Computer Algebra using Maple
Part III: Data structures;
   Procedures in more detail; Data output

Winfried Auzinger, Kevin Sturm (SS 2019)
1 Tables

> restart:

A table is an associative array, i.e., indices may be arbitrary objects, not only integers.

Example:

> day := table([m="Monday", t="Tuesday", w="Wednesday"]);
> is(day, table);
true
> day[m];
"Monday"

Default indices are 1,2,3,...:

> month:=table(["January", "February", "March"]);
> month[2];
"February"
> indices(month); # all indices
[1], [2], [3]
> entries(month); # all entries
["January"], ["February"], ["March"]

In contrast to lists, tables are organized dynamically. Further entries can be added in a simple way:

> month[5]:="May";
month := [1 = "January", 2 = "February", 3 = "March", 4 = "April", 5 = "May"]
> month[4]:="April";
> month; # specifying name of table does not evaluate it
month
> eval(month); # evaluate (show contents of table)
table([1 = "January", 2 = "February", 3 = "March", 4 = "April", 5 = "May"])
> print(month);
table([1 = "January", 2 = "February", 3 = "March", 4 = "April", 5 = "May"])

Specifying the name of the table does not output its contents, a principle called last name evaluation.

This is reasonable for objects of such a type, which may be large.
2 Arrays

An **Array** is a **static**, multidimensional ‘rectangular’ data structure; think of a vector (1D) or a matrix (2D).

**Explicit initialization** (storage allocation) is therefore required (default values: 0).

Example:

```plaintext
> A := Array(1..5);
whattype(A);

A := [ 0 0 0 0 0 ]

Array

> A[1]:=x; A[2]:=x->x^2;

A_1 := x

A_2 := x \mapsto x^2

> print(A);

[ x x \mapsto x^2 0 0 0 ]
```

Defaults for options:

- **datatype** = anything (any valid type)
- **storage** = rectangular ('normal' storage; other options exist!)
- **order** = Fortran_order (order in which elements are internally stored):
  Example: (1,1),(2,1),(3,1), ... (1,2),(2,2),(3,2) ('colum-major order', as in Matlab)

Normal use: for storing **numerical data**. Datatype may be specified (e.g. integer[4], float[8])

**float[8]** is standard double precision.

Default initialization with 0.

Example 2-dimensional (also with more general index range!)

```plaintext
> A := Array(0..2,0..3,datatype=float[8]);
A := Array(0..2,0..3, Ø, datatype=float[8])

> A[0,0] := 1;

A_{0,0} := 1

> A[0,0], A[0,1];

1., 0.

> print(A); # does not look not so nice

Array(0..2,0..3, {(0,0) = 1.}, datatype=float[8])
```
Example: with explicit initialization using listlist:

```maple
> A := Array([[1,2,3,4],[5,6,7,8]]);

A :=

1  2  3  4  
5  6  7  8

Example 3-dimensional (more general index range):

```maple
> A := Array(-3..3,-3..3,-3..3,datatype=integer[4]);

A :=

Data Type: integer
Storage: rectangular
Order: Fortran_order

> A[3,3,3];

0
```

Alternative: Generate entries using indexing function:

```maple
> A := Array(0..3,0..4,(i,j)->i*j);
A := Array(0..3,0..4, ((1, 1) = 1, (1, 2) = 2, (1, 3) = 3, (1, 4) = 4, (2, 1) = 2, (2, 2) = 4, (2, 3) = 6, (2, 4) = 8, (3, 1) = 3, (3, 2) = 6, (3, 3) = 9, (3, 4) = 12))
```

Test membership (this works for general data structures):

