Laplace

P!\K

COLLABORATORS			
	TITLE : Laplace		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	P!\K	April 18, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Laplace

1.1 syntax

```
                - Laplace Manual ----------------------------------------  ↩
                   Expression syntax -

6) Expression syntax

   Besides the usual evaluation  of expression, Laplace  now offers some  pro-
gramming facilities. The syntax is quite  similar to the C language,  although
there are some differences.

   Expressions are separated by semicolons  ;. Expressions can be enclosed  in
curly braces {, }, e.g. to have more than one expression in a while-loop.

   Compared to C,  Laplace works a  bit functional ,  this means almost  every
statement gives you a result, even without an assignment, result(...) or some-
thing similar. The result of an  expression block is the last expression.  You
can also abandon the execution of a block using the break()-function; the  op-
tional argument is the result of the block.
> { pi; 2+3; 3i }
>   => 3i
> { pi; break(2+3); 3i }
>   => 5
```

*
Comments

*
Expressions

*
References

*
Types

*
Loops

```
                  *
                   Procedures

                  *
                   Options

------------------------------------------------------------------ © by P!\K -
```

1.2 comments

```
- Laplace Manual -------------------------------------------------- Comments -

6.1) Comments

   Laplace uses C like  comments. Everything enclosed in  /* and */ is  simply
ignored. Nesting is not allowed, as you know it from any C compiler.

   C++ style comments are also  recognized. After // the  rest of the line  is
ignored (note that this is the  only case where Laplace distinguishes  between
spaces and line breaks).


------------------------------------------------------------------ © by P!\K -
```

1.3 expressions

```
                - Laplace Manual -----------------------------------------------  ↩
                   Expressions -

6.2) Expressions

   Expressions are entered in a usual manner, e.g.
> 1+2/3*(4-2) <return>

   Guess what's the result...

   More formally, an correct expression is:

 · an object (see
                types
                ) or
 · a reference (see
                references
                ) or
 · an built-in function (see functions) or
 · expression binary_operator expression
 · unary_operator expression
 · expression relation expression

   There are the following binary operators:
```

```
+         - add                 -         - substrac
*         - multiply            /         - divide
^         - power of            $\times$, cross  - vector product
||, or    - logical or          !||, nor  - logical nor
&&, and   - logical and         !&&, nand - logical nand
^^        - logical xor         union     - set union
intersect - set intersect       setminus  - set minus
```

Note: the vector product $\times$ is not a usual x, and can be entered by alt x .

Further more, here are the unary operators:

```
- - negative        !, not - logical not
```

A relations always results in a boolean value (TRUE or FALSE). This is the list of all relations:

```
== - equal               != - not equal
<  - less than           <= - less or equal
>  - greater than        >= - greater or equal
```


1.4 references

- Laplace Manual -- References -

6.3) References

If you want to use an result later or just define a variable, write name = expression . This creates an objects that can be referenced by it's name in later(!) entries. If you reference to an object, Laplace searches from the actual entry backwards, thereby you may define an object several times, but only the latest version is used. E.g.

```
> [1] a=1/2 <return>
> [2] 2*a <return>
>   => gets a  from [1]
> [3] a=a+1 <return>
>   => a  on the right side references to the definition in [1]
>   => and creates a new object called a  to be referenced below
> [4] a <return>
>   => gets a  from [3]
```

If you move back to [2], you still get the same result, because the new a is defined below and cannot be reference from [2].

You may overwrite built-in functions like sin () , exp () etc. and the internal constants e and pi . On the other hand, the following keywords are forbidden: if, else, while, for and all options (names starting with $).

If Laplace encounters an object name that was not defined earlier, a constant will be automatically created:

```
> f(x) = x^2 <return>
> f(a) <return>
>   => a^2
```

This is the same as

```
> f(x) = x^2 <return>
> const a <return>
> f(a) <return>
>   => a^2
```

This work only for number constants. If you require e.g. a constant vector object, you have to use the const keyword to declare it.

