the argument evaluates to a number of wrong type. `numlib::lambda` returns the function call with its argument evaluated if `m` is not a number.

Example 1. We compute the value of the Carmichael function λ in 97:

```
>> numlib::lambda(97)

                    96
```

Example 2. We compute the value of the Carmichael function λ in 96:

```
>> numlib::lambda(96)

                    8
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::legendre` – Legendre symbol

`numlib::legendre(a, p)` returns the Legendre symbol `(a|p)`.

Call(s):

- `numlib::legendre(a, p)`

Parameters:

`a` — an integer
`p` — an odd prime

Return Value: `numlib::legendre(a,p)` returns `-1`, `0`, `1`, or the function call with evaluated arguments.

Related Functions: `numlib::jacobi`, `numlib::isquadres`

Details:

- `numlib::legendre` returns an error if one of its arguments evaluates to a number of wrong type.
- `numlib::legendre` returns the function call with evaluated arguments if at least one of its arguments does not evaluate to a number.

Example 1. Computing the Legendre symbol $(132132 \mid 3231277)$:

```
>> numlib::legendre(132132,3231227)

                         1
```

Example 2. Computing the Legendre symbol $(132131 \mid 3231277)$:

```
>> numlib::legendre(132131,3231227)

                        -1
```

Example 3. Computing the Legendre symbol $(-303 \mid 101)$:

```
>> numlib::legendre(-303,101)

                         0
```

Background:

- If p is an odd prime and if a is an integer divisible by p then by definition the Legendre symbol $(a \mid p)$ is zero.

Changes:

- No changes.

`numlib::lincongruence` – **linear congruence**

For integers `a` and `b` and a nonzero integer `m` `numlib::lincongruence(a,b,m)` returns the sorted list of all solutions $x \in \{0, 1, \ldots, |\mathtt{m}| - 1\}$ of the linear congruence $\mathtt{a} \cdot x \equiv \mathtt{b} \pmod{\mathtt{m}}$ if this congruence is solvable. Otherwise `FAIL` is returned.

Call(s):

- `numlib::lincongruence(a, b, m)`

Parameters:

`a` — an integer
`b` — an integer
`m` — a non-zero integer

Return Value: `numlib::lincongruence(a,b,m)` returns a list of nonnegative integers if `a` and `b` are integers and `m` is a non-zero integer such that the linear congruence $\mathtt{a} \cdot x \equiv \mathtt{b} \pmod{\mathtt{m}}$ is solvable.

`numlib::lincongruence(a,b,m)` returns `FAIL` if `a` and `b` are integers and `m` is a non-zero integer such that the linear congruence $\mathtt{a} \cdot x \equiv \mathtt{b} \pmod{\mathtt{m}}$ is not solvable.

`numlib::lincongruence(a,b,m)` returns the function call with its arguments evaluated if one of the arguments is a symbolic expression.

Related Functions: `numlib::ichrem`, `numlib::mroots`, `numlib::msqrts`

Details:

- `numlib::lincongruence(a,b,m)` returns an error if one of the arguments evaluates to a number of wrong type.

Example 1. A linear congruence possessing one solution:

```
>> numlib::lincongruence(7,19,23)

                              [6]
```

Example 2. A linear congruence possessing several solutions:

```
>> numlib::lincongruence(77,209,253)
```

```
[6, 29, 52, 75, 98, 121, 144, 167, 190, 213, 236]
```

Example 3. A linear congruence possessing no solutions:

```
>> numlib::lincongruence(77,208,253)
```

```
FAIL
```

Changes:

- No changes.

`numlib::mersenne` – Mersenne primes

`numlib::mersenne`() returns the list of all known Mersenne primes.

Call(s):

- `numlib::mersenne()`

Return Value: `numlib::mersenne()` returns a list of natural numbers.

Details:

- `numlib::mersenne()` returns the list of the 38 primes `p`, as known today, Jan 1, 2000, such that the `p`-th Mersenne number `2^p - 1` is prime.

Example 1. The list containing all the primes p such that the p-th Mersenne number is prime (as known today):