```maple
> member(1,A), member(-1,A);
true, false
```

Remarks:

-- Main advantage of arrays: General index range. (Default: start from 1.)

-- Arrays are implemented using the general concept of an rtable.
  rtable (‘rectangular table’) is a general, versatile data structure.
  In particular, rtables support special storage modes, e.g. symmetric, triangular, sparse, empty...
  (think of matrices!), for efficient and automatic handling of special cases.

-- Storage mode empty: Used, e.g., for ZeroMatrix, IdentityMatrix (see Section 3) -
  these are represented as symbols, read-only, nothing is stored.

-- For standard cases vector and matrix, where index starting with 1,
  we better use the related, more special data structures Vector, Matrix
  (also derived from rtable).
3 Vectors and Matrices

These may be considered as special types of Arrays. They can be used for symbolic, numerical, or mixed data.

As for Arrays, static initialization with fixed dimension is required, typically with the 'constructors' Vector and Matrix (these have the same names as the corresponding data types).

Vector is one-dimensional, index starts with 1.

> v := Vector(5);
  whattype(v);

Vector column

> Vector[column]([3$5]); # same as Vector(...)

f := i -> i^2;
> v := Vector(5, f); # initialization with indexing function

Vector element, subvector:

> v[1], v[2];   # first, second
v[-1], v[-2]; # last, second to last
v[1..3]; # subvector

1, 4
25, 16

[1
 4
9]

> Vector[row](4); # a row Vector

[0 0 0 0]

> whattype(%);

Vector_row

Inner product of vectors: Use dot: .

> x:=Vector[row]([1,2,3]);
y:=Vector([xi,eta,zeta]);
x.y

x := [ 1 2 3 ]
y := [ ξ ]
     [ η ]
     [ ζ ]
ξ + 2 η + 3 ζ

The dot operator . generally stands for Vector-Vector and Vector-Matrix multiplication. For the case of an inner product of two Vectors, it also accepts two column Vectors as arguments, i.e., transposing the first argument is not necessary (a pragmatic implementation):

> x:=Vector([1,2,3]);
y:=Vector([xi,eta,zeta]);
x.y

x := [ 1 ]
    [ 2 ]
    [ 3 ]
y := [ ξ ]
     [ η ]
     [ ζ ]
ξ + 2 η + 3 ζ

Matrix is two-dimensional, indices start with 1.
M := Matrix(3,4);
whattype(M);

M :=
\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Matrix

Matrix([[a,2,c,4],[1,b,3,d]]); # specify row-wise

\[
\begin{bmatrix}
a & 2 & c & 4 \\
1 & b & 3 & d \\
\end{bmatrix}
\]

M := Matrix(5,5,(i,j)->1/(i+j-1)); # generate Hilbert Matrix using indexing function

\[
M :=
\begin{bmatrix}
1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\
\frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\
\frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\
\frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\
\frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\
\end{bmatrix}
\]

Matrix element, submatrix:

> M[1,1], M[1,2], M[1,-1]; # first two elements and last element in first row

M[1,2], M[2,2], M[-1,2]; # first two elements and last element in second column

M[-1,-1], M[-2,-2]; # last and second to last diagonal element

1, \frac{1}{2}, \frac{1}{5}

\frac{1}{2}, \frac{1}{3}, \frac{1}{6}

\frac{1}{9}, \frac{1}{7}

M[1..2,3..4]; # submatrix

\[
\begin{bmatrix}
\frac{1}{3} & \frac{1}{4} \\
\frac{1}{4} & \frac{1}{5} \\
\end{bmatrix}
\]

Row, column of a matrix ( .. is the analog to : in MATLAB):
Basic arithmetic with vectors and matrices:
for multiplication use \( \cdot \) (not \(*\)):

\[
\begin{bmatrix}
1 & 1 & 1 \\
2 & 3 & 4 \\
3 & 4 & 5 \\
4 & 5 & 6
\end{bmatrix}, \begin{bmatrix}
\frac{1}{2} \\
\frac{1}{3} \\
\frac{1}{4} \\
\frac{1}{5} \\
\frac{1}{6}
\end{bmatrix}
\]

\[a := 'a';\]
\[A, x := \text{Matrix}(3, 3, (i, j) \rightarrow i+j), \text{Vector}(3, i \rightarrow i^2);\]
\[2 \times x, \quad \text{# scalar} \times \text{Vector}\]
\[3 \times A, \quad \text{# scalar} \times \text{Matrix}\]
\[x + x, \quad \text{# Vector} + \text{Vector}\]
\[A + A; \quad \text{# Matrix} + \text{Matrix}\]
\[A \cdot x, \quad \text{# Matrix} \cdot \text{Vector}\]
\[A \cdot A, \quad \text{# Matrix} \cdot \text{Matrix}\]
\[A^3; \quad \text{# Matrix power}\]

The following commands (and many others) are part of the **LinearAlgebra** package
(to be discussed in more detail later on)

\[\text{LinearAlgebra[Dimension]}(M);\]
\[\text{5, 5}\]
\[\text{LinearAlgebra[Row]}(M, 1); \quad \text{# same as M[1,..]}\]
> LinearAlgebra[Column](M,2); # same as M[..,2]

\[
\begin{bmatrix}
1 & 1 & \frac{1}{3} & \frac{1}{4} & \frac{1}{5}
\end{bmatrix}
\]

Or: activate complete package:

> with(LinearAlgebra):

Example: Vandermonde matrix

> n := 20:
> V := Matrix(n,n,(i,j)->xi[i]^j):
> V;

\[
\begin{bmatrix}
1 & 2 & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6}
\end{bmatrix}
\]

Special matrices:

> LinearAlgebra[ZeroMatrix](3);
print(LinearAlgebra[ZeroMatrix](100));

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

> print(LinearAlgebra[ZeroMatrix](100));

100 x 100 Matrix
Data Type: anything
Storage: empty
Order: Fortran_order

> LinearAlgebra[IdentityMatrix](3);

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

> print(LinearAlgebra[IdentityMatrix](3));

100 x 100 Matrix
Data Type: anything
Storage: empty
Order: Fortran_order

> print(LinearAlgebra[IdentityMatrix](100));

100 x 100 Matrix
Data Type: anything
Storage: empty
Order: Fortran_order

> print(LinearAlgebra[IdentityMatrix](100));
4 Storage management

Consider sequence of statements:
(Note that this is usually not good style!)

\[
\begin{align*}
> \ a & := 1; & a := 1 \\
> \ b & := a; & \text{# current value of } a \text{ is copied, gives value for } b \\
> \ a & := 2; & a := 2 \\
> \ b; & \quad \# \text{ Change of } a \text{ does not affect } b. \\
\end{align*}
\]

But: For usually larger objects like [r]tables, Arrays, Vectors, Matrices, assigning via := does not copy the object, but sets a pointer to the original object:

\[
\begin{align*}
> \ A & := \text{Matrix}(\[[1,2], \\
& \quad [3,4]]) \; ; \\
A & := \begin{bmatrix} 1 & 2 \\
& 3 & 4 \end{bmatrix} \\
> \ B & := A; \\
B & := \begin{bmatrix} 1 & 2 \\
& 3 & 4 \end{bmatrix} \\
> A[1,2] & := 1000; \ 'B'=B; \ # \text{ now also } B \text{ has changed!} \\
A_{1,2} & := 1000 \\
B & := \begin{bmatrix} 1 & 1000 \\
& 3 & 4 \end{bmatrix}
\end{align*}
\]

To copy such objects, use copy:

\[
\begin{align*}
> \ A; \\
> \ B & := \text{copy}(A) ; \\
B & := \begin{bmatrix} 1 & 1000 \\
& 3 & 4 \end{bmatrix}
\end{align*}
\]
A[1,2]:=100000000; # now this does not affect the independent copy B

A := 'A'; 'B' = B;

A_{1,2} := 100000000

A := A

B = \begin{bmatrix}
1 & 1000 \\
3 & 4
\end{bmatrix}
5 Stack, queue, heap

Use of stacks, queues and heaps is supported (predefined functionality, implemented in so-called modules).

Example: use of the stack data structure