Generally you should always declare constants with const, otherwise you are in danger that the reference has already been defined (e.g. in a library file) and you get a type error, look at this:

```
> a=[1,2]
> ....
> [thousands of lines...]
> ....
> f(x)=x^2
> f(a)
> => error: Not defined to vectors.
```

Conventions

There are some special conventions for object names. Valid characters are A...Z, Ö, Ä, Ü, a...z, ö, ä, ü, 0...9, @, §, $, ’, , _. You may not use a number as the first character.

There are two special characters which may be used: (tilde) and _ (underscore). _ as the first character will create a line above the name, which is often used in mathematics. Inside the name _ will create an index - characters after _ will be used as the index at the bottom of the name. works similar, but creates an index at the top of the name. You can use both kinds of indeces together. E.g. _a, a_1, a 1 or everything at once _a 2_1.

Starting with version 0.3 Laplace support the usage of Greek symbols. If the name or an index matches the name of a Greek letter, then this letter is used to display the variable. Number may follow the letter name. E.g. alpha1_beta. A name in lower case represents a small Greek letter; if the first character of the name is uppercase, then you get an big Greek letter. All supported names are :

alpha	Alpha	beta	Beta	chi	Ch
delta	Delta	epsilon	Epsilon	eta	Eta
gamma	Gamma	jota	Jota	kappa	Kappa
lambda	Lambda	my	My	ny	Ny
omega	Omega	omikron	Omikron	phi	Phi
pi	Pi	psi	Psi	rho	Rho
sigma	Sigma	tau	Tau	theta	Theta
xi	Xi	ypsilon	Ypsilon	zeta	Zeta

This list can be configured (see preferences). You can add your own symbols or just use another font for greek letters.

An astonishing new feature is the possibility to use the result of any expression as the reference name or as a part of it. If you enclose an expression in reverse apostrophes `, it will be evaluated and the result replaces the expression. The only limitation is, that the result must be

- a string that contains only characters that are allowed for identifiers.
- a positive, integer
- empty. In that case there will be simply nothing inserted.

Here is an example of it:

```
> a = 2;
> B_`a` = 1;
>   => B_2 = 1
> a = ''alpha'';
> B~`a`_`2+3` = 1;
>   => B~alpha_5 = 1
> a_0 = 12; a_1 = 5; a_2 = 16;
> for ( i = 0, i < 3, i += 1 ) result( a_`i` );
>   => 12
>   => 5
>   => 16
```

As you can see, you can use an arbitrary number of subexpression in each identifier.

Definition types

There are three difference kinds of definitions:

- Variables - When you reference a variable, it’s contents will be inserted. Variable are defined using name = expression . E.g.
```
> a=3 <return>
> a+1 <return>
>   => 4
```
- Parameters - A reference to a parameter will not be evaluated, it will remain in the result. Use name := expression to create a parameter. If you want to get the final result use the xparm() function. E.g.
```
> a:=3 <return>
> 2*(a+2) <return>
>   => 2*a+4
> xparm(2*(a+2)) <return>
>   => 10
```
- Constants have no value, so they won’t be evaluate, even with the xparm() function. They are defined using the const keyword. E.g.
```
> const a <return>
> 2*(a+2) <return>
>   => 2*a+4
> xparm(2*(a+2)) <return>
>   => 2*a+4
```

You can also define functions. Just enter a argument list embedded in bracket after the object name, e.g.

```
> f(x)=x^3 <return>
> g(x,y):=sqrt(x^2+y^2) <return>
```

When referencing a function you have to specify the arguments to be inserted into the function's expression, e.g.

```
> f(3) <return>
>   => 27
> const a <return>
> f(a+1) <return>
>   => (a+1)^3
```

It is also possible to omit the arguments; if you do so, Laplace will insert the arguments as they were defined, e.g.

```
> f(x,y) = x^2 + y^2 <return>
> f'(x,y) = derive(f(x,y),x) <return>
> f'(x,y) = derive(f,x) <return>
```

Note that the last two lines are the same, because f will be expanded to f(x,y) .