```
>> numlib::mersenne()
```

```
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279,

   2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937,

   21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839,

   859433, 1257787, 1398269, 2976221, 3021377, 6972593]
```

Background:

- see `http://www.utm.edu/research/primes/largest.html/`

`numlib::moebius` – Möbius function

`numlib::moebius(n)` returns the value of the Möbius function at `n`.

Call(s):

- `numlib::moebius(n)`

Parameters:

`n` — a natural number

Return Value: `numlib::moebius(n)` returns a nonnegative integer.

Related Functions: `numlib::lambda`, `numlib::phi`

Details:

- If `n` is a natural number `numlib::moebius(n)` returns the value of the Möbius function in `n`.
- If `n` is not a number, `numlib::moebius(n)` returns the function call with its argument evaluated.
- `numlib::moebius` returns an error if the argument evaluates to a number of wrong type.

Example 1. Computing the value of the Möbius function μ in 99937:

```
>> numlib::moebius(99937)

                                 0
```

Example 2. Computing the value of the Möbius function μ in 453973694165307953197296969697410619233826:

```
>> numlib::moebius(453973694165307953197296969697410619233826)

                                -1
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::mpqs` – Multi-polynomial Quadratic Sieve

`numlib::mpqs(n)` returns a proper factor of `n`, using some version of the quadratic sieve. `n` is returned if it is prime.

Call(s):

- `numlib::mpqs(n <, options>)`

Parameters:

`n`	—	integer
`options`	—	one of the options below

Options:

InteractiveInput	— prompt the user for all parameters given below
SieveArrayLimit=M	— For any polynomial f, $f(x)$ is tested for $-M \leq x \leq M$. M must be a positive integer.
Tolerance=t	— Sets an exponent t that is used to define "smoothness" of values investigated by the sieve: if the maximum of the factorbase is b, let a value pass the first part of the sieve step if it has presumably no prime divisor greater than b^t. t must be a positive real number.
Factorbase=l	— Define l to be the factor base. l must be a list of primes; they are investigated whether they divide a certain set of values of each polynomial.
MaxInFactorbase=b	— the factorbase consists of all suitable primes that are smaller than b. b must be a positive integer. This option cannot be used together with Factorbase.
NumberOfPolynomials=N	— The number of polynomials the values of which are tested for smoothness. N must be a positive integer.
LargeFactorBound=K	— Define K to be the bound below which every factor of a given value must be to make that value pass the trial-division part of the sieve step and become a sieve report. All prime numbers outside the factor base, but below that bound, are added to the factor base if they divide at least two sieve reports. K must be a positive integer.
CollectInformation	— Do not return a prime factor of n, but some information on the course of the algorithm.

Return Value: numlib::mpqs returns a positive integer dividing n, or FAIL if n is not prime, but a proper factor could not be found.

If the option *CollectInformation*has been given, a list of equations is returned; each of the equations contains some piece of information on an intermediate result in some step of the algorithm.

Related Functions: `ifactor`

Details:

- The multi-polynomial quadratic sieve is an algorithm to factor large integers without small prime factors. For small integers that can be factored within a reasonable amount of time in MuPAD, using this algorithm does not pay off. However, `numlib::mpqs` may give you some insight how the algorithm works if you set the information level yo a high value (see `setuserinfo`).

Example 1. If `n` is prime, it is returned.

```
>> numlib::mpqs(10000000019)

                        10000000019
```

Example 2. Using the default parameters, no factor is found:

```
>> n:=300000000580000000019:
   numlib::mpqs(n)

                          FAIL
```

However, using more polynomials and a larger factor base, the input can be factored:

```
>> numlib::mpqs(n,MaxInFactorbase=200,NumberOfPolynomials=30)

                        30000000001
```

Background:

- For more information about the algorithm, see Silverman, *The multi-polynomial quadratic sieve*, Math.Comp. 48 (1987), pp.329–339.

Changes:

- The option `ExtendedInformation` has been renamed to `CollectInformation`.

`numlib::mroots` – modular roots of polynomials

For a univariate polynomial P over the integers and for a natural number m the function call `numlib::mroots(P,m)` returns the sorted list of all integers $x \in \{0, 1, \ldots, m-1\}$ such that $P(x) \equiv 0 \pmod{m}$ if such integers exist; otherwise `numlib::mroots(P,m)` returns `FAIL`.

Call(s):

- `numlib::mroots(P,m)`

Parameters:

P — a univariate polynomial over the integers
m — a natural number

Return Value: `numlib::mroots` returns either a list of nonnegative integers or `FAIL`.

