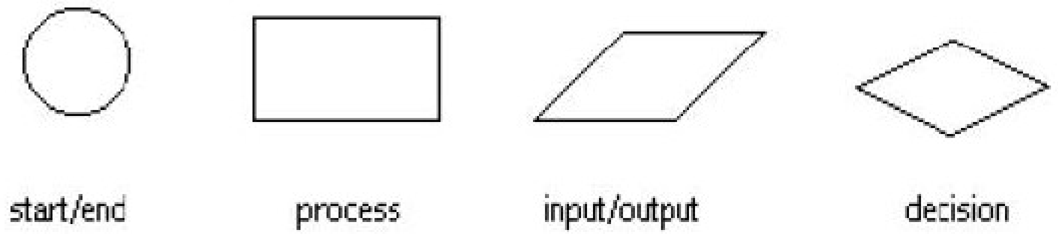
Introduction to Programming with SMath Studio

Prepared by Gilberto E. Urroz, September 2009

Programming structures and flowcharts

Programming, in the context of numerical applications, simply means controlling a computer, o other calculating device, to produce a certain numerical output. In this context, we recognize three main programming structures, namely, (a) sequential structures; (b) decision structures; and (c) loop structures. Most numerical calculations with the aid of computers or other programmable calculating devices (e.g., calculators) can be accomplished by using one of more of these structures.

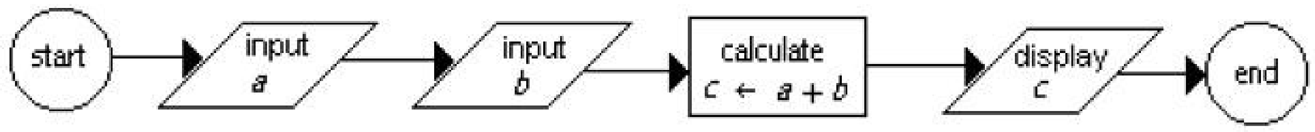
The operation of these programming structures will be illustrated by the use of flow charts. A flow chart is just a graphical representation of the process being programmed. It charts the flow of the programming process, thus its name. The figure below shows some of the most commonly used symbols in flowcharts:



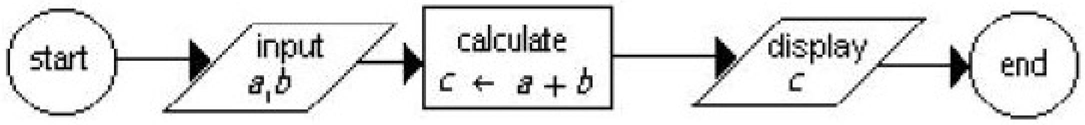
In a flowcharts, these symbols would be connected by arrows pointing in the direction of the process flow.

Sequential structures

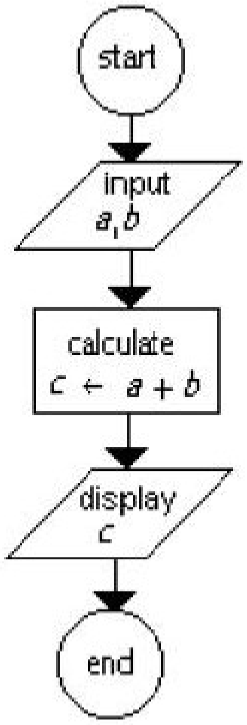
A complete programming flowchart would have *start* and *end* points, and at least one *process* block in between. That would constitute the simplest case of a *sequential structure.* The following figure shows a sequential structure for the calculation of a sum:



This flowchart can be simplified slightly by using a single input block to enter both *a* and *b*:



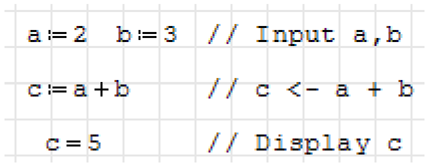
Typically, a sequential structure is shown following a vertical direction:



The sequential structure shown in this flowchart can also be represented using *pseudo-code*. Pseudo-code is simply writing the program process in a manner resembling common language, e.g.,

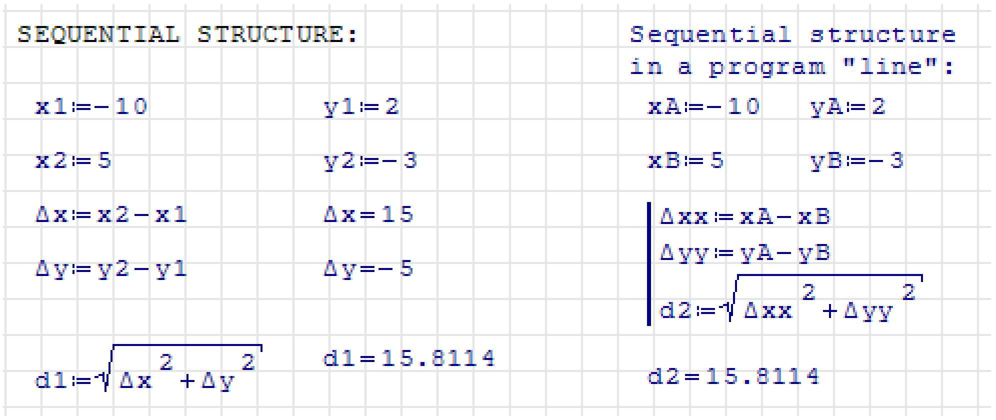
```
Start
Input a, b
c ← a + b[1]
Display c
End
```

A flowchart or pseudo-code can be translated into code in different ways, depending on the programming language used. In *SMath Studio*, this sequential structure could be translated into the following commands:



```
a:=2  b:=3   // Input a,b
c:=a+b       // c <- a + b
 c=5         // Display c
```

Here is another example of a sequential structure in *SMath Studio* showing more than one calculation step. The sequential structure in *SMath Studio* doesn't have to follow a strict vertical direction, as illustrated below.