Quick arithmetics

Similar to the C language Laplace knows some abbreviations for often used expression. Instead of writing

```
> a = a + 2;
```

you should use

```
> a += 2;
```

This is faster than the first form, because Laplace can make some internal optimizations. This works for the following binary operators:

```
  x += a   <=>   x = x + a
  x -= a   <=>   x = x - a
  x *= a   <=>   x = x * a
  x /= a   <=>   x = x / a
 x ||= a   <=>   x = x || a
 x or= a   <=>   x = x || a
 x &&= a   <=>   x = x && a
x and= a   <=>   x = x && a
```


1.5 types

- Laplace Manual ↩
--- Types -

6.4) Types

Each expression has a type. Laplace supports a complex hierarchy of types:

·
invalid
·
identifier
·
object
·
number
·
real
·
integer
·
posinteger
·
neginteger
·
complex
·
tensor
·
vector
·
matrix
·
array
·
boolean
·
string
·
equation
·
interval

You can query the type of an object with the typeof() function. ↩ It returns a string with the object's type. To check, if an object is of a given type or a subclass of that type, use issubtype(), which returns boolean. The vector [1,2] for example has a type of vector, which is a subclass of tensor and object.

```
> v = [1,2]
> typeof(v)
>    => "vector"
> issubtype(v, "vector")
>    => TRUE
> issubtype(v, "tensor")
>    => TRUE
> issubtype(v, "object")
>    => TRUE
> issubtype(v, "array")
>    => FALSE
```

A variable declaration is a type name followed by the variable's name. Additionally to the type name, you can set some special attributes:

- const - A constant object of the given type is created. You cannot assign a value to a constant.

- numeric - not implemented.

Laplace recognizes a declaration, when it encounters a type name or one of the attributes. If you use an attribute, you may omit the type name, which is number by default. Without attributes, you must give the type name, otherwise the declaration won't be recognized.

```
> number a;
> a = 1;
> const vector a;
> const a;
```


1.6 type-invalid

- Laplace Manual -- type 'invalid' -

Type invalid

invalid is not a real type, but is assigned to every object that doesn't have a valid type. For example det(2) has the type invalid, because you can calculate the determinant only for matrices.

1.7 type-identifier

- Laplace Manual --- type 'identifier' -

Type identifier

Currently it doesn't make sense to declare an object of type identifier. But you might get an error message telling you, that a function expected an object of type identifier as an argument, e.g. the derive() function, if you only give one argument. This means that you have to pass the name of a previously declared object as the argument.

```
> derive(x^2)
>    => ERROR
> f(x) = x^2
> derive(f)
```


1.8 type-object

```
- Laplace Manual ---------------------------------------------- type 'object' -

Type object

   This is the base type for all  usually used types. An expression is  either
invalid or has a sub type of object.


------------------------------------------------------------------ © by P!\K -
```

1.9 type-number

```
- Laplace Manual ---------------------------------------------- type 'number' -

Type number

   These are just numbers as we all know them (or do we know them all?!).

   Numbers can be entered in exponential style, which is
             {+/-/<nothing>}ddddd[.ddddd][e{+/-/<nothing>}dddd].
   Ufff, but  it's  quite  easy;-)  to  enter  e.g.  6.626  *  10^(-34)  write
6.626e-34. The  (whole)  number  after  e is  just  the  exponent.  Don't  use
brackets, if the exponent is negative!

   Laplace supports complex numbers. To  enter a complex number, simply  write
something  like  1  +  2i  .   But  they  are  not  fully  implemented,   e.g.
error-distribution doesn't work correctly when you have complex values in your
expression and not all functions work with complex arguments.

   As long as you enter only whole numbers, Laplace tries to use fractions, so
enter 1/2 instead of 0.5 (in this simple case Laplace converts 0.5 to 1/2  it-
self, but this may not  always work). This way you  can enter things like  1/7
exactly, and you won't get result which are almost exact zero, when it  should
be exact ;-)

   A fantastic  new feature  of  Laplace is  the  direct support  of  Gaussian
error-distribution. Now it is possible to enter values with their standard er-
ror, do what ever calculations  you have to do, and  then get the result  with
the correct error value! This is very useful, if you are calculating with  re-
sult of some measurements, like I did during my physical practical.

   For example, you want  to measure the earth  acceleration by measuring  the
falling time of a ball. You measured the falling height to 1.119 meters with a
precision of one millimeter and the time  to 0.50 seconds with a precision  of
1/50 second. Now simply enter:
> t=0.50\ensuremath{\pm}0.02
> h=1.119\ensuremath{\pm}0.001
> g=2*s/h^2

   and that's all!! (Next time you should do it more precisely..)

   The \ensuremath{\pm}  can be  entered  by alt-y  (on the  German  keyboard) or ↩
        alt-z  (on
others).
```

But you should be aware, that in some cases you might get wrong result, because 10\ensuremath{\pm}1*10\ensuremath{\pm}1 is not the same as 10\ensuremath{\pm}1 ^ 2 . So, if the same value is used at different places in a formula, it is evaluated as if they had independent error ranges!

