Introduction to Programming with SMath Studio

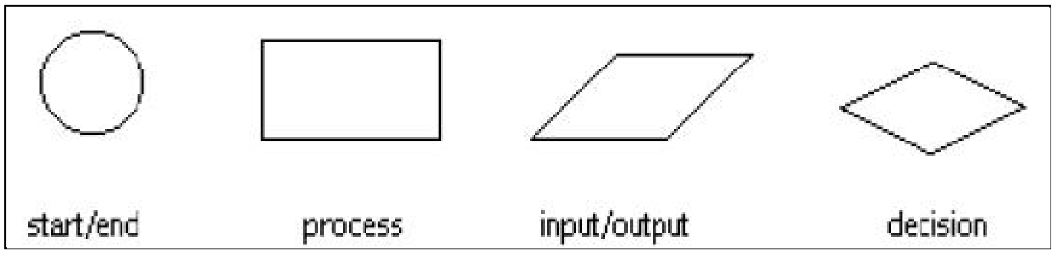
By Gilberto E. Urroz, October 2010 - Updated July 2012

In this section we introduce basic concepts of programming for numerical solutions in SMath Studio.

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, or combinations of the same.

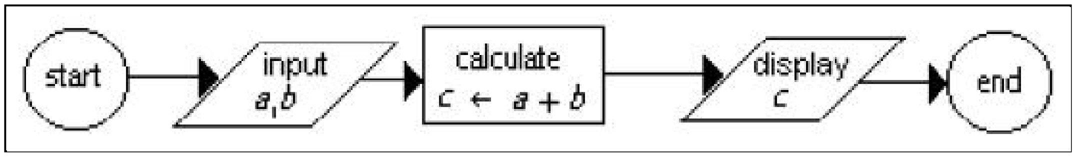
The operation of these programming structures will be illustrated by the use of flow charts. A flow chart is just a graphical representation of the process being programmed. It charts the flow of the programming process, thus its name. The figure below shows some of the most commonly used symbols in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

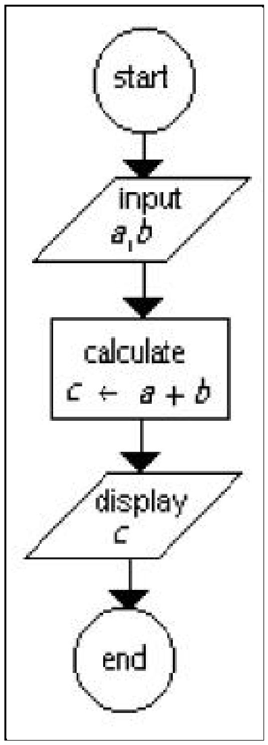


In a flowchart, these symbols would be connected by arrows pointing in the direction of the process flow.

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:





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
   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:

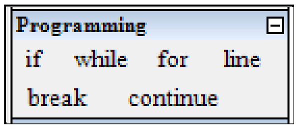
```
----------------------------------
  a:= 2   b:= 3    // Input a,b
  c:= a + b        // c <- a + b
  c = 5            // Display c
----------------------------------
```

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.

$x1 := -10$	$y1 := 2$
$x2 := 5$	$y2 := -3$
$\Delta x := x2 - x1$	$\Delta x = 15$
$\Delta y := y2 - y1$	$\Delta y = -5$
$d1 := \sqrt{\Delta x^2 + \Delta y^2}$	$d1 = 15.81$

The "line" command and the Programming palette

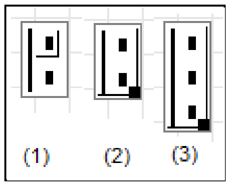
A couple of examples of sequential structures were shown in the examples above. In SMath Studio, we can collect the calculation steps under a programming line. The figure to the right -> -> -> -> illustrate the instructions to insert a programming line in a SMath Studio worksheet. The "line" command, together with other programming commands, is listed in the Programming palette shown.



The "line" command can be entered in one of these ways:
  (a) Using "line" in the "Programming" palette
  (b) Typing "line(" in the worksheet

Creating a programming line - adding entry points

By default, the "line" command produces a vertical line with two entry points or placeholders, as shown in (1), below:



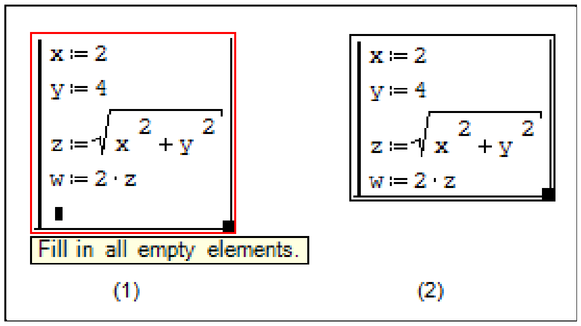
To add additional entry points, or placeholders, to a line proceed as follows:

(1) Click between the two entry points, or to the left of the line
(2) Drag down the lower right corner button that shows up
(3) A new entry point is added

Repeat steps (1) and (2) to add more entry points as needed.

Removing entry points

If you extend the programming line past the number of entries needed, you can reduce the number of entry points by clicking on the lower right corner button, and dragging it upwards until the entry points not needed have been removed. The figure below shows the two steps described above.



Using the "line"command

The "line" command can be used to keep together a series of commands in a sequential programming structurein a SMath Studio Worksheet. For example, the commands calculating Δxx, Δyy, and d2, in the following listing have been placed within a programming "line":

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

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

$$\left| \begin{array}{l} \Delta xx := xA - xB \\ \Delta yy := yA - yB \\ d2 := \sqrt{\Delta xx^{2} + \Delta yy^{2}} \end{array} \right.$$

$d2 = 15.81$

----

in the example shown above, the only purpose of using the programming line is to keep those three calculation lines together. You can then select them and move them as a single unit. Next, we show how to use a command line for defining functions.

Defining a function with a "line" command

SMath Studio allows the definition of functions whose body is defined by a programming s equence. For example, the following function, f.1(x,y), is defined as a sequence structure contained within a "line" command:

$a := 2 \quad b := 5$

$$f1(x, y) := \left| \begin{array}{l} r := a \cdot x + b \cdot y \\ s := b \cdot x + a \cdot y \\ \sqrt{r^{2} + s^{2}} \end{array} \right.$$

Examples of calculating with f1(x,y): $f1(-2, 4) = 16.12 \quad f1(-2, -4) = 30$

$f1(2, -10) = 47.07 \quad f1(-2, 10) = 47.07$

Here is another example of using a programming line to define a function. Function Rh(D,y) calculates the hydraulic radius of a circular open channel given the diameter, D, and the flow depth, y:

$$Rh(D, y) := \left| \begin{array}{l} \theta := \arccos\left(1 - 2\cdot\frac{y}{D}\right) \\ A := \frac{D^2}{4}\cdot(\theta - \sin(\theta)\cdot\cos(\theta)) \\ P := D\cdot\theta \\ \frac{A}{P} \end{array} \right.$$

Calculating with Rh(D,y):

$Rh(10, 5) = 2.5$ $Rh(3.5, 0.5) = 0.31$

$Rh(5, 2) = 1.07$ $Rh(1.2, 0.1) = 0.06$

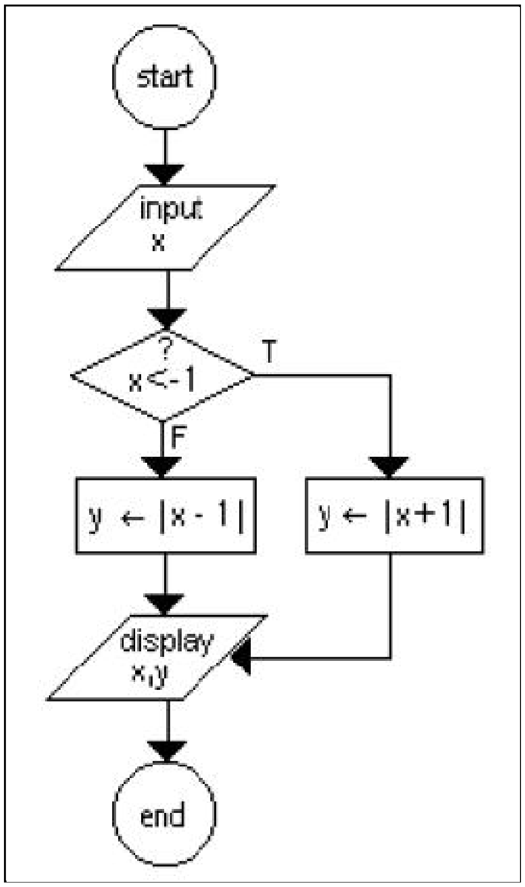
Decision structure

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:

DECISION STRUCTURE - The "if" command:

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 form: -> -> -> -> ->

```
if ∎
   ∎
else
   ∎
```

The general operation of the "if" command is as follows:

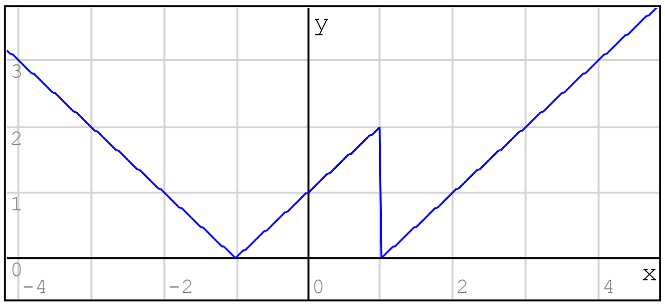
```
if condition
    command(s) 1
else
    command(s) 2
```

* The "condition" is a logical statement that could be true or false.
* If "condition" is true then "command(s) 1" get(s) executed.
* If "condition" is false, then "command(s) 2", associated with the "else" particle get(s) executed as default command(s).