SEQUENTIAL STRUCTURE:

$x1:=-10$ $y1:=2$

$x2:=5$ $y2:=-3$

$\Delta x:=x2-x1$ $\Delta x=15$

$\Delta y:=y2-y1$ $\Delta y=-5$

$d1:=\sqrt{\Delta x^2+\Delta y^2}$ $d1=15.8114$

Sequential structure in a program "line":

$xA:=-10$ $yA:=2$

$xB:=5$ $yB:=-3$

$\Delta xx:=xA-xB$

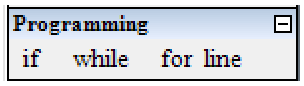
$\Delta yy:=yA-yB$

$d2:=\sqrt{\Delta xx^2+\Delta yy^2}$

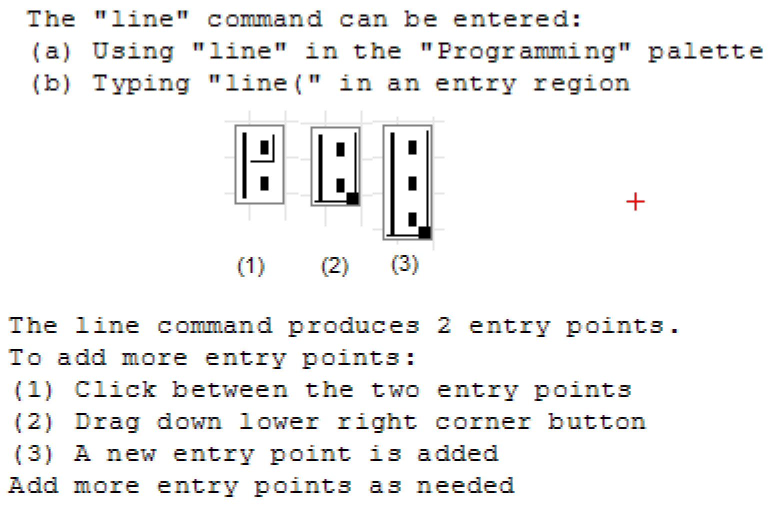
$d2=15.8114$

<u>The *line* command and the *Programming* palette</u>

The sequential structure is shown in the right-hand side of the figure above collecting the calculation steps under a <u>programming line</u>. The figure below illustrate the instructions to insert a programming line in a *SMath Studio* worksheet. The *line* command, together with other programming commands, is listed in the *Programming* palette shown here:



1 The algorithmic statement $c \leftarrow a+b$ represents the assignment $c := a + b$ in *SMath Studio*



```
The "line" command can be entered:
 (a) Using "line" in the "Programming" palette
 (b) Typing "line(" in an entry region
```

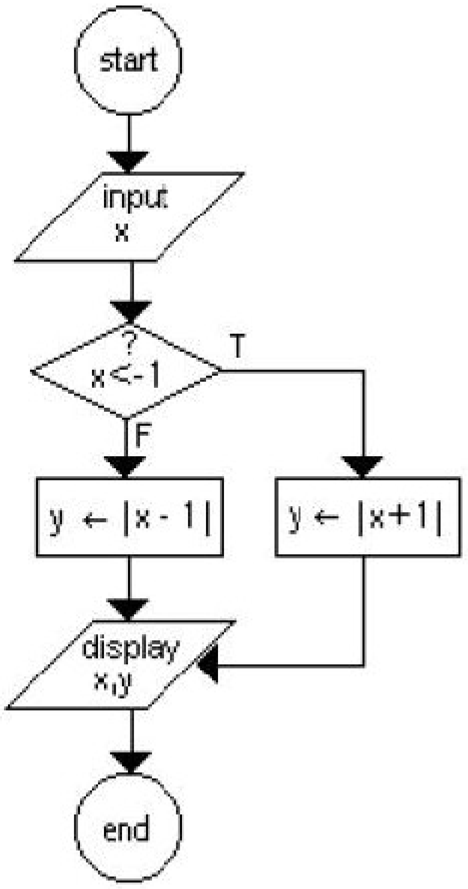
```
The line command produces 2 entry points.
To add more entry points:
(1) Click between the two entry points
(2) Drag down lower right corner button
(3) A new entry point is added
Add more entry points as needed
```

The *line* command can be used to add sequential structures to entry points in other programming instructions as will be illustrated below.

Decision structure

A decision structure provides for an alternative path to the program process flow based on whether a logical statement is true or false. As an example of a decision structure, consider the flowchart for the function listed below:

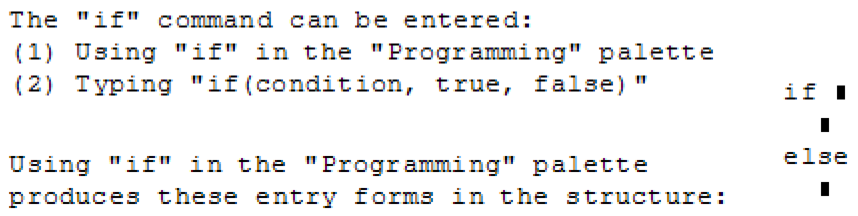
$$f(x) = \begin{cases} |x+1|, \text{ if } x < -1 \\ |x-1|, \text{ if } x \geq -1 \end{cases}.$$



The flowchart is shown on the right. The corresponding pseudo-code is shown below:

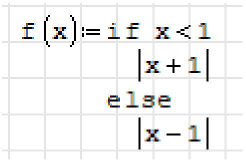
```
start
input x
if  x < -1  then
   y ← |x+1|
else
   y ← |x-1|
display x,y
end
```

In *SMath Studio* a decision structure is entered using the *if* command. Instructions on entering the *if* command are shown below:



```
The "if" command can be entered:
(1) Using "if" in the "Programming" palette
(2) Typing "if(condition, true, false)"

Using "if" in the "Programming" palette
produces these entry forms in the structure:
```