The best way to avoid this, is declaring all values with errors as parameters, construct your (possibly big) formula and simplify it, so that every parameter occurs only a single time, e.g.:

```
> x := 10\ensuremath{\pm}1
> y := 12\ensuremath{\pm}2
> a = 3*x^2*y^3
> b = 5*x^3*y^2
> result = a/b
> xparm(result)
```

If you on the other hand have something like this:

```
> x := 10\ensuremath{\pm}1
> result = x/(x+1)
```

you'll get a wrong error value and there is no way to avoid it, sorry.

1.10 type-real

- Laplace Manual --- type 'real' -

Type real

real is a sub-type of number, specifying numbers without imaginary part, e.g. 5.34

1.11 type-integer

- Laplace Manual -- type 'integer' -

Type integer

integer is a sub-type of real, specifying real, whole numbers, e.g. -3

1.12 type-posinteger

```
- Laplace Manual ----------------------------------------- type 'posinteger' -

Type posinteger

   posinteger is a sub-type of integer, specifying real, whole numbers  bigger
than zero, e.g. 7


------------------------------------------------------------------- © by P!\K -
```

1.13 type-neginteger

```
- Laplace Manual ----------------------------------------- type 'neginteger' -

Type neginteger

   neginteger is a sub-type of integer, specifying real, whole numbers smaller
than zero, e.g. -9


------------------------------------------------------------------- © by P!\K -
```

1.14 type-complex

```
- Laplace Manual --------------------------------------------- type 'complex' -

Type complex

   complex is a sub-type of number, specifying numbers without real part, e.g.
3.5i


------------------------------------------------------------------- © by P!\K -
```

1.15 type-tensor

```
- Laplace Manual ---------------------------------------------- type 'tensor' -

Type tensor

   tensor is the base type for vectors and matrices. A tensor is a  orthogonal
arrangement of numbers of n   dimensions. n = 1   is a linear arrangement and
therefore a vector. n = 2  specifies  a square arragement of numbers and is a
matrix.

   To access single members  of a tensor, write  the index in square  brackets
right after the tensor T[a,b,...] . The index must have exactly n  components,
and numbering of the members starts at 1 .
```

You can also access sub-tensors by using intervals in the index. T[2..4] constructs a one dimensinal tensor from the one dimensional tensor T with three members 2, 3, 4 . T[2..4,2] is again a one dimensional tensor, this time constructed from a two dimension tensor T taking the three members [2,2], [3,2], [4,2] . T[2..4,1..2] creates a two dimensional tensor with the members [2,1], [3,1], [4,1], [2,2], [3,2], [4,2] .

1.16 type-vector

- Laplace Manual --- type 'vector' -

Type vector

A vector is a tuple of numbers. To create the vector (2,4,1) enter [2,4,1]. Any expression can be (of course) a component of a vector.

You can add and multiply (scalar) two vectors, or multiply a number or matrix with a vector (in this order, not vector*number !). You get the vector product of two 3-dimensional vector by typing vector $\times$ vector. The $\times$ ↩
is not an
ordinary x, but can be entered by alt-x (on the German keyboard and probably all others, too).

To access the n -th member of a vector v (which can be any expression that evaluates into a vector), append the index in square brackets v[n] . The first member has the index one.

You can also use an interval a..b as the index, extracting a vector with the members a to b . -infty as the lower limit is replaced by 1 , while infty as the upper limit is replaced by the dimension of the vector. This way v[2..] will replicate the vector v without the first element.