Related Functions: `numlib::lincongruence`, `numlib::msqrts`

Details:

- `numlib::mroots(P,m)` returns an error if P is not a univariate polynomial over the integers or m is not a natural number.

Example 1. Defining a polynomial

```
>> P := poly(3*T^7 + 2*T^2 + T - 17, [T])

                          7       2
                 poly(3 T  + 2 T  + T - 17, [T])
```

and computing its roots modulo 1751:

```
>> numlib::mroots(P, 1751)

                [221, 260, 612, 736, 1127, 1496]
```

The polynomial P does'nt have roots modulo 1994:

```
>> numlib::mroots(P, 1994)

                              FAIL
```

Background:

- `numlib::mroots` uses `factor`.

`numlib::msqrts` – modular square roots

`numlib::msqrts(a,m)` returns the list of all integers $x \in \{0, 1, \ldots, \mathtt{m} - 1\}$ such that $x^2 \equiv \mathtt{a} \bmod (\mathtt{m})$.

Call(s):

- `numlib::msqrts(a, m)`

Parameters:

`a` — an integer
`m` — a natural number relatively prime to `a`

Return Value: `numlib::msqrts(a,m)` returns a list of nonnegative integers

Related Functions: `numlib::lincongruence`, `numlib::mroots`

Details:

- `numlib::msqrts(a,m)` returns the function call with evaluated arguments if one of the arguments is not a number.
- `numlib::msqrts` returns an error if the arguments evaluate to numbers which are not both of the correct type.

Example 1. Computing the square roots of 132132 modulo 3231227:

```
>> numlib::msqrts(132132,3231227)

                        [219207, 3012020]
```

Example 2. There are no square roots of 222222 modulo 324899:

```
>> numlib::msqrts(222222,324899)

                               []
```

Example 3. Computing the square roots of 37 modulo 48884:

```
>> numlib::msqrts(37,48884)

      [383, 585, 23857, 24059, 24825, 25027, 48299, 48501]
```

Background:

- `numlib::msqrts` uses D. Shanks' algorithm RESSOL.

`numlib::numdivisors` – number of divisors of an integer

`numlib::numdivisors(n)` returns the number of positive divisors of `n`.

Call(s):

- `numlib::numdivisors(n)`

Parameters:

`n` — an integer

Return Value: `numlib::numdivisors(n)` returns a nonnegative integer.

Related Functions: `numlib::divisors`, `numlib::numprimedivisors`, `numlib::primedivisors`

Details:

- `numlib::numdivisors(0)` returns `0`.
- `numlib::numdivisors` returns the function call with evaluated argument if the argument is not a number.
- `numlib::numdivisors` returns an error if the argument evaluates to a number of wrong type.
- `numlib::numdivisors` is the same function as `numlib::tau`.

Example 1. We compute the number of positive divisors of the number 6746328388800 (one of the highly composite numbers studied by S. Ramanujan in 1915):

```
>> numlib::numdivisors(6746328388800)

                          10080
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::numprimedivisors` – number of prime factors of an integer

`numlib::numprimedivisors(n)` returns the number of prime factors of the integer `n`, counted without multiplicity.

Call(s):

- `numlib::numprimedivisors(n)`

Parameters:

`n` — an integer

Return Value: `numlib::numprimedivisors(n)` returns a nonnegative integer.

Related Functions: `numlib::primedivisors`, `numlib::numdivisors`

Details:

- `numlib::numprimedivisors(0)` returns `0`.
- `numlib::numprimedivisors` returns the function call with evaluated argument if the argument is not a number.
- `numlib::numprimedivisors` returns an error if the argument evaluates to a number of wrong type.

Example 1. We compute the number of primes dividing 6746328388800:

```
>> numlib::numprimedivisors(6746328388800)

                              9
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::Omega` – Number of prime divisors (with multiplicity)

`numlib::Omega(a)` returns, for a given positive integer `a`, the finite sum $\sum_p \alpha(p,a)$, where p runs through all primes, and $\alpha(p,a)$ denotes the highest exponent for which p^α divides `a`.

Call(s):

- `numlib::Omega(a)`

Parameters:

`a` — positive integer

Return Value: `numlib::Omega` returns a positive integer.