```plaintext
> S := stack[new]();
  stack[depth](S);  
```

```plaintext
S := table([0 = 0])  
0
```

```plaintext
> for i from 1 to 5 do
  stack[push](i,S) 
end do;
```

```plaintext
1
2
3
4
5
```

```plaintext
> stack[depth](S); 
```

```plaintext
5
```

```plaintext
> for i from 1 to stack[depth](S) do
  stack[pop](S) 
end do;
```

```plaintext
5
4
3
2
1
```

```plaintext
> stack[empty](S); 
```

```plaintext
true
```
6 General form of a procedure; examples

A procedure template:

```maple
> p := proc(FORMAL_PARAMETERS)
    description "Text";
    option OPTIONS;
    local LOCAL_VARIABLES;
    global GLOBAL_VARIABLES;
    # procedure body
    return ...
    # result
    # or:
    error "errormessage"; # error exit
end proc:
```

Note that the result of a procedure returned using `return` may be any valid Maple object or sequence of Maple objects.

Example: search next prime number after given number n

For demo purpose, this also calls another procedure (advance).

```maple
> search_prime := proc(n,nmax)
    description "Search prime number >= n";
    option trace; # activate verbose behavior, see output below
    local i:=n;   # initialize local variable
    if is(i,even) then
        i:=i+1
    end if;
    do # general loop
        if isprime(i) then
            return i  # normal return
        end if;
        i:=advance(i);
        if i>nmax then
            error "Limit exceeded" # error exit with your own error message
        end if;
    end do;
end proc:
```

```maple
> search_prime( n, nmax )
```

```maple
> Describe(search_prime); # show description
# Search prime number >= n
search_prime( n, nmax )
```
By the way: this is also predefined in Maple:

> nextprime(74);

In the next example, we use a general loop `do ... end do`, with `break` / `next`, and the random number generator `rand`

Procedure generates lottery quick tip, but all numbers are required to be divisible by q.

This may result in an infinite loop, especially if q is too large.
Stop with error after nmax rejections of a drawn number.

If second parameter q is not specified (this is admitted), q:=1 is assumed by default:
Default applies if second parameter q is not passed.
(We could also stop immediately if 1<=q<=6 is not satisfied.)
drawn := drawn union {number}:
if numelems(drawn)=6 then
  return convert(drawn,list)
end if:
end if
done:
error "# of maximal tries exceeded!"
end proc:

> lotto(100); 
[1, 13, 17, 21, 24, 40] 
> lotto(100,2); 
[8, 10, 26, 28, 36, 38] 
> lotto(100,3); 
[3, 21, 27, 30, 33, 42] 
> lotto(1000,10); 
Error, (in lotto) # of maximal tries exceeded!

Two further examples: use of option remember. This is especially useful for recursions where multiple calls occur. A so-called remember table is generated, where results already obtained are automatically stored and retrieved.

We measure the performance using the CPU clock (see section 9 below).

> fibonacci := proc(n)
# without option remember
  if n<=2 then
    return 1
  else
    return fibonacci(n-1)+fibonacci(n-2)
  end if
end proc:

> start:=time():
  fibonacci(30);
  time()-start;

832040
0.625

> fibonacci := proc(n)
option remember;
  if n<=2 then
    return 1
  else
    return fibonacci(n-1)+fibonacci(n-2)
  end if
end proc:

> start:=time():
  fibonacci(30);
  time()-start;

832040
0.0

Naturally, this much more simple to realize using an explicit loop.
Example for a 'full' recursion (for each n, all lower values are required):

```plaintext
> p := proc(n)
    # without option remember
    local i;
    if n=1 then
        return 1
    else
        return add(n*(n-i)*p(i),i=1..n-1)
    end if
end proc:
> start:=time():
p(22);
time()-start;

76597888641145621209482
1.531

> p := proc(n)
    option remember; # with option remember
    local i;
    if n=1 then
        return 1
    else
        return add(n*(n-i)*p(i),i=1..n-1)
    end if
end proc:
> start:=time():
p(22);
time()-start;

76597888641145621209482
0.
```

In this example, the storage requirement for remember table is proportional to n.

Converting this to an explicit loop also requires allocating storage:

```plaintext
> p := proc(n)
    local i,j,values:=[0$n]:
    values[1]:=1:
    for i from 2 to n do
        print(seq(values[j],j=1..i-1));
        values[i]:=add(i*(i-j)*values[j],j=1..i-1):
    end do:
    return values[n]
end proc:
> start:=time():
p(22);
time()-start;

1
1, 2
1, 2, 12
1, 2, 12, 76
```
Note: Recursive formulation of algorithms is often elegant and convenient, but this is not always of optimal efficiency. In particular, it requires a certain amount of storage, because the data for the recursively pending calls are put on the internal stack.

Algorithmically, a recursive procedure performs an loop 'in disguise'.

```plaintext
Note: Recursive formulation of algorithms is often elegant and convenient, but this is not always of optimal efficiency. In particular, it requires a certain amount of storage, because the data for the recursively pending calls are put on the internal stack.