To illustrate the use of the *if* command within *SMath Studio*, we enter the function *f*(*x*), defined above, as shown here: → → → → → → → → → → → → →

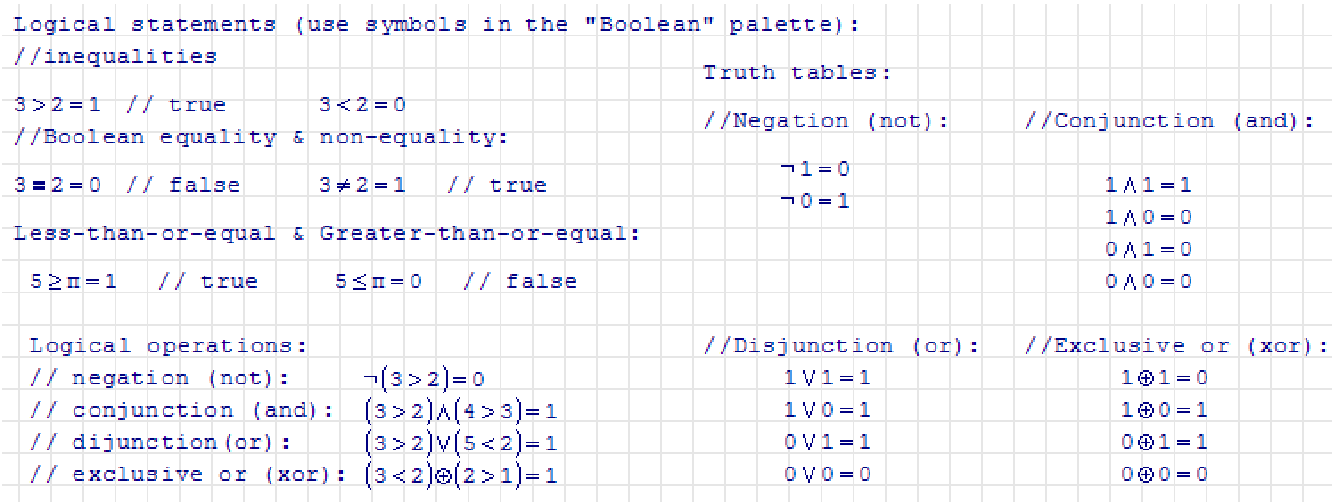


$$f(x) := \text{if } x<1 \quad |x+1| \quad \text{else} \quad |x-1|$$

Logical statements and logical operations

Decision structures require a *condition* to trigger a decision on the program flow. Such condition is represented by a logical statement. Within the context of programming numerical calculations, a *logical statement* is a mathematical statement that can be either true or false, e.g., *3>2*, *5<2*, etc. In *SMath Studio* the logical outcomes *true* and *false* are represented by the integer values *1* and *0*, respectively.

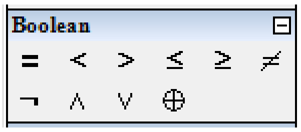
The following figure illustrates examples of logical statements. It also includes examples of the four *logical operations*: (1) negation (*not* ¬); (2) conjunction (*and* ∧); (3) disjunction (*or* ∨); and, (4) exclusive or (*xor* ⊗). The figure also includes the truth tables for these four operations. A *truth table* is a table showing the outcome of all possible combinations of *true* and *false* statements.



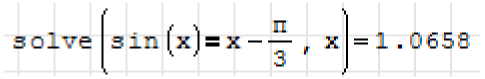
```
Logical statements (use symbols in the "Boolean" palette):
//inequalities
3>2=1  // true        3<2=0
//Boolean equality & non-equality:
3=2=0  // false       3≠2=1   // true
Less-than-or-equal & Greater-than-or-equal:
 5≥π=1   // true      5≤π=0   // false

 Logical operations:
 // negation (not):       ¬(3>2)=0
 // conjunction (and):    (3>2)∧(4>3)=1
 // dijunction(or):       (3>2)∨(5<2)=1
 // exclusive or (xor):   (3<2)⊕(2>1)=1
```

```
Truth tables:

//Negation (not):      //Conjunction (and):
     ¬1=0                    1∧1=1
     ¬0=1                    1∧0=0
                             0∧1=0
                             0∧0=0

//Disjunction (or):    //Exclusive or (xor):
     1∨1=1                   1⊕1=0
     1∨0=1                   1⊕0=1
     0∨1=1                   0⊕1=1
     0∨0=0                   0⊕0=0
```

The symbols for the comparison operators (=, <, >, ≤ , ≥ , ≠) and logical operations (¬ , ∧ , ∨, ⊗) are available in the *Boolean* palette shown here:



Notice the presence of the *Boolean equal* (bold =) which represents a comparison, rather than an evaluation. The Boolean equal sign can be used to define equations in *SMath Studio*, e.g., in this *solve* command:



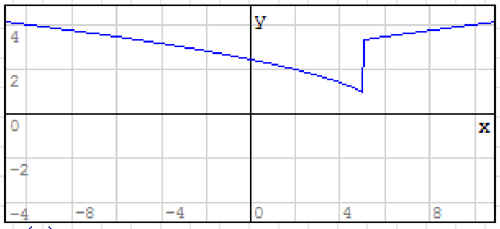
$$\text{solve}\left(\sin(x) = x - \frac{\pi}{3}, x\right) = 1.0658$$

© 2009 Gilberto E. Urroz

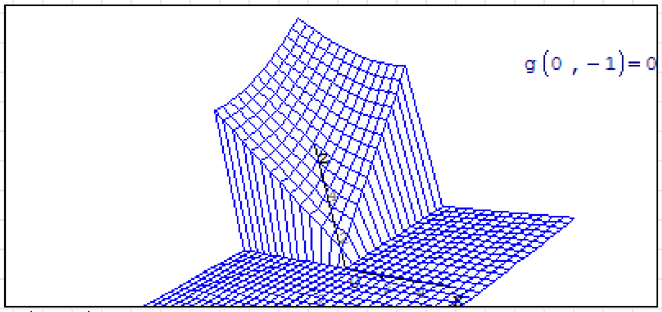
Examples of *if* statements used to define functions