Related Functions: `numlib::numprimedivisors`

Details:

- `numlib::Omega` returns the function call with evaluated argument if the argument is not a number.
- `numlib::Omega` returns an error if the argument evaluates to a number of wrong type.

Example 1. In contrast to `numlib::numprimedivisors`, the prime factor 2 of 120 is counted thrice:

```
>> numlib::Omega(120)

                                  5
```

The same happens here:

```
>> numlib::Omega(8)

                                  3
```

Changes:

- No changes.

`numlib::order` – **order of a residue class**

`numlib::order(a,m)` returns the order of the residue class modulo `m` of `a` in the group of units modulo `m` if `a` and `m` are coprime.

Call(s):

- `numlib::order(a, m)`

Parameters:

`a` — an integer
`m` — a natural number

Return Value: `numlib::order(a,m)` returns a natural number if `a` is coprime to `m`, and `FAIL` if `a` is not coprime to `m`.

Related Functions: `numlib::lambda`, `numlib::phi`

Details:

- `numlib::order(a,m)` returns the function call with its arguments evaluated if `a` or `m` is not a number.
- `numlib::order` returns an error if one of the arguments evaluates to a number of wrong type.

Example 1. We compute the order of the residue class of 23 in the unit group modulo 2161:

```
>> numlib::order(23, 2161)

                        2160
```

Example 2. We compute the order of all elements in the unit group modulo 13:

```
>> map([$ 1..12],numlib::order,13)

          [1, 12, 3, 6, 4, 12, 12, 4, 3, 6, 12, 2]
```

Example 3. The residue class of 7 modulo 21 isn't a unit in the ring $\mathbb{Z}/21\mathbb{Z}$:

```
>> numlib::order(7,21)
```

```
FAIL
```

Background:

- `numlib::order` uses `ifactor` and `numlib::phi`.

Changes:

- No changes.

`numlib::phi` – Euler φ function, Euler totient function

`numlib::phi(n)` calculates the Euler φ function of `n`.

Call(s):

- `numlib::phi(n)`

Parameters:

`n` — integer not equal to zero

Return Value: `numlib::phi` returns a positive integer, if the argument evaluates to an integer unequal zero. If the argument cannot be evaluate to a number, the function call with evaluated arguments is returned .

Overloadable by: `n`

Related Functions: `numlib::invphi`

Details:

- `numlib::phi(n)` calculates the Euler φ function of the argument `n`, i.e. the number of numbers smaller than $|n|$ which are relatively prime to `n`. Cf. Example 1.
- `numlib::phi` returns an error if the argument is a number but not an integer unequal to zero.
- `numlib::phi` returns the function call with evaluated arguments if the argument is not a number. Cf. Example 2.

Example 1. `numlib::phi` works on integers unequal zero:

```
>> numlib::phi(-7), numlib::phi(10)

                                6, 4
```

Example 2. `numlib::phi` is returned as a function call with evaluated argument:

```
>> x := a: numlib::phi(x)

                          numlib::phi(a)
```

Changes:

- `numlib::phi` used to be `phi`.

`numlib::pollard` – Pollard's rho factorization algorithm

`numlib::pollard(n, m)` tries to find a factor of `n` using `m` iterations of Pollard's rho algorithm.

If `m` is missing, 10000 iterations are carried out.

Call(s):

- `numlib::pollard(n, <m>)`

Parameters:

`n,m` — positive integers

Return Value: `numlib::pollard` returns `n`, a sequence of two factors, or `FAIL`.

Related Functions: `ifactor`, `numlib::ecm`, `numlib::mpqs`

Details:

- `numlib::pollard` returns either `n` if `n` is said to be prime by `isprime`, or `g,n/g` if a factor `g` was found. If after `m` iterations still no factor has been found, `FAIL` is returned.
- Please note that the algorithm is not deterministic, thus two calls with the same arguments may give different results.

Example 1.

```
>> numlib::pollard(10000000019)
```

```
                                10000000019
```

```
>> numlib::pollard(278218430085289734806642953)
```

```
                                   FAIL
```

```
>> numlib::pollard(278218430085289734806642953,10^5)
```

```
                   3486784409, 79792266297612017
```

Changes:

- No changes.