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

```
f2(x) := if x < 1
             |x + 1|
         else
             |x − 1|
```

A plot of this function is shown below:

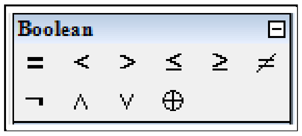


f2(x)

Comparison operators and logical operators - The "Boolean" palette

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.

Some of the most elementary logical statements are those obtained by comparing numbers. Comparison operators and logical operators are available in the "Boolean" palette in SMath Studio:



The top line of the "Boolean" palette contains the comparison operators, while the bottom line of the same palette contains the logical operators. The comparison operators should be familiar to all: equal (=), less than (<), greater than (<), etc. The logical operators are: negation (¬), conjunction (and), disjunction (or), and exclusive or, or xor.

Examples of comparison operations

The following are some examples of comparison operations which resultin a logical value of 0 or 1:

Inequalities: $\boxed{3>2=1}$ // true $\boxed{3<2=0}$ // false

Boolean equality & non-equality:

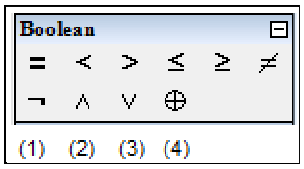
$\boxed{3=2=0}$ // false $\boxed{3\neq 2=1}$ // true

Less-than-or-equal & Greater-than-or-equal:

$\boxed{5\geq \pi=1}$ // true $\boxed{5\leq \pi=0}$ // false

Logical operators and truth tables

The "Boolean" palette in SMath Studio includes the following four logical operations:



(1) negation (not):	$\neg(3<2)=1$	$\neg(3>2)=0$
(2) conjunction (and):	$(3>2)\wedge(4>3)=1$	$(3<2)\wedge(4>3)=0$
(3) dijunction (or):	$(3>2)\vee(5<2)=1$	$(3<2)\vee(5<2)=0$
(4) exclusive or (xor):	$(3<2)\oplus(2>1)=1$	$(3>2)\oplus(2>1)=0$

The "negation" operator is referred to as a "unary" operator because it applies to only one logical statement. On the other hand, the other three operators (conjunction, disjunction, and exclusive or) are known as "binary" operators, because they require two logical statements to operate upon.

Truth tables refer to the result of the different logical operators. For example, the negation of a true statement is false , and vice versa. Recalling that in SMath Studio 1 stands for "true", and 0 for "false", we can show the truth tables of the four logical operators as follows:

Negation (not):

$\neg 1=0$
$\neg 0=1$

Conjunction (and):

$1\wedge 1=1$
$1\wedge 0=0$
$0\wedge 1=0$
$0\wedge 0=0$

Disjunction (or):

$1\vee 1=1$
$1\vee 0=1$
$0\vee 1=1$
$0\vee 0=0$

Exclusive or (xor):

$1\oplus 1=0$
$1\oplus 0=1$
$0\oplus 1=1$
$0\oplus 0=0$

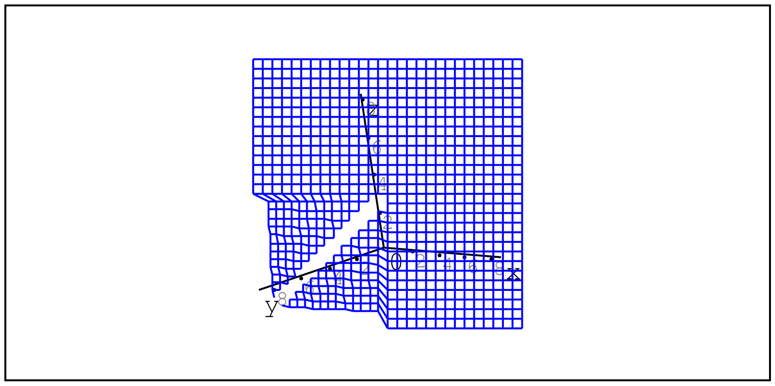
Examples of "if" statements with logical operations

The following function g1(x,y) is defined using the logical statement "(x<0) and (y<0)". Evaluations of the function, and a graph of the same, are also shown below.

```
g1(x, y) := if (x<0)∧(y<0)
                 $\sqrt{x^2+y^2}$
             else
                 0
```

$g1(2, 0) = 0$

$g1(-3, -2) = 3.61$



$g1(x, y)$

Nested "if" statements

If the decision tree includes more than one condition, then it may be necessary to "nest" one "if" statement within another as illustrated in the definition of the following function s(x,y):

```
s(x, y) := if x>0
               if y>0
                   $\sqrt{x+y}$
               else
                   $\sqrt{2\cdot x+y}$
           else
               $\sqrt{3\cdot x+y}$
```

In this case, if the condition "x>0" is true, then it is necessary to check the inner "if" statement, namely:

```
if y>0
    $\sqrt{x+y}$
else
    $\sqrt{2\cdot x+y}$
```

whose operation was discussed earlier.

Thus, for function s(x,y), the evaluation proceeds as follows:

* if x>0 and if y>0, then s = $\sqrt{x+y}$, e.g., $s(2, 2) = 2$
* if x>0 and if y<0, then s = $\sqrt{2\cdot x+y}$, e.g., $s(3, -2) = 2$
* if x<0, then s = $\sqrt{3\cdot x+y}$, e.g., $s(-2, 10) = 2$, or, $s(-2, -2) = 2.83\cdot i$

Combining the "if" command with the "line" command

The true or false conditions in a "if" statement may lead to the execution of more than one statement as illustrated in the following examples, where, "line" statements are used to produce more than one operation:

case (a): xx<yy, exchange xx and yy:

```
xx:= 3    yy:= 4

if xx< yy
    temp:= xx
    xx:= yy
    yy:= temp
else
    xx:=- xx
    yy:=- yy

xx= 4    yy= 3
```

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
```

If we wanted to turn this "if" statement into a function, we need to keep in mind the fact that a function can only return a single value. To be able to return more than one value, we need to put our results, x and y, into a column vector of two rows, e.g.,

```
f3(x, y):= | if x< y
           |     t:= x
           |     x:= y
           |     y:= t
           | else
           |     x:=- x
           |     y:=- y
           | (x)
           | (y)
```

Evaluations of this function follow:

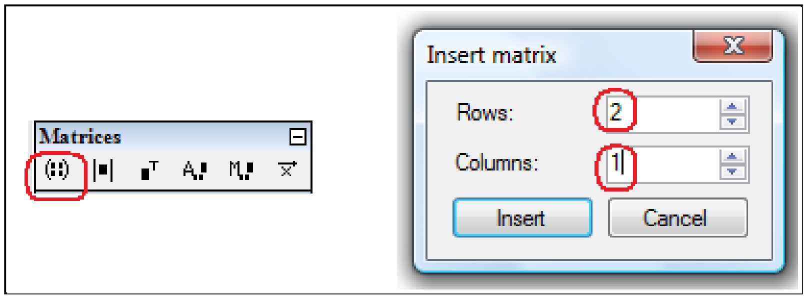
Case (a), x<y:

$$f3(3, 4)=\begin{pmatrix}4\\3\end{pmatrix}$$

Case (b), x>y:

$$f3(4, 3)=\begin{pmatrix}-4\\-3\end{pmatrix}$$

NOTE: To enter a vector you need to use the "Matrix (Cntl+M)" icon in the "Matrices" palette. Then, enter the number of rows and columns in the resulting entry form, and press [Insert]. (See figure below)

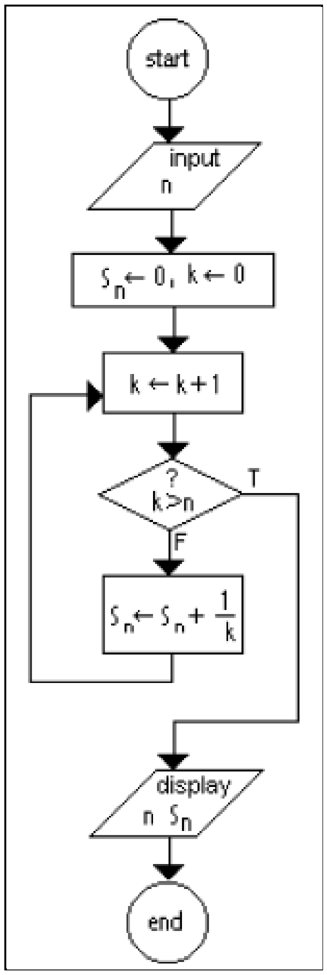


Rows: 2
Columns: 1

Then type the components of the vector or matrix in the proper placeholders.

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,


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

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

The sum Sn is initialized as Sn ← 0, and an index, k, is initialized as k ← 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, Sn ← Sn + 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
```