The following figure illustrates other examples of *if* statements used in the definitions of functions:

```
Decision structures applied to definition of functions:

                                                   g(x , y):=if (x<0)∧(y<0)    g(2 , 0)=0
                                                          √(x^2 + y^2)
f(x):=if x<5                                            else                 g(-3 , -2)=3.6056
          √(6-x)              f(2)=2                     0
      else
          √(6+x)              f(19)=5
```



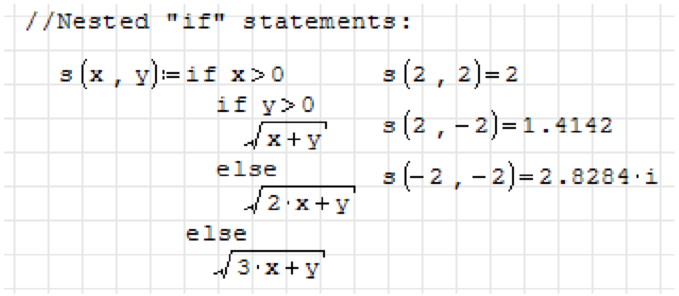
f(x)



g(x , y)

Nested *if* statements

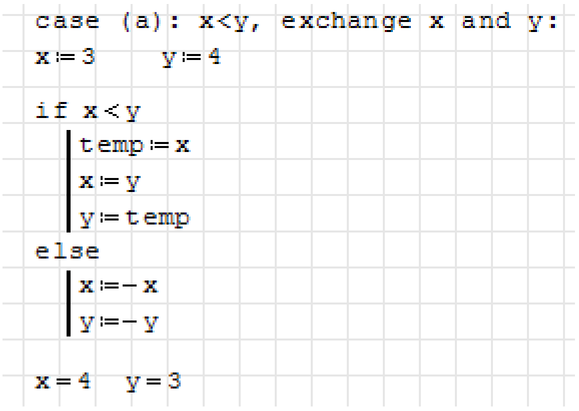
Decision structures, i.e., *if* statements, can be nested as illustrated below:



```
//Nested "if" statements:

  s(x , y):=if x>0             s(2 , 2)=2
                if y>0
                   √(x+y)      s(2 , -2)=1.4142
                else
                   √(2·x+y)    s(-2 , -2)=2.8284·i
            else
               √(3·x+y)
```

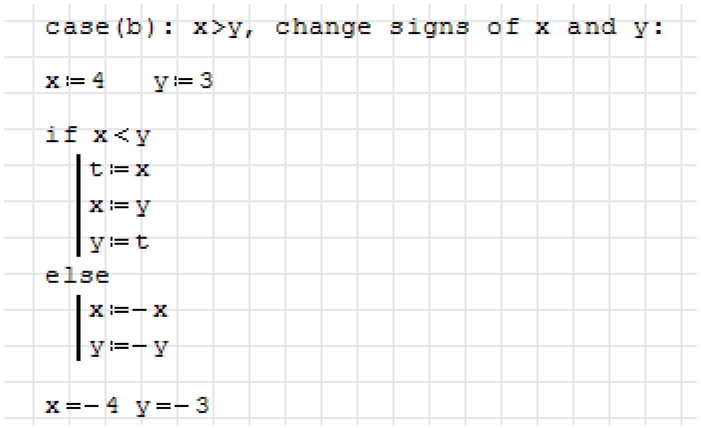
Combination of *if* statements with *line* statements

The following two examples illustrate an *if* statement that exchanges the values of two variables *x* and *y* if *x<y,* or changes the signs of both variables otherwise. In the first case, *x<y,* thus, the values get exchanged:



```
case (a): x<y, exchange x and y:
x:=3    y:=4

if x<y
  | temp:=x
  | x:=y
  | y:=temp
else
  | x:=-x
  | y:=-y

x=4   y=3
```

In the second case, $x<y$, thus the *else* clause is activated and both x and y have their signs changed:

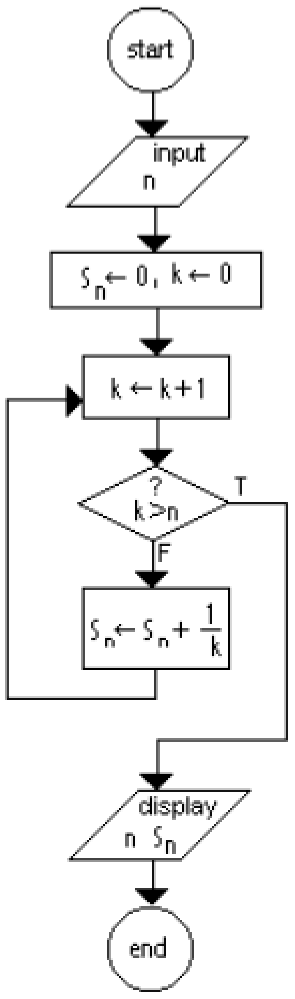


```
case(b): x>y, change signs of x and y:

x := 4    y := 3

if x < y
   | t := x
   | x := y
   | y := t
else
   | x := -x
   | y := -y

x = -4  y = -3
```

Loop structures

In a loop structure the process flow is repeated a finite number of times before being send out of the loop. The middle part of the flowchart to the right illustrates a loop structure. The flowchart shown represents the calculation of a sum, namely,

$$S_n = \sum_{k=1}^{n} \frac{1}{k} \quad .$$


start
input n
$S_n \leftarrow 0,\ k \leftarrow 0$
$k \leftarrow k+1$
? $k>n$
T
F
$S_n \leftarrow S_n + \frac{1}{k}$
display n S_n
end

The sum S_n is initialized as $S_n \leftarrow 0$, and an index, k, is initialized as $k \leftarrow 0$ before the control is passed on to the loop. The loop starts incrementing k and then it checks if the index k is larger than its maximum value n. The sum is incremented within the loop, $S_n \leftarrow S_n + 1/k$, and the process is repeated until the condition $k>n$ is satisfied. After the condition in the decision block is satisfied (T = true), the control is sent out of the loop to the *display* block.