`numlib::prevprime` – **next smaller prime**

For an integer `a` > 1 `numlib::prevprime(a)` returns the greatest prime number $\leq$ a. If `a` < 2 then `numlib::prevprime(a)` returns `FAIL`.

Call(s):

- `numlib::prevprime(a)`
- `numlib::prevprime(symb)`

Parameters:

`a` — an integer

Return Value: `numlib::prevprime(a)` returns either a natural number or `FAIL`.

Related Functions: `isprime`, `ithprime`, `nextprime`, `numlib::proveprime`

Details:

- `numlib::prevprime` returns the function call with evaluated argument if the argument is not a number.
- `numlib::prevprime` returns an error if the argument evaluates to a number which is not an integer.

Example 1. Computing the largest prime ≤ 15485865:

```
>> numlib::prevprime(15485865)
```

```
15485863
```

Example 2. There are no primes < 2:

```
>> numlib::prevprime(1)
```

```
FAIL
```

Background:

- `numlib::prevprime` uses `isprime`.

Changes:

- No changes.

`numlib::proveprime` – primality proving using elliptic curves

`numlib::proveprime(n)` tests whether `n` is a prime.

Unlike `isprime`, `numlib::proveprime` always returns a correct answer.

Call(s):

- `numlib::proveprime(n)`

Parameters:

`n` — positive integer

Return Value: `numlib::proveprime` may simply return `TRUE` or `FALSE`.
`numlib::proveprime` may return `FAIL` to indicate that the input is prime with high probability, but no proof could be found.
`numlib::proveprime` may also return a list or a sequence of lists containing a proof for the primality of `n`.

Side Effects: A particular domain `numlib::Ecpp` contains three parameters that control the algorithm:

- `Ecpp::maxit` (default 10000) is the maximal number of iterations in Pollard's rho factorization method.
- `Ecpp::maxh` (default 17) is an integer that controls the number of possibilities tried at each level of the algorithm [in technical words, it is the maximum value of the order $h(-D)$].
- `Ecpp::B` (default 1000) controls the size of the numbers to check. If `n<=Ecpp::B`, the program simply calls isprime to check if `n` is prime. In this case the program returns either `TRUE` or `FALSE`. The integer `Ecpp::B` should be at least 11, because the algorithm used does not work for n=2, 3, 5, 7 or 11.

Increasing `Ecpp::maxit` or `Ecpp::maxh` will make the algorithm more powerful, but slower.

Related Functions: `ifactor`, `isprime`, `ithprime`, `nextprime`, `numlib::prevprime`

Details:

- If `numlib::proveprime` manages to prove that `n` is a prime, it returns a primality certificate. A primality certificate is a sequence of lists of the form $[N, D, l_m, a, b, x, y, l_s]$ where 'N is a pseudo-prime, D is an integer (fundamental discriminant), l_m is a list of prime factors, a, b, x, y are integers modulo N, and l_s is another list of prime factors (subset of the factors in l_m).
- Each primality certificate produced by `numlib::proveprime` can be checked by the function `numlib::check`.
- Some information about the steps of the proof and checking can be obtained by using the function setuserinfo (see Example 3).

Example 1. Proving that 10007 is prime can be reduced to proving that 317 is prime. The primality of 317 is known because 317 is sufficiently small.

```
>> numlib::proveprime(10007)

        [10007, 43, [31, 317], 9439, 4778, 0, 5622, [317]]
```

Example 2. Normally, the primality of the input is reduced to the primality of a smaller integer, the primality of that integer is reduced to the primality of an even smaller integer, and so on.

```
>> numlib::proveprime(1048583)
```

```
 [1048583, 7, [2, 2, 2, 130817], 665765, 793371, 1, 44804,

    [130817]], [130817, 7, [2, 7, 2, 4663], 26992, 105206, 0,

    75747, [4663]], [4663, 24, [2, 5, 463], 1343, 4004, 1,

    2809, [463]]
```

`numlib::check` can be used to check the result:

```
>> numlib::check(%)
```

```
                              TRUE
```