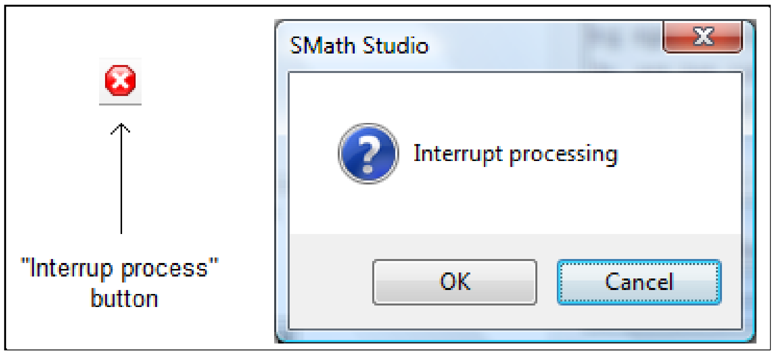
Since a "while" loop checks the condition at the top of the loop, the condition was converted to ~ (k > n), i.e., not(k>n) = $k \le n$

The while command in SMath Studio

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

```
while ■
  ■
```

The expression defining the "condition" must be modified within the "while" loop so that an exit can be provided. Otherwise, you may end up in an infinite loop that can only be stopped by using the "Interrupt" button in the SMath Studio menu (see below):



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 with
  | k := k + 2             a "line" command

k = 22   S00 = 110       //Results
```

This operation can be accomplished using a summation:

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

The "summation" symbol is available in the "Functions" palette.

Nested "while" loops

While loops can be nested as illustrated in the example below (left-hand side). The corresponding double summation is shown on the right-hand side:

Double summation with 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
```

Double-summation symbol to calculate same sum:

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

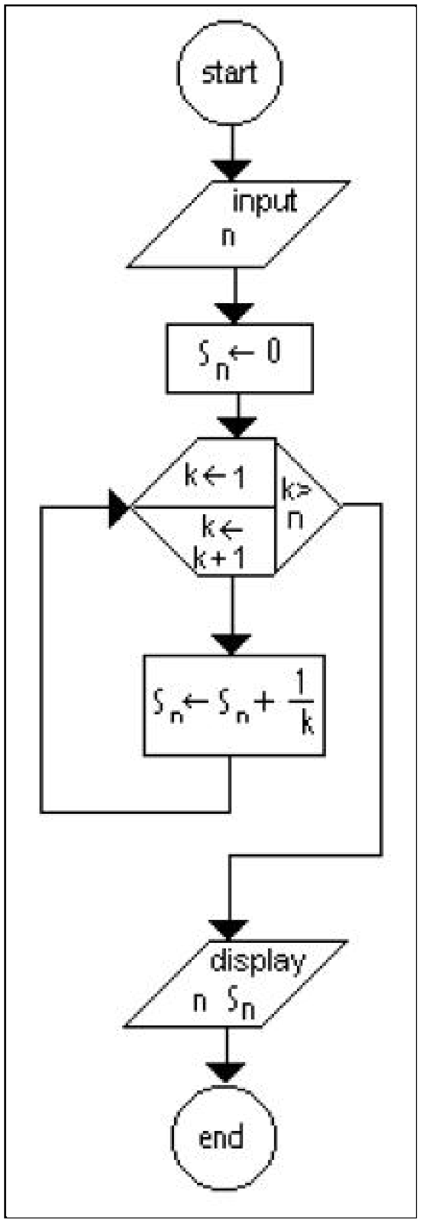
$S05 = 225$

The "for" loop flowchart

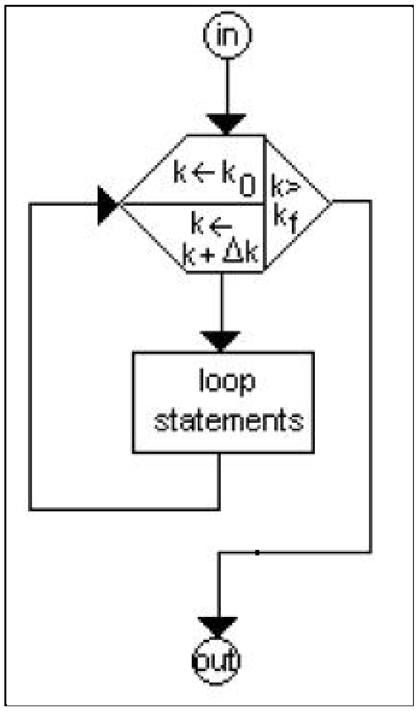
The figure to the right shows an alternative flowchart for calculating the summation:

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

------------------------->



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



The index, therefore, takes values

k = k0, k0+Δk, k0+2*Δk,…,kend,

such that $k_{end} \le k_f$ within one Δk.

The "range" command in SMath Studio

The "range" command produces a vector of indices required for setting up a "for" loop. We describe the "range" command first.

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" command generates 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: $\boxed{r1 := 2..5}$ $r1^T = (2\ 3\ 4\ 5)$

//Type "range(10,18)" to produce: $\boxed{r2 := 10..18}$ $r2^T = (10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18)$

Examples of ranges with positive increment:

// Type "range(2,18,4)" to produce: $\boxed{r3 := 2, 4..18}$ $r3^T = (2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18)$

// Type "range(20,300,80)" to produce: $\boxed{r4 := 20, 80..300}$ $r4^T = (20\ 80\ 140\ 200\ 260)$

Examples of ranges with negative increment:

// Type "range(100,20,80)" to produce: $\boxed{r5 := 100, 80..20}$ $r5^T = (100\ 80\ 60\ 40\ 20)$

// Type "range(5,1,4)" to produce: $\boxed{r6 := 5, 4..1}$ $r6^T = (5\ 4\ 3\ 2\ 1)$

The "for" command in SMath Studio

The "for" command can be entered by using: $\boxed{\text{for } ■ \in ■ \quad ■}$

(1) Using "for" in the "Programming" palette
(2) Typing "for(index,range,body)"

Here is an example of the "for" command in SMath Studio using the range 1..10 to calculate a summation: $\boxed{S03 = \sum_{k=1}^{10} 2 \cdot k}$

```
----------------------------------------------------
 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
------------------------------------------------------
```

Nested "for" loops

As in the case of "while" loops, "for" loops can also be nested, as illustrated in the following example that calculates a double summation:

$$\boxed{S04 = \sum_{j=1}^{5} \sum_{k=1}^{5} (k \cdot j)}$$

```
-----------------------------------------------------
  S04:= 0                // initialize S04

  for j∈1 ..5
    for k∈1 ..5          // nested "for "loops with
      S04:= S04+ k·j     // the same range of i and k:
                         // 'range(1,5)'

  S04= 225               // final value of S04
-----------------------------------------------------
```

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 (“lightest”) 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[1,k], etc. The example shows how to enter sub-indices. The output from the program is the sorted vector rS.

```
----------------------------------------------------------------------
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 in SMath Studio using function "sort":

$$rT := \begin{pmatrix} 5.4 & 1.2 & 3.5 & 10.2 & -2.5 & 4.1 \end{pmatrix}$$

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

Creating a bubble sort function

The bubble-sort algorithm can be turned into a function as follows:

```
mySort(rS):= | nS:= length(rS)
             | for k ∈ 1 .. nS-1
             |   for j ∈ k+1 .. nS
             |     if rS_{1 k} > rS_{1 j}
             |       | temp:= rS_{1 j}
             |       | rS_{1 j}:= rS_{1 k}
             |       | rS_{1 k}:= temp
             |     else
             |       0
             | rS
```

Here is an application of this function:

$rS:=(5.4\ \ 1.2\ \ 3.5\ \ 10.2\ \ -2.5\ \ 4.1)$

$mySort(rS)=(-2.5\ \ 1.2\ \ 3.5\ \ 4.1\ \ 5.4\ \ 10.2)$

The "break" and "continue" statements

These statements are available in the second line of the "Programming" palette.

* The "break" statement provides an earlier way out of a "for" loop if a condition is detected within the loop before all the repetitions of the loop, as required by the index definition, are completed.

* The "continue" statement basically lets the process of a program go through the location of the statement without any action taken.

To illustrate the use of the "break" and "continue" statements, consider the program written here: ----> The variables XS is first initialized as zero, then a for look with index k = 1, 2, ..., 10, activates a "for" loop, in which XS is incremented by 1 in each loop cycle. In this program, however, we have included an "if" statement that will let the control break out of the loop as soon as k > 5. The "if" statement automatically includes an "else" statement, however, we want no action in the case in which k<5 for this "if". Therefore, we placed a "continue" statement for the "else" option.