1.17 type-matrix

- Laplace Manual --- type 'matrix' -

Type matrix

The matrix with the row vectors (1,2,3), (4,5,6) and (7,8,9) can be created with [1,2,3;4,5,6;7,8,9]. If you have column vector in mind, use [1,4,7;2,5,8;3,6,9]! instead (append an exclamation mark).

You can add and multiply two matrices, or multiply a number and a matrix (in this order, not matrix*number !).

To access the element in the n -th row and m -th column of a matrix M (which can be any expression that evaluates into a matrix), append the indeces in square brackets M[n,m] . The first row/column has the index one.

By using an interval in the index, you can access row or column vectors of the matrix: M[n,..] extracts the n -th row into a vector, while M[..,m] composes a vector out of the m -th column. The resulting vectors are always column vectors.

With two intervals you can extract a sub-matrix, e.g. to get the matrix N from a matrix M , leaving out the first row and the first column, write N = M[2..,2..] .

1.18 type-array

- Laplace Manual -- type 'array' -

Type array

To create a array use the array() command.

A array is a collection of objects of the any type. Members can occur multiple times in an array.

If you calculate with array, all operations (except those affecting the array directly) are applied to all members of the array, e.g.

```
> A=array(1,2,3) <return>
> A+2 <return>
>    => {3,4,5}
```

The function count() returns the number of members in an array. To access the n -th member of an array A (which can be any expression, which evaluates to an array), write the index in square brackets right after the array: A[n] . The first member has the index 1. If you use an interval a..b as the index, this creates a new array with the members, taking the members a to b from A .

```
> A=array(5,7,12,7) <return>
> A[1] <return>
>    => 5
> (2*A)[2] <return>
>    => 14
> A[2..3] <return>
>    => [7,12]
```


1.19 type-boolean

```
- Laplace Manual -------------------------------------------- type 'boolean' -

Type boolean

   A boolean object can only contain the two values TRUE and FALSE.

   Possible operation for boolean objects are: && (and), || (or), !&&  (nand),
!|| (nor), ^^ (xor) and ! (not).

> a=TRUE
> b=FALSE
> c=TRUE
> a && (b || !c)


------------------------------------------------------------------ © by P!\K -
```

1.20 type-string

```
- Laplace Manual --------------------------------------------- type 'string' -

Type string

   Strings are surrounded by double quotes ".

   Similar to strings in C, you can use the backslash @{UB}\ as an escape  ↩
      character
to insert special characters to a string:
```

- \" - add a double quote (otherwise you'd get problems with the surrounding quotes!).
- \n - add a new line (ASCII code 10).
- \\ - the backslash itself.

```
   To concatenate two strings, simply add  them:
> " Hi " + " there!"
>   => " Hi there!"


------------------------------------------------------------------ © by P!\K -
```

1.21 type-equation

```
- Laplace Manual ------------------------------------------- type 'equation' -

Type equation

   An equation is created by the =? operator.
> E= 2*x+4 =? 0 <return>

   To access the left or  right hand side of a  given equation, use the  func-
```

tions lhs() or rhs().

```
> lhs(E) <return>
>   => 2*x+4
> rhs(E) <return>
>   => 0
```

You can of course calculate with equations. The operations will be applied on both side of the equation.

```
> E= 2*x+4 =? 0
> E= E-4
> E= E/2
```

You can also apply functions line sin(), exp()... on equations, e.g.

```
> E= ln(x) =? 12*e <return>
> E=exp(E) <return>
```


1.22 type-range

- Laplace Manual --- type 'interval' -

Type interval

An interval is a range of values, starting at a , ending at b , and is defined by a..b . a and b can usually be any number, but there are restrictions depending on the context the interval is used for. In most cases, a and b have to be real numbers with a < b.

You may omit the lower and/or upper limit of the range, which is than replaced by -infty respectively infty .

1.23 loops

- Laplace Manual -- Loops -

6.5) Loops

In contrast to most other statements, the loops won't return a result.