Example 3. Use `userinfo` to get more detailed information:

```
>> setuserinfo(Any,1):
   numlib::proveprime(1048583)
```

```
 Info: found next candidate: 130817
 Info: found next candidate: 4663
 Info: found next candidate: 463

 [1048583, 7, [2, 2, 2, 130817], 665765, 793371, 1, 44804,

    [130817]], [130817, 7, [2, 7, 2, 4663], 26992, 105206, 0,

    75747, [4663]], [4663, 24, [2, 5, 463], 1343, 4004, 1,

    2809, [463]]
```

```
>> numlib::check(%)
```

```
 Info: 1048583 is prime if 130817 is prime
 Info: 130817 is prime if 4663 is prime
 Info: 4663 is prime if 463 is prime

                              TRUE
```

Background:

- This function implements the algorithm described in "Elliptic curves and primality proving", by A. O. Atkin and F. Morain, Mathematics of Computation, volume 61, number 203, 1993.

Changes:

- No changes.

`numlib::primedivisors` – prime factors of an integer

`numlib::primedivisors(n)` returns a list containing the different prime divisors of the integer `n`.

Call(s):

- `numlib::primedivisors(n)`

Parameters:

`n` — an integer

Return Value: `numlib::primedivisors(n)` returns a list of nonnegative integers.

Related Functions: `ifactor`, `isprime`, `numlib::divisors`, `numlib::numdivisors`, `numlib::numprimedivisors`, `numlib::proveprime`

Details:

- If `a` is a non-zero integer then, `numlib::primedivisors(a)` returns the sorted list of the different prime divisors of `a`.
- `numlib::primedivisors(0)` returns `[0]`.
- `numlib::primedivisors` returns the function call with evaluated argument if the argument is not a number.
- `numlib::primedivisors` returns an error if the argument evaluates to a number of wrong type.

Example 1. We compute the list of prime divisors of the number 6746328388800 (one of the highly composite numbers studied by S. Ramanujan in 1915):

```
>> numlib::primedivisors(6746328388800)

                [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::primroot` – primitive roots

`numlib::primroot(m)` returns the least positive primitive root modulo `m` if there exist primitive roots modulo `m`.

`numlib::primroot(a, m)` returns the least primitive root modulo `m` not smaller than `a` if there exist primitive roots modulo `m`.

Call(s):

- `numlib::primroot(m)`
- `numlib::primroot(a, m)`

Parameters:

`a` — an integer
`m` — a natural number

Return Value: `numlib::primroot` returns an integer or `FAIL`.

Related Functions: `numlib::order`

Details:

- `numlib::primroot` returns `FAIL` if there exist no primitive roots modulo `m`.
- `numlib::primroot` returns the function call with evaluated argument(s) if at least one argument is not a number.
- `numlib::primroot` returns an error if the arguments evaluate to numbers which are not both of the correct type.

Example 1. We compute the least positive primitive root modulo the prime number 40487:

```
>> numlib::primroot(40487)
```

```
5
```

Example 2. We compute the least primitive root modulo $40487^2 = 1639197169$:

```
>> numlib::primroot(1639197169)
```

```
10
```

Example 3. Now we compute least primitive root modulo 40487 which is $\geq$ 111111111:

```
>> numlib::primroot(111111111,40487)
```

```
111111116
```

Example 4. There are no primitive roots modulo 324013370:

```
>> numlib::primroot(324013370)
```

```
FAIL
```

Background:

- `numlib::primroot` uses `ifactor`.

Changes:

- No changes.

`numlib::sigma` – sum of divisors of an integer

`numlib::sigma(n)` returns the sum of the positive divisors of `n`.

`numlib::sigma(n, k)` returns the sum of the `k`-th powers of the positive divisors of `n`.

Call(s):

- `numlib::sigma(n)`
- `numlib::sigma(n, k)`

Parameters:

`n` — an integer
`k` — a nonnegative integer

Return Value: `numlib::sigma` returns an integer.

Related Functions: `numlib::divisors`, `numlib::numdivisors`

Details:

- `numlib::sigma(0)` returns `0`.
- `numlib::sigma` returns the function call with evaluated argument if at least one argument is not a number.
- `numlib::sigma` returns an error if one of its arguments evaluates to a number of wrong type.
- `numlib::sigma(n,0)` is the same as `numlib::numdivisors(n)` and `numlib::tau(n)`.
- `numlib::sigma(n,1)` is the same function as `numlib::sumdivisors(n)` and `numlib::sigma(n)`.

Example 1. The sum of the positive divisors of 120 is 360:

```
>> numlib::sigma(120)

                         360
```

Example 2. The sum of the fifth powers of the positive divisors of 120 is 25799815800:

```
>> numlib::sigma(120,5)

                     25799815800
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::sqrt2cfrac` – **continued fraction expansion of square roots**

`numlib::sqrt2cfrac(a)` returns the continued fraction expansion of the square root of `a` as a sequence of two lists: the first one contains the non–periodic (integer) part, the second one contains the periodic part of the expansion.