```
XS:= 0

for k ∈ 1 .. 10
  | XS:= XS+1
  | if k>5
  |   break
  | else
  |   continue

XS= 6
```

NOTE: The example shown above shows a very inefficient loop, since the index is initially defined to run from 1 to 10, but then the "if" statements effectively reduces the index range to the numbers 1 through 4. The purpose of this example was, therefore, only to illustrate the use of the "break" and "continue" statements. A more efficient way to program this calculation, without using "if", "break", or "continue", is shown below:

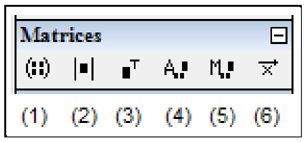
```
XS:= 0

for k ∈ 1 .. 4
  XS:= XS+1

XS= 4
```

Many numerical programs require the use of vectors and matrices, whose operation in SMath Studio, is presented below.

Using the "Matrices" palette in SMath Studio + Matrix operations

Following we show some examples of matrix creation and manipulation using the "Matrices" palette in SMath Studio, as well as other matrix functions. Programming often requires the use of one-dimensional arrays (vectors) and two-dimensional arrays (matrices). Therefore, knowledge of the vector/matrix functions available in SMath Studio is essential for program development.

The "Matrices" palette is shown below:

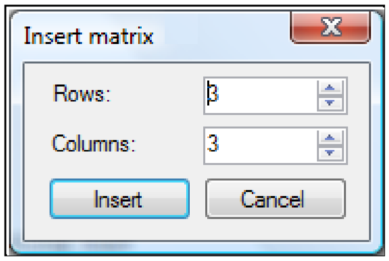


The paletted consists of 6 buttons identified as follows:

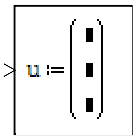
(1) Matrix (Cntl+M): enter a matrix specifying number of rows and columns
(2) Determinant: calculate the determinant of a matrix
(3) Matrix Transpose (Cntl+1): obtain the transpose of a matrix
(4) Algebraic addition to matrix: similar to minor
(5) Minor: calculates the determinant of a matrix minor
(6) Cross product: calculates the cross product of two 3-element vectors

Entering Matrices with the "Matrix" button

Suppose we want to define two 3-element column vectors u and v using the "Matrix" button. First, let's define vector u by clicking in an empty section of the worksheet and typing "u:". Then, press the "Matrix" button in the "Matrices" palette to produce the following entry form -> -> in which, by default, we would enter a matrix with 3 rows and 3 columns.



Change the number of "Columns" to 1 and press the [Insert] button. This produces the result -->



The next step is to enter the components of the vector by clicking on the different placeholders and typing the corresponding entries. The vectors u and v, entered thisway, are shown to the right: ------------------------->

$$u := \begin{pmatrix} 3 \\ 5 \\ -2 \end{pmatrix} \qquad v := \begin{pmatrix} 3 \\ -5 \\ 7 \end{pmatrix}$$

By using the "Matrix" button, or the command "Cntl+M", we can also define other matrices such as the 5x4 matrix A and the 6x6 matrix B shown below.

$$A := \begin{pmatrix} 5 & -2 & 3 & 8 \\ -4 & -7 & 0 & 3 \\ 7 & 2 & 3 & -5 \\ -2 & -8 & 3 & 5 \\ 6 & 2 & -4 & 9 \end{pmatrix} \qquad B := \begin{pmatrix} 2 & -8 & -3 & 5 & -6 & 3 \\ 3 & 1 & -5 & 4 & 4 & 2 \\ 6 & 8 & 6 & -1 & 8 & 8 \\ 9 & -3 & -5 & -6 & 8 & 7 \\ -3 & 8 & 0 & 9 & -7 & 8 \\ 6 & 0 & -7 & 4 & -1 & 2 \end{pmatrix}$$

Calculating the determinant of matrices

To calculate the determinant of a matrix that has been already defined, such as matrices A or B, above, click in the worksheet, then press the "Determinant" button in the "Matrices" palette. Type the name of the matrix in the placeholder and press the equal sign (=) in your keyboard. Here is an example: $\boxed{|B| = -6.19 \cdot 10^{5}}$

Note that an attempt to obtain the determinant of A would fail because determinants are only defined for square matrices such as B (6x6), but not for a rectangular matrix such as A (5x4).

Obtaining the transpose of a matrix

To obtain the transpose of a matrix that has been already define, click on the worksheet, then press the "Matrix Transpose" button in the "Matrices" palette. Type the name of the matrix in the placeholder and press the equal sign (=) in your keyboard. Here are some examples:

$$u^T = \begin{pmatrix} 3 & 5 & -2 \end{pmatrix} \qquad v^T = \begin{pmatrix} 3 & -5 & 7 \end{pmatrix}$$

$$A^T = \begin{pmatrix} 5 & -4 & 7 & -2 & 6 \\ -2 & -7 & 2 & -8 & 2 \\ 3 & 0 & 3 & 3 & -4 \\ 8 & 3 & -5 & 5 & 9 \end{pmatrix} \qquad B^T = \begin{pmatrix} 2 & 3 & 6 & 9 & -3 & 6 \\ -8 & 1 & 8 & -3 & 8 & 0 \\ -3 & -5 & 6 & -5 & 0 & -7 \\ 5 & 4 & -1 & -6 & 9 & 4 \\ -6 & 4 & 8 & 8 & -7 & -1 \\ 3 & 2 & 8 & 7 & 8 & 2 \end{pmatrix}$$

Algebraic addition (4) and Minor (5) buttons

The operation of these two buttons is very similar: Click in the worksheet, press button (4) or button (5), this produces either the symbol A or the symbol M with two sub-index placeholders and one placeholder between parentheses. Enter integer numerical values in the sub-index placeholders, let's call them i and j. Also, enter the name of a square matrix in the placeholder between parentheses, say, B. Symbol A produces the absolute value of the determinant of the minor matrix of B resulting from removing row i and column j. On, the other hand, symbol M produces the determinant of the same minor matrix. For example, for the 6x6 matrix B defined above we have:

$$A_{2\,3}(B) = 22763 \qquad M_{2\,3}(B) = -22763$$

Getting a minor matrix (not in the "Matrices" palette)

To see the minor matrix, use the function "vminor()", available under the "Matrix and vector" category in the function button in the SMath Studio toolbar --> f(x)
This produces a symbol M very similar to that produced with button (5) in the "Matrices" palette, but the result is the minor matrix, rather than the determinant of the matrix, e.g.,

$$M_{2\,3}(B) = \begin{pmatrix} 2 & -8 & 5 & -6 & 3 \\ 6 & 8 & -1 & 8 & 8 \\ 9 & -3 & -6 & 8 & 7 \\ -3 & 8 & 9 & -7 & 8 \\ 6 & 0 & 4 & -1 & 2 \end{pmatrix}$$

Calculating a cross-product

A cross product, or vector product, can be performed only on column vectors of three elements, such as vectors u and v defined above.

To perform a cross product, click somewhere in your worksheet, and press the "Cross product" button in the "Matrices" palette. This will produce two placeholders separated by the multiplication symbol (x). Fill the placeholders with the name of vectors previously defined, or insert vectors using the "Matrix (Cntl+M)" button in the "Matrices" palette, and then enter the equal sign (=). Here are some examples:

$$u \times v = \begin{pmatrix} 25 \\ -27 \\ -30 \end{pmatrix} \qquad v \times u = \begin{pmatrix} -25 \\ 27 \\ 30 \end{pmatrix}$$

Other common matrix operations (not in the "Matrices" palette)

1 - Calculating the INVERSE of a square matrix: type the matrix name and raise it to the power -1, e.g.,

$$B^{-1}=\begin{pmatrix} 0.02 & -0.1 & 0.08 & -0.05 & -0.06 & 0.17 \\ -0.09 & -0.05 & 0 & -0.01 & 0.03 & 0.08 \\ 0.07 & -0.04 & 0.09 & -0.07 & -0.05 & -0.02 \\ 0.07 & 0.13 & 0.05 & -0.08 & -0.03 & -0.02 \\ 0.01 & 0.16 & 0.01 & 0 & -0.03 & -0.11 \\ 0.03 & 0.01 & -0.01 & 0.08 & 0.07 & -0.09 \end{pmatrix}$$

2 - Matrix addition, subtraction, product: simply use the same arithmetic operators (+,-,*) as used with scalars, e.g.,

$$B+B^{T}=\begin{pmatrix} 4 & -5 & 3 & 14 & -9 & 9 \\ -5 & 2 & 3 & 1 & 12 & 2 \\ 3 & 3 & 12 & -6 & 8 & 1 \\ 14 & 1 & -6 & -12 & 17 & 11 \\ -9 & 12 & 8 & 17 & -14 & 7 \\ 9 & 2 & 1 & 11 & 7 & 4 \end{pmatrix} \qquad B^{T}-B=\begin{pmatrix} 0 & 11 & 9 & 4 & 3 & 3 \\ -11 & 0 & 13 & -7 & 4 & -2 \\ -9 & -13 & 0 & -4 & -8 & -15 \\ -4 & 7 & 4 & 0 & 1 & -3 \\ -3 & -4 & 8 & -1 & 0 & -9 \\ -3 & 2 & 15 & 3 & 9 & 0 \end{pmatrix}$$

$$B\cdot B^{T}=\begin{pmatrix} 147 & 15 & -99 & 0 & 41 & 65 \\ 15 & 71 & 40 & 71 & 23 & 69 \\ -99 & 40 & 265 & 126 & 45 & -2 \\ 0 & 71 & 126 & 264 & -105 & 71 \\ 41 & 23 & 45 & -105 & 267 & 41 \\ 65 & 69 & -2 & 71 & 41 & 106 \end{pmatrix} \qquad B\cdot B^{-1}=\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$C:=3\cdot B-5\cdot B^{T}$, or,