In terms of programming statements, there are two possible commands in *SMath Studio* to produce a loop: *while* and *for*. The *pseudo-code* for a *while* loop interpreting the flowchart to the right is shown below:

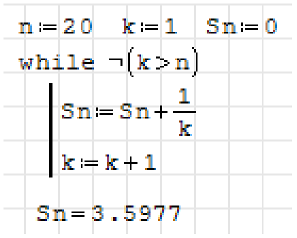
```
start
input n
Sn ← 0
k ← 0
do while ~(k>n)
   k ← k + 1
   Sn ← Sn + 1/k
end loop
display n, Sn
end
```

Since a *while* loop checks the condition at the top of the loop, the condition was converted to ~ ($k > n$), i.e., $not(k>n) = k \le n$.

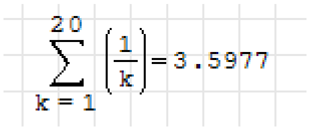
The *while* command in *SMath Studio*

```
//The "while" command can be entered by using:
(1) Using "while" in the "Programming" palette          while ▮
(2) Typing "while(condition,body)"                          ▮

 The expression defining the "condition" must be modified
 within the "while" loop so that an exit can be provided.
```

The pseudo-code listed above is translated as follows in *SMath Studio*: → →
This summation can also be calculated using *SMath Studio's* summation command:

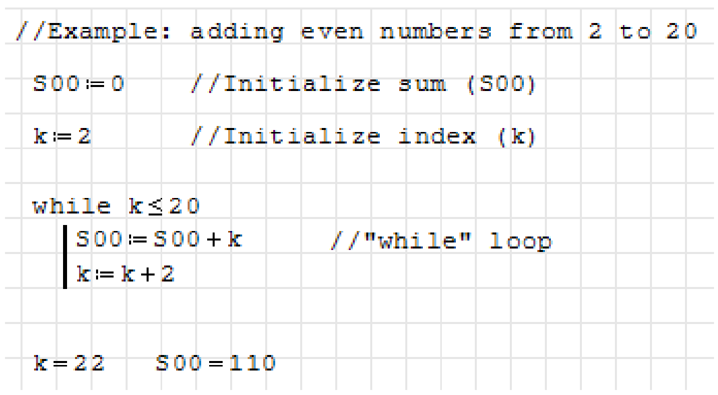


```
n:=20   k:=1   Sn:=0
while ¬(k>n)
    | Sn:=Sn+$\frac{1}{k}$
    | k:=k+1
 Sn=3.5977
```

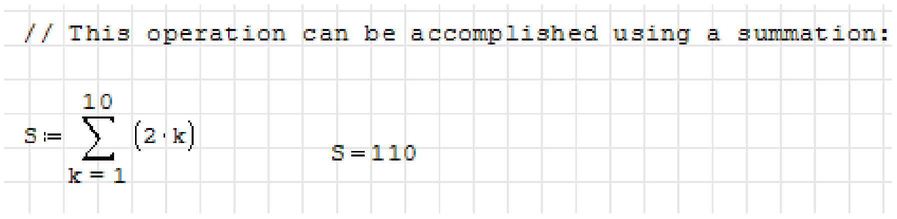


$$\sum_{k=1}^{20}\left(\frac{1}{k}\right)=3.5977$$

Here is another example of a *while* loop in *SMath Studio*:



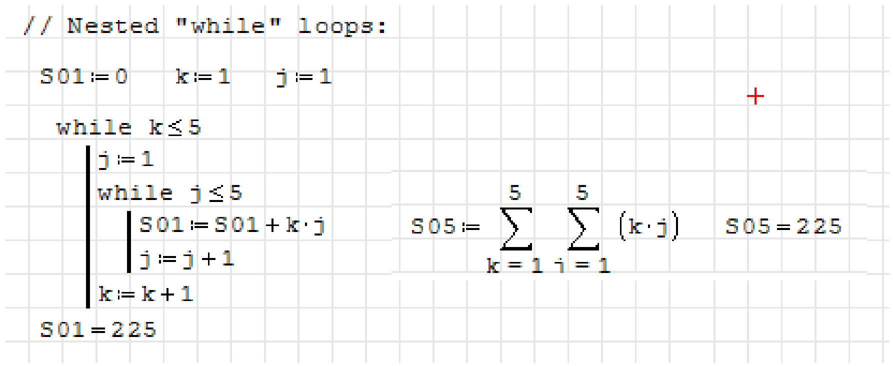
```
//Example: adding even numbers from 2 to 20

 S00:=0     //Initialize sum (S00)

 k:=2       //Initialize index (k)

 while k≤20
    | S00:=S00+k        //"while" loop
    | k:=k+2

 k=22    S00=110
```



```
// This operation can be accomplished using a summation:
```

$$S:=\sum_{k=1}^{10}(2\cdot k) \qquad S=110$$

While loops can be nested as illustrated in the left-hand side of the figure shown below. The corresponding double summation is shown on the right-hand side of the figure below.