You can always interrupt the loop using the break() function.

the while-statement

The statement or the block of statements following the while keyword is executed as long as the argument evaluates to the boolean TRUE.

```
> a = 0; b = 0;
> while ( a < 10 )
```

```
> {
>   a += 1;
>   b += irandom(10);
> }
```

the for-statement

The for-loop is very close to the C for-loop: the first argument is evaluated before the loop is started, the second on is a condition that is checked at the beginning of the loop and the third argument is evaluated at the end of the loop.

```
> b = 0;
> for ( a = 0; a < 10; a += 1 )
>   b += irandom(10);
```


1.24 conditionals

- Laplace Manual -- Conditionals -

6.6) Conditionals

The if-condition works exactly the way you expect it to do: if the condition is TRUE the first statement or statement block is executed, otherwise the else-statement is executed, if there is one.

```
> if ( a > 5 )
> {
>   b += 10;
>   c -= sin(b);
> }
> else
>   b -= 10;
```


1.25 procedures

- Laplace Manual -- Procedures -

6.7) Procedures

The ability to create procedures is the most powerful invention of V0.8.

The syntax for creating a procedure is now very similar to the syntax of the C language: The return type of the procedure, followed by an identifier with optional arguments and then a statement block:

```
> number foobar(number a)
> {
>     if ( a > 0 )
```

```
>         return(-1);
>     else
>         return(1);
> }
```

As you can see, the return()-functions, aborts the procedure and returns the optional argument as the result of the procedure. If you omit the return() command, and the procedure reaches the end of the statement block, the result of the last expression is returned.

1.26 options

- Laplace Manual --- Options -

6.8) Options

There are some options to influence to calculation or presentation of expressions. They can be used like any other reference, but they all start with a $ (dollar sign). You can assign a new value to it, just by typing e.g.

```
> $dispprec = 6
> a = 3
> $dispprec = 2*a
```

or you can query the current value:

```
> a = $dispprec
>   => 6
```

As always, Laplace searches backwards until it finds an option, this way setting an option will not influence the lines above. If no explicit assignment has been done, the default value is used.

These are the keyword:

-

- $convprec - Default: 14 - Possible value: 1..20. Laplace tries to convert floatpoints into fractions whenever possible. But the floatpoint routines are not too exact, so there might be an small difference to the correct value (e.g. 1.2000000000000001 when it should be 1.2). If the difference is smaller than the specified precision (precision 12 means 10^(-12)), Laplace assumes the value as a fraction and converts it (in this case to 6/5). This is not applied to value near zero, so 1e-30 won't be converted to 0.

- $dispexp - Default: 6 - Possible value: 1..12. Select number of digits to be displayed before switching to exponential display.

- $dispprec - Default: 6 - Possible value: 1..15. Select maximum number of decimal digits to be displayed.

- $iref - Default: TRUE - Possible value: TRUE, FALSE. The usage of the single letter i as a reference name can be somehow confusing. If you are working with complex numbers, you probably mean i as sqrt(-1), but in other cases you use i as an index variable or something like that.

If you set $iref to TRUE, i always stands for an reference, and you have to enter 1i to get the complex number. Otherwise i means 1i and there is no way to enter a reference called i. (Actually there is one: `"i"`, but this is a quite difficult way to enter a single letter...)

- $simplify - Default: TRUE - Possible value: TRUE, FALSE. Enable/Disable simplification pass. Without simplification, the calculation will be faster, but the result might look quite ugly.

- $transpose - Default: FALSE - Possible value: TRUE, FALSE. A matrix is usually entered ordered by rows, similar to programs like MAPLE or MATLAB. If you prefer column vectors, set this option to TRUE.

  This option can be overwritten be entering [..]~ (always use row-order) or [..]! (always use column-order).

- $useerror - Default: TRUE - Possible value: TRUE, FALSE. If set to FALSE, Laplace will not use and display error ranges.

- $usefloat - Default: FALSE - Possible value: TRUE, FALSE. If set to TRUE, Laplace will always use floatpoints and never converts them to fractions.

  For example:

```
> $usefloat = FALSE
> sin(2)
>   => sin(2)
> $usefloat = TRUE
> $dispprec = 5
> sin(2)
>   => 0.9093
> $dispprec = 8
> sin(2)
>   => 0.90929743
> a=12000.3
> $dispexp = 3
> a
>   => 1.20003*10^ 4
> $dispexp = 6
> a
>   => 12000.3
```