$$C=\begin{pmatrix} -4 & -39 & -39 & -30 & -3 & -21 \\ 49 & -2 & -55 & 27 & -28 & 6 \\ 33 & 49 & -12 & 22 & 24 & 59 \\ 2 & -29 & -10 & 12 & -21 & 1 \\ 21 & 4 & -40 & -13 & 14 & 29 \\ 3 & -10 & -61 & -23 & -43 & -4 \end{pmatrix}$$

3 - Creating specific types of matrices: use the following functions in the "Matrix and vector" option under the functions button:--------------------------> f(x)

diag(v): produces a matrix whose main diagonal elements are the components of column vector v, while all off-diagonal elements are zero, e.g.,

$$\operatorname{diag}(u)=\begin{pmatrix} 3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & -2 \end{pmatrix} \qquad \operatorname{diag}(v)=\begin{pmatrix} 3 & 0 & 0 \\ 0 & -5 & 0 \\ 0 & 0 & 7 \end{pmatrix}$$

identity(n): produces an identity matrix of order nxn, e.g.,

$$\operatorname{identity}(3)=\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \operatorname{identity}(4)=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

matrix(n,m): produces a matrix of n rows and m columns with all its elements equal to zero, e.g.,

$$\mathrm{matrix}(3,4)=\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\qquad \mathrm{matrix}(4,4)=\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}$$

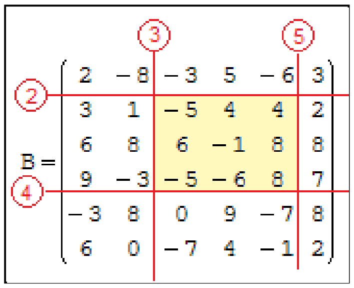
reverse(matrix): reverses the order of rows in matrices or vectors, e.g.,

$$u=\begin{pmatrix}3\\5\\-2\end{pmatrix}\qquad \mathrm{reverse}(u)=\begin{pmatrix}-2\\5\\3\end{pmatrix}$$

$$B=\begin{pmatrix}2&-8&-3&5&-6&3\\3&1&-5&4&4&2\\6&8&6&-1&8&8\\9&-3&-5&-6&8&7\\-3&8&0&9&-7&8\\6&0&-7&4&-1&2\end{pmatrix}\qquad \mathrm{reverse}(B)=\begin{pmatrix}6&0&-7&4&-1&2\\-3&8&0&9&-7&8\\9&-3&-5&-6&8&7\\6&8&6&-1&8&8\\3&1&-5&4&4&2\\2&-8&-3&5&-6&3\end{pmatrix}$$

submatrix(matrix,is,ie,js,je): extracts a submatrix of "matrix" including rows is to ie, and columns js to je, e.g.,

$$\mathrm{submatrix}(B,2,4,3,5)=\begin{pmatrix}-5&4&4\\6&-1&8\\-5&-6&8\end{pmatrix}$$,i.e.,



col(matrix,j): extracts column j of "matrix" as a column vector, e.g.,

$$\mathrm{col}(A,3)=\begin{pmatrix}3\\0\\3\\3\\-4\end{pmatrix}\qquad \mathrm{col}(B,2)=\begin{pmatrix}-8\\1\\8\\-3\\8\\0\end{pmatrix}$$

row(matrix,i): extracts row i of "matrix" as a row vector, e.g.,

$$\mathrm{row}(A,2)=\begin{pmatrix}-4&-7&0&3\end{pmatrix}\qquad \mathrm{row}(B,3)=\begin{pmatrix}6&8&6&-1&8&8\end{pmatrix}$$

augment(m1,m2): creates a new matrix by juxtapositioning matrices m1 and m2 by columns. Matrices m1 and m2 must have the same number of rows, e.g.,

$$u=\begin{pmatrix}3\\5\\-2\end{pmatrix}\quad v=\begin{pmatrix}3\\-5\\7\end{pmatrix}\quad \mathrm{augment}(u,v)=\begin{pmatrix}3&3\\5&-5\\-2&7\end{pmatrix}$$

stack(m1,m2): creates a new matrix by juxtapositioning matrices m1 and m2 by rows. Matrices m1 and m2 must have the same number of columns, e.g.

$$u^{T}=\begin{pmatrix} 3 & 5 & -2 \end{pmatrix} \qquad v^{T}=\begin{pmatrix} 3 & -5 & 7 \end{pmatrix} \qquad \mathrm{stack}\left(u^{T}, v^{T}\right)=\begin{pmatrix} 3 & 5 & -2 \\ 3 & -5 & 7 \end{pmatrix}$$

4 - Functions that characterize matrices: these are functions that produce numbers that represent some characteristics of a matrix. One such number is the determinant of a matrix, which can be calculated using button (2) in the "Matrices" palette, or using function "det."

The following are functions of interest, available under the "Matrix and vector" option under the function button in the SMath Studio toolbar:

f(x)

* cols(matrix): determines the number of columns in a matrix
* det(matrix): calculates the determinant of "matrix"
* el(matrix,i,j): extracts element i,j from "matrix"
* length(matrix): determines the number of elements in "matrix" (or vector)
* max(matrix): determines the maximum value in a matrix (or vector)
* min(matrix): determines the minimum value in a matrix (or vector)
* norm1(matrix): determines the L1 norm of "matrix"
* norme(matrix): determines the Euclidean norm of "matrix"
* normi(matrix): determines the infinite norm of "matrix"
* rank(matrix): determines the rank of "matrix"
* rows(matrix): determines the number of rows of "matrix
* trace(matrix): determines the trace (sum of diagonal elements) of a square "matrix"

Examples of these functions are shown below:

$\mathrm{cols}(A)=4$ $\qquad$ $\mathrm{rows}(A)=5$ $\qquad$ $B_{2\,3}=-5$ <-- this is "el(B,2,3)"

$\mathrm{length}(B)=36$ $\qquad$ $\max(B)=9$ $\qquad$ $\min(B)=-8$

$\mathrm{norm1}(A)=30$ $\qquad$ $\mathrm{norme}(A)=$ ▮ $\qquad$ $\mathrm{normi}(A)=21$

$\mathrm{rank}(A)=4$ $\qquad$ $\mathrm{tr}(B)=-2$

5- Operations that involve sorting by columns, rows, or sorting a vector: These operations are also available under the "Matrix and vector" option of the function button -> f(x)

csort(matrix,j): sorts elements of column j in ascending order while dragging along the elements in the other columns, e.g., sorting by column 3 in matrix B:

$$B=\begin{pmatrix} 2 & -8 & -3 & 5 & -6 & 3 \\ 3 & 1 & -5 & 4 & 4 & 2 \\ 6 & 8 & 6 & -1 & 8 & 8 \\ 9 & -3 & -5 & -6 & 8 & 7 \\ -3 & 8 & 0 & 9 & -7 & 8 \\ 6 & 0 & -7 & 4 & -1 & 2 \end{pmatrix} \qquad \mathrm{csort}(B, 3)=\begin{pmatrix} 6 & 0 & -7 & 4 & -1 & 2 \\ 3 & 1 & -5 & 4 & 4 & 2 \\ 9 & -3 & -5 & -6 & 8 & 7 \\ 2 & -8 & -3 & 5 & -6 & 3 \\ -3 & 8 & 0 & 9 & -7 & 8 \\ 6 & 8 & 6 & -1 & 8 & 8 \end{pmatrix}$$

rsort(matrix,i): sorts elements of row i in ascending order while dragging along the elements in the other rows, e.g., sorting by row 4 in matrix B:

$$B=\begin{pmatrix} 2 & -8 & -3 & 5 & -6 & 3 \\ 3 & 1 & -5 & 4 & 4 & 2 \\ 6 & 8 & 6 & -1 & 8 & 8 \\ 9 & -3 & -5 & -6 & 8 & 7 \\ -3 & 8 & 0 & 9 & -7 & 8 \\ 6 & 0 & -7 & 4 & -1 & 2 \end{pmatrix} \qquad \mathrm{rsort}(B\,,\,4)=\begin{pmatrix} 5 & -3 & -8 & 3 & -6 & 2 \\ 4 & -5 & 1 & 2 & 4 & 3 \\ -1 & 6 & 8 & 8 & 8 & 6 \\ -6 & -5 & -3 & 7 & 8 & 9 \\ 9 & 0 & 8 & 8 & -7 & -3 \\ 4 & -7 & 0 & 2 & -1 & 6 \end{pmatrix}$$

sort(vector): sorts elements of a column vector in ascending order, e.g.,

$$v=\begin{pmatrix} 3 \\ -5 \\ 7 \end{pmatrix} \qquad \mathrm{sort}(v)=\begin{pmatrix} -5 \\ 3 \\ 7 \end{pmatrix} \qquad u=\begin{pmatrix} 3 \\ 5 \\ -2 \end{pmatrix} \qquad \mathrm{sort}(u)=\begin{pmatrix} -2 \\ 3 \\ 5 \end{pmatrix}$$

Typical programming steps

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 user when running the program.

(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.

PROGRAMMING EXAMPLE No. 1 - The Newton-Raphson method for solving $f(x)=0$

The Newton-Raphson method used for solving an equation of the form $f(x)=0$ requires requires the knowledge of the derivative f'(x). This can be easily accomplished in SMath Studio using the "Derivative" option in the "Functions" palette:

$$fp(x)=\frac{d}{dx}f(x)$$

Given an initial guess of the solution, $x=x_0$, the solution can be approximated by the iterative calculation:

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)}$$

for $k=0$,1, ...

The iteration continues until either the solution converges, i.e., $\left|f\left(x_{k+1}\right)\right|<\varepsilon$, or a certain large number of iterations are performed without convergence, i.e., $k>n_{max}$

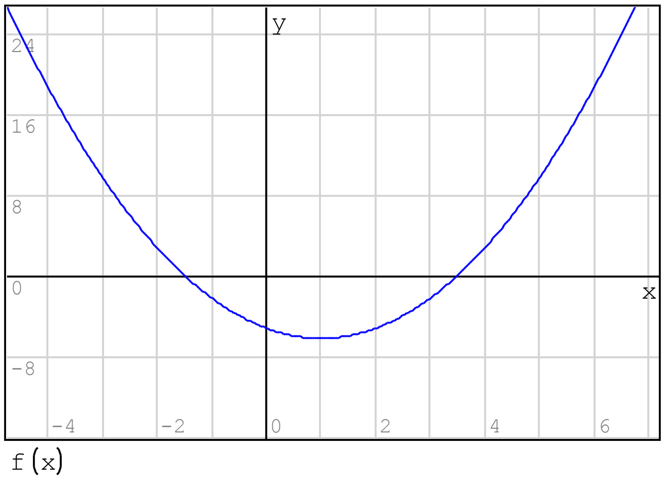
Example: Solve the equation: $x^2-2\cdot x-5=0$

Solution: A graph of the function can help us find where the solutions may be located:

Define the function:

$$f(x) := x^2 - 2 \cdot x - 5$$

Produce a graph of f(x):



The graph shows solutions near x = -2 and x = 3. We can implement the solution using the Newton-Raphson method as follows:

$$fp(x) := \frac{d}{dx} f(x) \qquad fp(x) = -10$$

Parameters of the solution are: $\varepsilon := 1.0 \cdot 10^{-6}$ $\quad nmax := 100$

Calculating a solution:
Starting with a guess of $xG := -2.5$ we find a solution by using the following iterative procedure:

```
k := 0
while ((k ≤ nmax) ∧ (|f(xG)| > ε))
    xGp1 := xG - $\frac{f(xG)}{fp(xG)}$
    k := k + 1
    xG := xGp1
```

$xG = -1.45$ This is the solution found

$k = 4$ After this many iterations

$f(xG) = 2.14 \cdot 10^{-11}$ The function at the solution point

Putting together a Newton-Raphson program

The steps listed above to calculate the solution to the equation f(x) = 0 can be put together into a function as follows:

```
fNewton(f, x0, ε, Nmax) :=
    fp(x) := $\frac{d}{dx} f(x)$
    xG := x0
    k := 0
    while ((k ≤ nmax) ∧ (|f(xG)| > ε))
        xGp1 := xG − $\frac{f(xG)}{fp(xG)}$
        k := k + 1
        xG := xGp1
    $\begin{pmatrix} xG \\ f(xG) \\ k \end{pmatrix}$
```

NOTE: The output is given as a column vector of 3 elements: (1) the solution, xG, (2) the function at the solution, f(xG), and the number of iterations needed to converge to a solution, k.

Solving for a particular case:

$ff(x) := x^2 - 2 \cdot x - 5 \quad \varepsilon := 1.0 \cdot 10^{-6} \quad Nmax := 100$

Solution 1:

$x0 := 2.5$

$$fNewton(ff(x), x0, \varepsilon, Nmax) = \begin{pmatrix} 3.45 \\ 2.99 \cdot 10^{-9} \\ 4 \end{pmatrix}$$

Solution 2:

$x0 := -2.2$

$$fNewton(ff(x), x0, \varepsilon, Nmax) = \begin{pmatrix} 3.45 \\ 2.99 \cdot 10^{-9} \\ 4 \end{pmatrix}$$

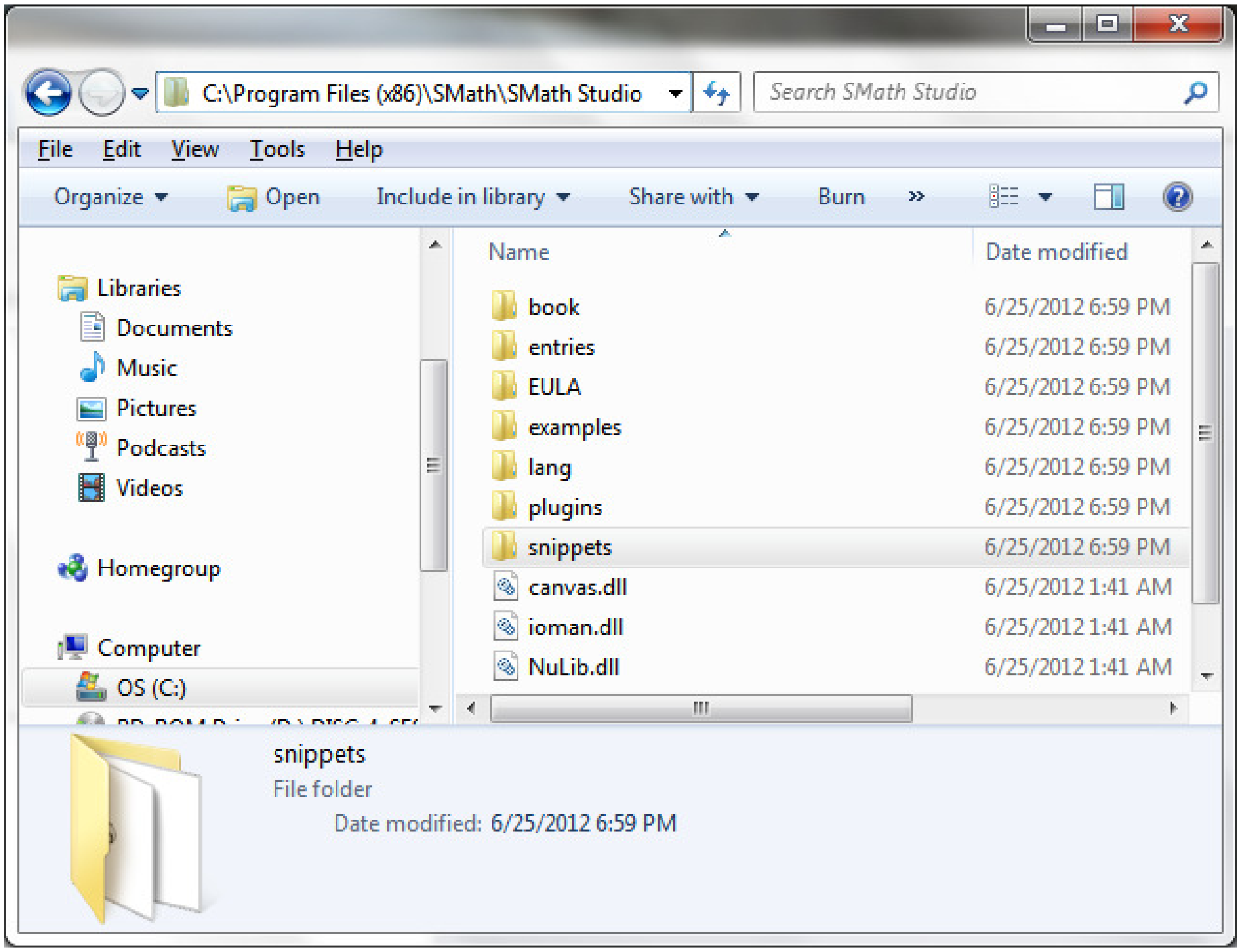
Compare with solution given by function "solve":

$solve(ff(x) = 0, x, -5, 0) = -1.45$

$solve(ff(x) = 0, x, 0, 5) = 3.45$

Using code snippets

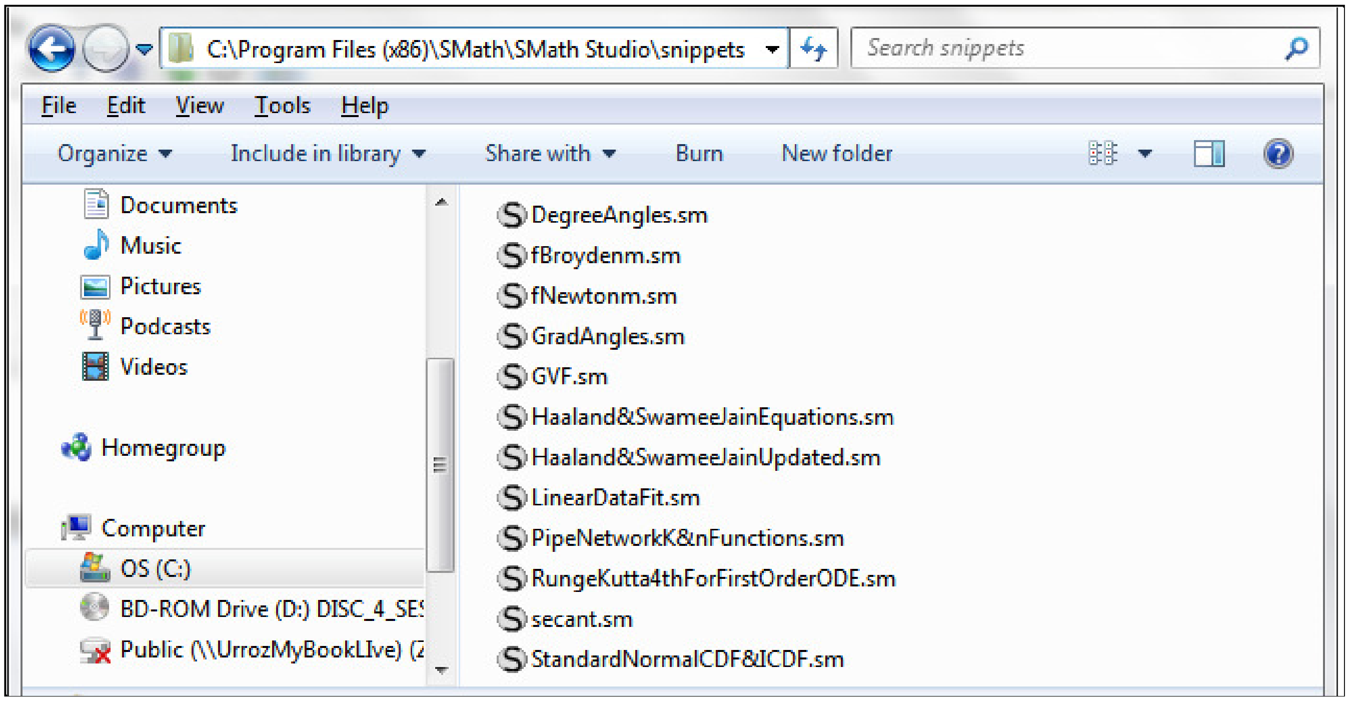
A code snippet is simply an user-defined function defined in a SMath Studio file and placed in the "snippets" folder in the SMath Studio installation in your computer. For example, the figure below shows the location of the "snippets" folder in a Windows 7 installation.


C:\Program Files (x86)\SMath\SMath Studio
Search SMath Studio