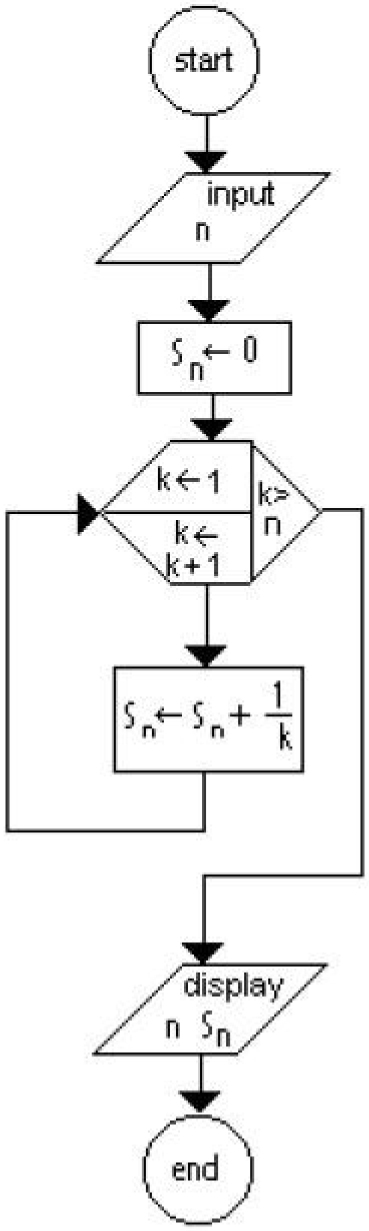


```
// Nested "while" loops:

 S01:=0    k:=1    j:=1

  while k≤5
     | j:=1
     | while j≤5
     |    | S01:=S01+k·j
     |    | j:=j+1
     | k:=k+1
 S01=225
```

$$S05:=\sum_{k=1}^{5}\sum_{j=1}^{5}(k\cdot j) \qquad S05=225$$

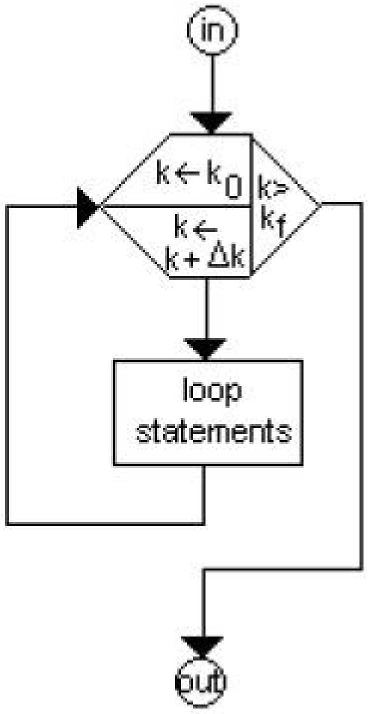
The figure to the right shows an alternative flowchart for calculating the summation:

$$S_n = \sum_{k=1}^{n} \frac{1}{k}$$



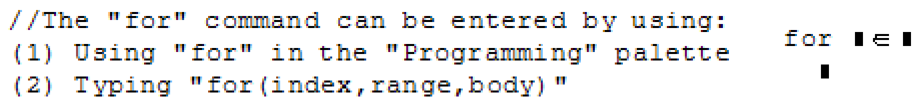
The hexagonal symbol in the flowchart shows three elements: (1) the initialization of the index, $k \leftarrow 1$; (2) the index being incremented, $k \leftarrow k + 1$; and, (3) the condition checked to exit the loop, $k > n$. This hexagonal symbol represents the *for* command for this summation.

A more general symbol for a *for* command is shown below. This symbol shows a *for* loop with index k, starting at value k_0, and ending at value k_f, with increment Δk:



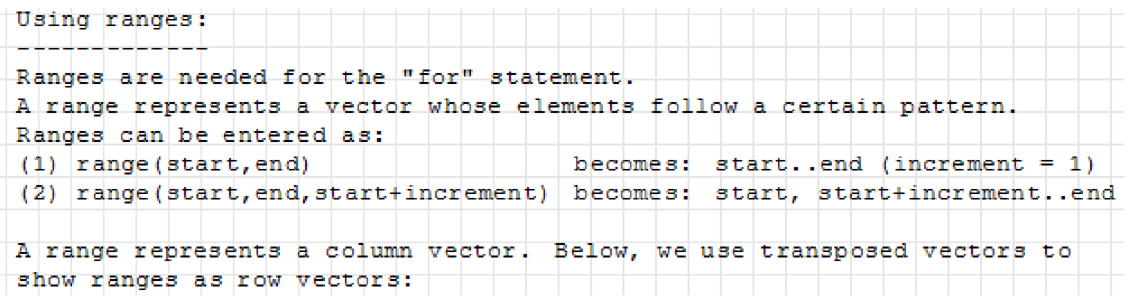
The index, therefore, takes values $k = k_0, k_0+\Delta k, k_0+2\Delta k, \ldots, k_{end}$, such that $k_{end} \le k_f$ within one Δk.

The *for* and *range* commands in *SMath Studio*

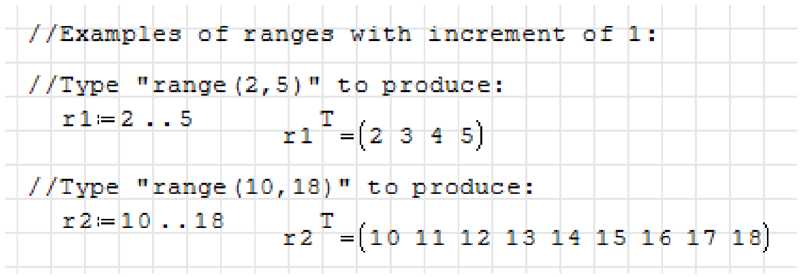


```
//The "for" command can be entered by using:
(1) Using "for" in the "Programming" palette
(2) Typing "for(index,range,body)"
```

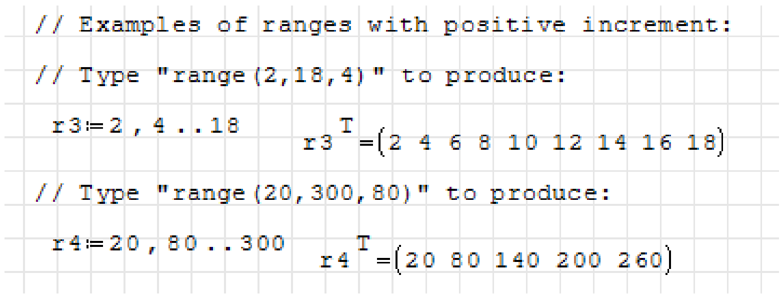
The *for* command in *SMath Studio* uses a *range* of values to indicate the values that the index k takes to complete the loop.