Call(s):

- `numlib::sqrt2cfrac(a)`

Parameters:

`a` — a positive integer

Return Value: If `a` is a perfect square, `numlib::sqrt2cfrac` returns a list with one entry; otherwise `numlib::sqrt2cfrac` returns a sequence of two lists, the first consisting of one integer, the second consisting of one or more integers.

Related Functions: `numlib::contfrac`

Example 1. The square root of 87 can be written as $9 + q$, where q is a rational number satisfying $q = 1/(3 + 1/(18 + q))$:

```
>> numlib::sqrt2cfrac(87)

                    [9], [3, 18]
```

Example 2. Since 81 is a perfect square, there is no periodic part in the continued fraction expansion of its square root:

```
>> numlib::sqrt2cfrac(81)

                        [9]
```

Changes:

- No changes.

`numlib::sumdivisors` – sum of divisors of an integer

`numlib::sumdivisors(n)` returns the sum of the positive divisors of the integer `n`.

Call(s):

- `numlib::sumdivisors(n)`

Parameters:

`n` — an integer

Return Value: `numlib::sumdivisors(n)` returns a nonnegative integer.

Related Functions: `numlib::sigma`, `numlib::divisors`, `numlib::numdivisors`

Details:

- `numlib::sumdivisors(0)` returns `0`.
- `numlib::sumdivisors` returns the function call with evaluated argument if the argument is not a number.
- `numlib::sumdivisors` returns an error if the argument evaluates to a number of wrong type.
- `numlib::sumdivisors(n)` is the same as `numlib::sigma(n,1)`.

Example 1. The sum of the positive divisors of 120 is 360:

```
>> numlib::sumdivisors(120)

                    360
```

Example 2. The sum of the positive divisors of -63 is 104:

```
>> numlib::sumdivisors(-63)

                    104
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::tau` – number of divisors of an integer

`numlib::tau(n)` returns the number of positive divisors of `n`.

Call(s):

- `numlib::tau(n)`

Parameters:

`n` — an integer

Return Value: `numlib::tau` returns a nonnegative integer.

Related Functions: `numlib::divisors`, `numlib::numprimedivisors`, `numlib::primedivisors`

Details:

- `numlib::tau(0)` returns `0`.
- `numlib::tau` returns the function call with evaluated argument if the argument is not a number.
- `numlib::tau` returns an error if the argument evaluates to a number of wrong type.
- `numlib::tau` is the same function as `numlib::numdivisors`.

Example 1. We compute the number of positive divisors of the number 6746328388800 (one of the highly composite numbers studied by S. Ramanujan in 1915):

```
>> numlib::tau(6746328388800)

                              10080
```

Background:

- Internally, `ifactor` is used for factoring `n`.

Changes:

- No changes.

`numlib::toAscii` – ASCII encoding of a string

`numlib::toAscii(s)` returns the list of ASCII codes of the characters in the string `s`.

Call(s):

- `numlib::toAscii(s)`

Parameters:

`s` — a string

Return Value: `numlib::toAscii(s)` returns a list of nonnegative integers.

Related Functions: `numlib::fromAscii`

Details:

- `numlib::toAscii` returns an error if its argument is not a string.

Example 1. The ASCII coding of a well-known name:

```
>> numlib::toAscii("MuPAD - Multi Processing Algebra Data Tool")

 [77, 117, 80, 65, 68, 32, 45, 32, 77, 117, 108, 116, 105, 32,

    80, 114, 111, 99, 101, 115, 115, 105, 110, 103, 32, 65,

    108, 103, 101, 98, 114, 97, 32, 68, 97, 116, 97, 32, 84,

    111, 111, 108]
```

and the ASCII coding of an empty string:

```
>> numlib::toAscii("")

                                   []
```

Changes:

- No changes.