Algorithmically, a recursive procedure performs an loop 'in disguise'.
```
7 Argument evaluation in procedure calls

RULE: 'Call by value', i.e.,

On procedure call, arguments are first evaluated and then passed to the procedure.

Example:

```
> p := proc(a)
    print(a)
    end proc:
> p(a);
2
> a:=1;
   a := 1
> p(a);
1
> p('a');
a
> Do not try to generate 'hidden' output by redefining arguments within a procedure. This may work to some extent but is not reliable.

For output, use return to return the desired values or objects.
8 Evaluation rules for variables

restart:

Usually, variables defined along 'chains' are fully evaluated.

Example: (Note: This is not good style!)

```plaintext
> c:=b;
  c := b
> b:=a;
  b := a
> c;
  a
```

Within a procedure, this does not work:

```plaintext
> p:=proc()
  local a,b,c:
  c:=b:
  b:=a:
  c;
  end proc:
> p();
  b
```

This rule is called level-1-evaluation
(implemented for efficiency reasons).

On the other hand, the following version uses a chain of
level-1-evaluation and works as expected:

```plaintext
> p:=proc()
  local a,b,c:
  b:=a:
  c:=b:
  c;
  end proc:
> p();
  a
```
9 CPU stop watch

Using the cpu timer `time()` you can measure computing times, e.g., for a block of statements or function or procedure call:

```
> start_time := time(); # CPU seconds consumed in this session
int(exp(-x^n),x); # do something
time_used := time()-start_time;
```

```
\[
\begin{align*}
\frac{1}{n} & \left( n^2 x^{-n+1} (n x^n + n + 1) \left( x^n \right)^{-n+1} \frac{x^n}{2n} e^{-\frac{x^n}{2}} \text{WhittakerM}\left( \frac{1}{n} - \frac{n+1}{2n}, \frac{n+1}{2n} + \frac{1}{2}, x^n \right) \\
& \quad + n x^{-n+1}(n+1) \left( x^n \right)^{-n+1} \frac{x^n}{2n} e^{-\frac{x^n}{2}} \text{WhittakerM}\left( \frac{1}{n} - \frac{n+1}{2n} - 1, \frac{n+1}{2n} + \frac{1}{2}, x^n \right) \right) \\
& \quad \frac{2 n + 1}{(n + 1) (2 n + 1)} \\
\end{align*}
\]
```

```
> start_time := time(); # CPU seconds consumed in this session
int(exp(-x^n),x); # do something
time_used := time()-start_time;
```

```
\[
\begin{align*}
\frac{1}{n} & \left( n^2 x^{-n+1} (n x^n + n + 1) \left( x^n \right)^{-n+1} \frac{x^n}{2n} e^{-\frac{x^n}{2}} \text{WhittakerM}\left( \frac{1}{n} - \frac{n+1}{2n}, \frac{n+1}{2n} + \frac{1}{2}, x^n \right) \\
& \quad + n x^{-n+1}(n+1) \left( x^n \right)^{-n+1} \frac{x^n}{2n} e^{-\frac{x^n}{2}} \text{WhittakerM}\left( \frac{1}{n} - \frac{n+1}{2n} - 1, \frac{n+1}{2n} + \frac{1}{2}, x^n \right) \right) \\
& \quad \frac{2 n + 1}{(n + 1) (2 n + 1)} \\
\end{align*}
\]
```

Often one observes that doing the same thing again consumes much less CPU time. (Caching of results?)
\section*{10 Formatted output; using data files}

Remember:

-- \texttt{print} does 'normal' printing of an object;
(very large objects are not printed in detail).

-- \texttt{lprint} uses 'typewriter' printing

Examples:

\begin{verbatim}
> print(sqrt(x^3));
\end{verbatim}
\[
\sqrt{x^3}
\]

\begin{verbatim}
> lprint(sqrt(x^3));
\end{verbatim}
\[
(x^3)^{1/2}
\]

\begin{verbatim}
> A := Matrix(10,10,(i,j)->i+j):
> print(A);
\end{verbatim}
\begin{verbatim}
     [ 2  3  4  5  6  7  8  9 10 11 ]
     [ 3  4  5  6  7  8  9 10 11 12 ]
     [ 4  5  6  7  8  9 10 11 12 13 ]
     [ 5  6  7  8  9 10 11 12 13 14 ]
     [ 6  7  8  9 10 11 12 13 14 15 ]
     [ 7  8  9 10 11 12 13 14 15 16 ]
     [ 8  9 10 11 12 13 14 15 16 17 ]
     [ 9 10 11 12 13 14 15 16 17 18 ]
     [10 11 12 13 14 15 16 17 18 19 ]