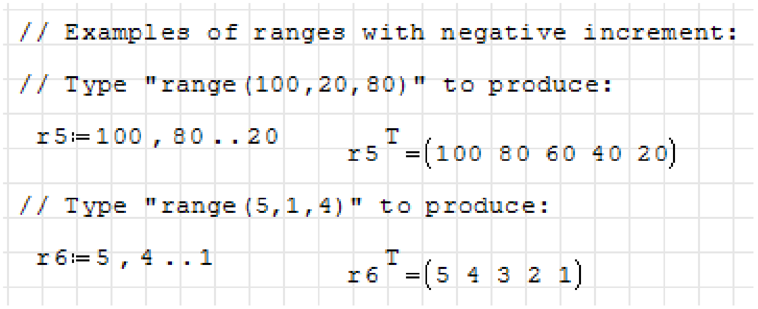
```
Using ranges:
-------------
Ranges are needed for the "for" statement.
A range represents a vector whose elements follow a certain pattern.
Ranges can be entered as:
(1) range(start,end)                   becomes: start..end (increment = 1)
(2) range(start,end,start+increment) becomes: start, start+increment..end

A range represents a column vector.  Below, we use transposed vectors to
show ranges as row vectors:
```



```
//Examples of ranges with increment of 1:

//Type "range(2,5)" to produce:
  r1≔2..5            r1^T=(2 3 4 5)

//Type "range(10,18)" to produce:
  r2≔10..18          r2^T=(10 11 12 13 14 15 16 17 18)
```

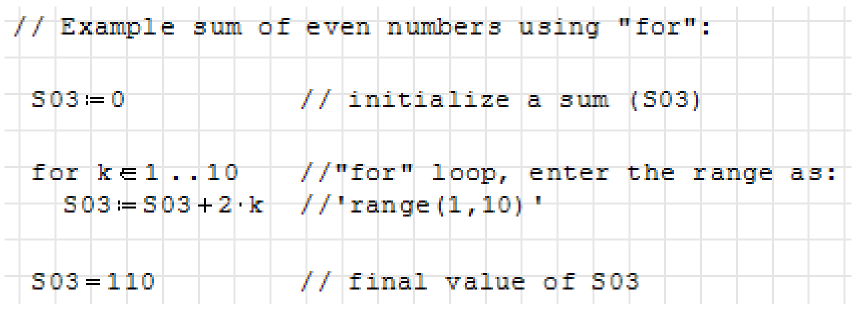


```
// Examples of ranges with positive increment:

// Type "range(2,18,4)" to produce:

 r3≔2,4..18             r3^T=(2 4 6 8 10 12 14 16 18)

// Type "range(20,300,80)" to produce:

 r4≔20,80..300     r4^T=(20 80 140 200 260)
```



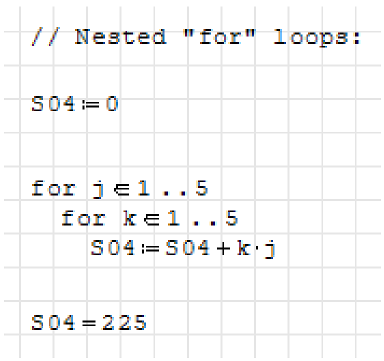
```
// Examples of ranges with negative increment:

// Type "range(100,20,80)" to produce:

 r5≔100,80..20          r5^T=(100 80 60 40 20)

// Type "range(5,1,4)" to produce:

 r6≔5,4..1              r6^T=(5 4 3 2 1)
```

Here is an example of the *for* command in *SMath Studio* using the range *1..10*:



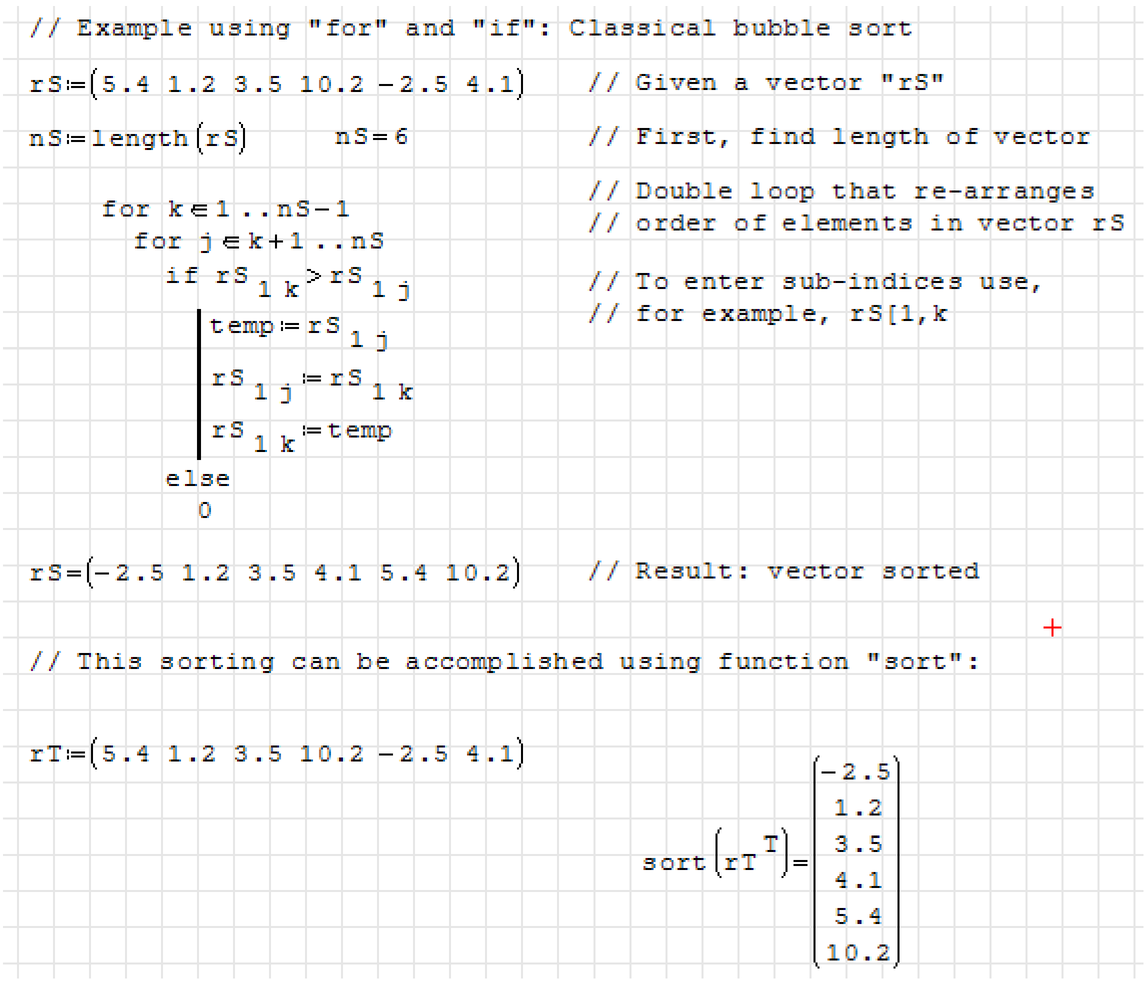
```
// Example sum of even numbers using "for":

 S03≔0              // initialize a sum (S03)

 for k∈1..10        //"for" loop, enter the range as:
   S03≔S03+2·k      //'range(1,10)'


 S03=110            // final value of S03
```

The double summation programmed earlier using *while* loops, can be programmed also using *for* loops:



```
// Nested "for" loops:

S04:=0

for j∈1..5
  for k∈1..5
    S04:=S04+k·j

S04=225
```

A programming example using sequential, decision, and loop structures
This example illustrates a program in *SMath Studio* that uses all three programming structures. This is the classic "bubble" sort algorithm in which, given a vector of values, the smallest ("lighter") value bubbles up to the top. The program shown uses a row vector *rS*, and refers to its elements using sub-indices, e.g., S_{1k}, etc. The example shows how to enter sub-indices. The output from the program is the sorted vector *rS*.



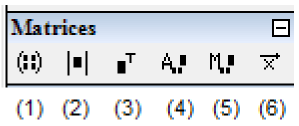
```
// Example using "for" and "if": Classical bubble sort

rS:=(5.4 1.2 3.5 10.2 -2.5 4.1)      // Given a vector "rS"

nS:=length(rS)      nS=6             // First, find length of vector

                                     // Double loop that re-arranges
     for k∈1..nS-1                   // order of elements in vector rS
       for j∈k+1..nS
         if rS_{1 k}>rS_{1 j}         // To enter sub-indices use,
           | temp:=rS_{1 j}           // for example, rS[1,k
           | rS_{1 j}:=rS_{1 k}
           | rS_{1 k}:=temp
         else
           0

rS=(-2.5 1.2 3.5 4.1 5.4 10.2)       // Result: vector sorted

// This sorting can be accomplished using function "sort":

rT:=(5.4 1.2 3.5 10.2 -2.5 4.1)
```

$$\mathrm{sort}\left(rT^{T}\right)=\begin{pmatrix}-2.5\\1.2\\3.5\\4.1\\5.4\\10.2\end{pmatrix}$$

Many programming applications, such as the one shown above, use vectors and matrices. Luckily, *SMath Studio* already includes a good number of functions that apply to matrices, e.g,:

- Creating matrices: *augment, diag, identity, mat, matrix, stack*
- Extracting rows, columns, elements: *col, el, row, submatrix, vminor, minor*
- Characterizing matrices: *cols, det, length, max, min, norm1, norme, normi, rank, rows, tr*
- Sorting functions: *csort, reverse, rsort, sort*
- Matrix operations: *alg, invert, transpose*

Some of these functions are also available in the *Matrices* palette:



(1) Matrix 3x3 (Ctrl+M) (*mat*)
(2) Determinant (*det*)
(3) Matrix transpose (Ctrl+1) (*transpose*)
(4) Algebraic addition to matrix (*alg*)
(5) Minor (*minor*)
(6) Cross product

Examples of matrix operations were presented in the document ***Introduction to the use of SMath Studio.***

Steps in programming
The following are recommended steps to produce efficient programs:

(1) Clearly define problem being solved
(2) Define inputs and outputs of the program
(3) Design the algorithm using flowcharts or pseudo-code
(4) Program algorithm in a programming language (e.g., *SMath Studio* programming commands)
(5) Test code with a set of known values

Errors in programming
Typically there are three main types of errors in developing a program:

(1) *Syntax errors*: when the command doesn't follow the programming language syntax. These are easy to detect as the program itself will let you know of the syntax violations.
(2) *Run-time errors*: errors due to mathematical inconsistencies, e.g., division by zero. These may be detected by the program also.
(3) *Logical errors*: these are errors in the algorithm itself. These are more difficult to detect, thus the need to test your programs with known values. Check every step of your algorithm to make sure that it is doing what you intend to do.