> lprint(A); # not so nice
\end{verbatim}
\begin{verbatim}
Matrix(10, 10, {(1, 1) = 2, (1, 2) = 3, (1, 3) = 4, (1, 4) = 5, (1, 5) = 6, (1, 6) = 7, (1, 7) = 8, (1, 8) = 9, (1, 9) = 10, (1, 10) = 11, (2, 1) = 3, (2, 2) = 4, (2, 3) = 5, (2, 4) = 6, (2, 5) = 7, (2, 6) = 8, (2, 7) = 9, (2, 8) = 10, (2, 9) = 11, (2, 10) = 12, (3, 1) = 4, (3, 2) = 5, (3, 3) = 6, (3, 4) = 7, (3, 5) = 8, (3, 6) = 9, (3, 7) = 10, (3, 8) = 11, (3, 9) = 12, (3, 10) = 13, (4, 1) = 5, (4, 2) = 6, (4, 3) = 7, (4, 4) = 8, (4, 5) = 9, (4, 6) = 10, (4, 7) = 11, (4, 8) = 12, (4, 9) = 13, (4, 10) = 14, (5, 1) = 6, (5, 2) = 7, (5, 3) = 8, (5, 4) = 9, (5, 5) = 10, (5, 6) = 11, (5, 7) = 12, (5, 8) = 13, (5, 9) = 14, (5, 10) = 15, (6, 1) = 7, (6, 2) = 8, (6, 3) = 9, (6, 4) = 10, (6, 5) = 11, (6, 6) = 12, (6, 7) = 13, (6, 8) = 14, (6, 9) = 15, (6, 10) = 16, (7, 1) = 8, (7, 2) = 9, (7, 3) = 10, (7, 4) = 11, (7, 5) = 12, (7, 6) = 13, (7, 7) = 14, (7, 8) = 15, (7, 9) = 16, (7, 10) = 17, (8, 1) = 9, (8, 2) = 10, (8, 3) = 11, (8, 4) = 12, (8, 5) = 13, (8, 6) = 14, (8, 7) = 15, (8, 8) = ]
\end{verbatim}
A := Matrix(11,11, (i,j)->i+j):
print(A);

11 x 11 Matrix

Data Type: anything
Storage: rectangular
Order: Fortran_order

Formatted output uses a syntax analogous as in C, or Matlab:

printf("This is the integer number %d",100):
This is the integer number 100

> a,b:=3.987,4.098;

3.99 + 4.10 = 8.08

> printf("%5.2f + %5.2f = %5.2f",a,b,a+b):
3.99 + 4.10 = 8.08
%a is a general purpose format specifier
(in particular, for printing symbolic objects):

> x := 1;

> printf("%a, %a, %a", x, exp(I*z), int(y^2, y));

1, exp(I*z), 1/3*y^3

Example: print a Matrix

> A := Matrix(5, 5, (i, j) -> 1/(i+j-1)):
> for i from 1 to 5 do
> printf("%5.2f \n", A[i,1..5])
> end do:

1.00 0.50 0.33 0.25 0.20
0.50 0.33 0.25 0.20 0.17
0.33 0.25 0.20 0.17 0.14
0.25 0.20 0.17 0.14 0.12
0.20 0.17 0.14 0.12 0.11

sprintf prints to string:

> str := sprintf("x = %5.2f",2.2676);
> str;

"x = 2.27"

fprintf prints to data file:

Example:

Open (new) data file in my home directory:

> currentdir();
    "C:\Users\wauzinge\Documents\act\L V A\CM\Maple-VO\VO SS 2019"
> fd := fopen("my_data_file.dat",WRITE); # fd is file pointer
    fd := 1
> fprintf(fd,"%12.8f \n",Pi); # returns number of characters
    written

14
> fclose(fd);

Data input (interactive or from data file): See Part V.

================================================================================ end of Part III ==