File Edit View Tools Help
Organize Open Include in library Share with Burn
Libraries
Documents
Music
Pictures
Podcasts
Videos
Homegroup
Computer
OS (C:)

Name	Date modified
book	6/25/2012 6:59 PM
entries	6/25/2012 6:59 PM
EULA	6/25/2012 6:59 PM
examples	6/25/2012 6:59 PM
lang	6/25/2012 6:59 PM
plugins	6/25/2012 6:59 PM
snippets	6/25/2012 6:59 PM
canvas.dll	6/25/2012 1:41 AM
ioman.dll	6/25/2012 1:41 AM
NuLib.dll	6/25/2012 1:41 AM

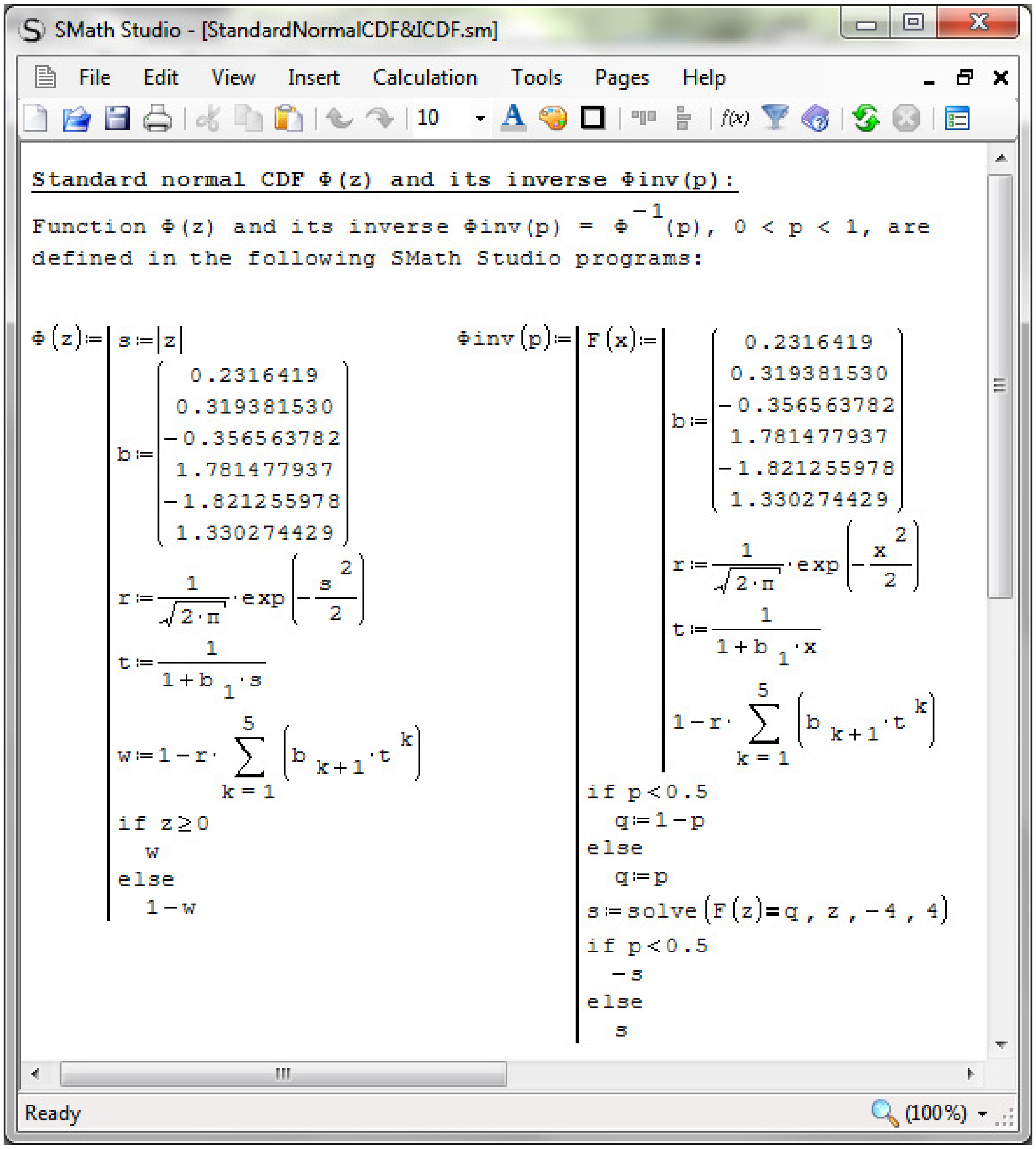
snippets
File folder
Date modified: 6/25/2012 6:59 PM

In my computer, for example, I have the following code snippets that I use for my courses:


C:\Program Files (x86)\SMath\SMath Studio\snippets
Search snippets
File Edit View Tools Help
Organize Include in library Share with Burn New folder
Documents
Music
Pictures
Podcasts
Videos
Homegroup
Computer
OS (C:)
BD-ROM Drive (D:) DISC_4_SE
Public (\\UrrozMyBookLIve) (Z

DegreeAngles.sm
fBroydenm.sm
fNewtonm.sm
GradAngles.sm
GVF.sm
Haaland&SwameeJainEquations.sm
Haaland&SwameeJainUpdated.sm
LinearDataFit.sm
PipeNetworkK&nFunctions.sm
RungeKutta4thForFirstOrderODE.sm
secant.sm
StandardNormalCDF&ICDF.sm

The following figure shows an SMath Studio file entitled "StandardNormalCDF&ICDF.sm", showing programs used to estimate the Cumulative Distribution Function (CDF) and the Inverse Cumulative Distribution Function (ICDF) of the Standard Normal distribution. This is an example of a code snipped defining two functions Φ(z), the Standard Normal CDF, and Φinv(p), the Standard Normal ICDF.


SMath Studio - [StandardNormalCDF&ICDF.sm]
File Edit View Insert Calculation Tools Pages Help
Ready
(100%)

Standard normal CDF Φ(z) and its inverse Φinv(p):

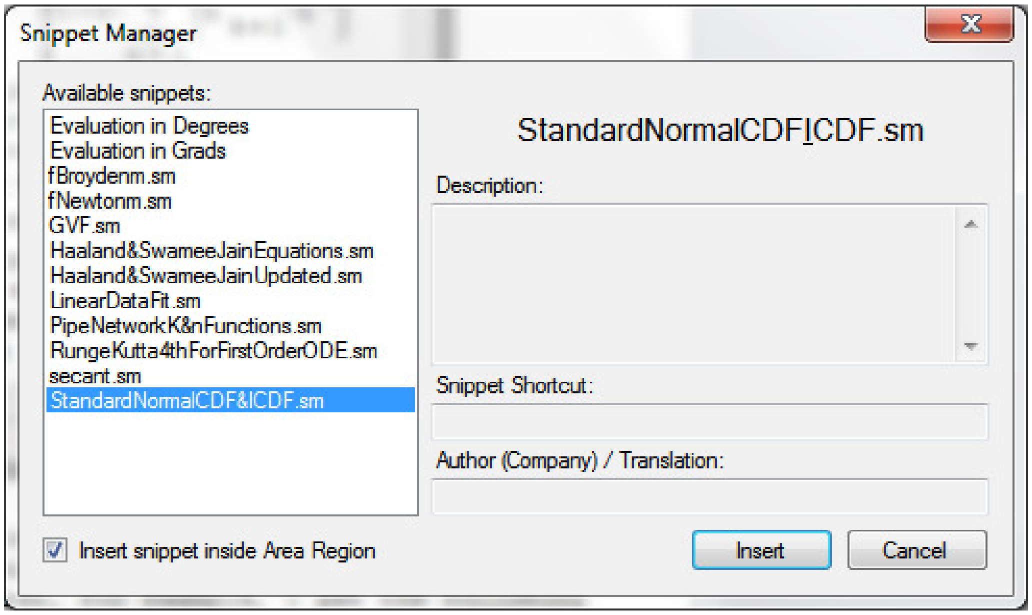
Function Φ(z) and its inverse Φinv(p) = $\Phi^{-1}(p)$, 0 < p < 1, are defined in the following SMath Studio programs:

```
Φ(z) := | s := |z|
        | b := [0.2316419; 0.319381530; -0.356563782; 1.781477937; -1.821255978; 1.330274429]
        | r := 1/√(2·π) · exp(-s^2/2)
        | t := 1/(1 + b_1·s)
        | w := 1 - r · Σ_{k=1}^{5} (b_{k+1}·t^k)
        | if z ≥ 0
        |   w
        | else
        |   1 - w

Φinv(p) := | F(x) := | b := [0.2316419; 0.319381530; -0.356563782; 1.781477937; -1.821255978; 1.330274429]
           |         | r := 1/√(2·π) · exp(-x^2/2)
           |         | t := 1/(1 + b_1·x)
           |         | 1 - r · Σ_{k=1}^{5} (b_{k+1}·t^k)
           | if p < 0.5
           |   q := 1 - p
           | else
           |   q := p
           | s := solve(F(z) = q, z, -4, 4)
           | if p < 0.5
           |   -s
           | else
           |   s
```

S

Code snippets can be inserted into an SMath Studio file by first clicking on an empty location in the workbook, and then using the option: "Tools > Snippet Manager" in the SMath Studio toolbar.

This opens up a list of all the code snippets available in your computer. In my computer, for example, I get the following list:


Snippet Manager
Available snippets:
Evaluation in Degrees
Evaluation in Grads
fBroydenm.sm
fNewtonm.sm
GVF.sm
Haaland&SwameeJainEquations.sm
Haaland&SwameeJainUpdated.sm
LinearDataFit.sm
PipeNetworkK&nFunctions.sm
RungeKutta4thForFirstOrderODE.sm
secant.sm
StandardNormalCDF&ICDF.sm
StandardNormalCDFICDF.sm
Description:
Snippet Shortcut:
Author (Company) / Translation:
Insert snippet inside Area Region
Insert
Cancel

Once you select the code snippet that you want to insert, simply press the [Insert] button.

For example, following I insert the "StandardNormalCDF&ICDF.sm"code snippet:

⊞— StandardNormalCDF&ICDF.sm

The result is a collapsed area named after the code snipped file. In this case, it is "StandardNormalCDF&ICDF.sm". The [+] symbol located to the left of the code snippet name indicates that the area containing the code snippet is collapsed. If you click on the [+] symbol, the collapsed area opens up showing the contents of the code snippet.

After you have inserted a code snippet in your worksheet you can use the functions defined in the code snippet for calculations at any location below the insertion point. For example, the following are calculations performed using functions Φ(z) and Φinv(p):

$\Phi(1.2)=0.8849$ $\Phi(-1.2)=0.1151$ $\Phi(1.2)+\Phi(-1.2)=1$

$\Phi inv(0.6)=0.2533$ $\Phi inv(0.2)=-0.8416$ $\Phi inv(0.86)=1.0803$

PROGRAMMING EXAMPLE No.2 - Generalized Newton-Raphson method for a system of equations

Given the system of equations: $f1(x) = 0$, $f2(x) = 0$, $\dots$, $fn(x) = 0$ with $x = \begin{pmatrix} x_1 \\ x_2 \\ . \\ x_n \end{pmatrix}$

we set up the vector equation: $f(x) = \begin{pmatrix} f1(x) \\ f2(x) \\ . \\ fn(x) \end{pmatrix}$

Let $x0 = \begin{pmatrix} x0_1 \\ x0_2 \\ \blacksquare \\ x0_n \end{pmatrix}$ be an initial guess for the solution.

The generalized Newton-Raphson method indicates that we can calculate a better approximation for the solution by using:

$$\boxed{x_{k+1} = x_k - J^{-1} \cdot f(x_k)} \quad , k = 0, 1, 2, \dots$$

where J is the Jacobian of the vector function, i.e., $\boxed{J_{i\,j} = \frac{d}{d\,x_j} fi}$

The Jacobian can be calculated using SMath Studio function "Jacob."

To check convergence we use the criteria:

$$\text{norme}\left(f\left(x_{k+1}\right)\right) \le \varepsilon$$

where function "norme" calculates the Euclidean norm of the function.

Also, we check for divergence by keeping the iterations to a maximum value "Nmax."

The following function "fNewtonm" performs the multi-variate Newton- Raphson calculation:

```
fNewtonm(f(x), xs, x0, ε, Nmax) :=
    fJ(x) := Jacob(f(x), xs)
    k := 0
    xG := x0
    while ((k ≤ Nmax) ∧ (norme(f(xG)) > ε))
        JJ := eval(fJ(xG))
        JJI := eval(invert(JJ))
        fxG := eval(f(xG))
        DxG := eval(JJI · fxG)
        xGp1 := eval(xG - DxG)
        k := k + 1
        xG := xGp1
    (xG)
    ( k)
```

In here, f(x) is a column vector function whose elements correspond to the different non-linear equations being solved, and xs is a column vector listing the variables involved in the equations. Function "fNewtonm" uses function "Jacob", which calculates the Jacobian matrix for function f(x).

x0 is an intial guess to the solution, ε is the tolerance for convergence, and "Nmax" is the maximum number of iterations allowed before cancelling solution.

Solution is a vector: $\begin{pmatrix} xG \\ k \end{pmatrix} = \begin{pmatrix} \text{solution} \\ \text{Number\_of\_iterations} \end{pmatrix}$

EXAMPLE of "fNewtonm" application to a system of non-linear equations

Solving the following pipeline-junction system of equations

$$280 - HJ = 0.0603 \cdot Q1 \cdot |Q1|$$
$$290 - HJ = 0.0203 \cdot Q2 \cdot |Q2|$$
$$150 - HJ = 0.0543 \cdot Q3 \cdot |Q3|$$
$$Q1 + Q2 + Q3 - 80 = 0$$

Change variables to: $x_1 = Q1$ $x_2 = Q2$ $x_3 = Q3$ $x_4 = HJ$

Re-write equationt to produce the following column vector function, fK, and the variable list vector, xK:

$$fK(x) := \begin{pmatrix} 280 - x_4 - 0.0603 \cdot x_1 \cdot |x_1| \\ 290 - x_4 - 0.0203 \cdot x_2 \cdot |x_2| \\ 150 - x_4 - 0.0543 \cdot x_3 \cdot |x_3| \\ x_1 + x_2 + x_3 - 80 \end{pmatrix} \qquad xK := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Using the following values of ε and Nmax: $\varepsilon := 1 \cdot 10^{-10}$ $Nmax := 20$

and the initial guess: $x0 := \begin{pmatrix} 0.5 \\ 0.4 \\ 0.3 \\ 200 \end{pmatrix}$ we find a solution as follows:

$$XSol := fNewtonm(fK(x), xK, x0, \varepsilon, Nmax)$$

Extracting the solutions for x:

$$XSol = \begin{pmatrix} \begin{pmatrix} 0.5 \\ 0.4 \\ 0.3 \\ 200 \end{pmatrix} \\ 0 \end{pmatrix}$$

$xsol := XSol_1$ $\qquad xsol = \begin{pmatrix} 0.5 \\ 0.4 \\ 0.3 \\ 200 \end{pmatrix}$

$$fK(xsol) = \begin{pmatrix} 79.98 \\ 90 \\ -50 \\ -78.8 \end{pmatrix}$$

$$fNewtonm(fK(x), xK, x0, \varepsilon, Nmax) = \begin{pmatrix} \begin{pmatrix} 0.5 \\ 0.4 \\ 0.3 \\ 200 \end{pmatrix} \\ 0 \end{pmatrix}$$

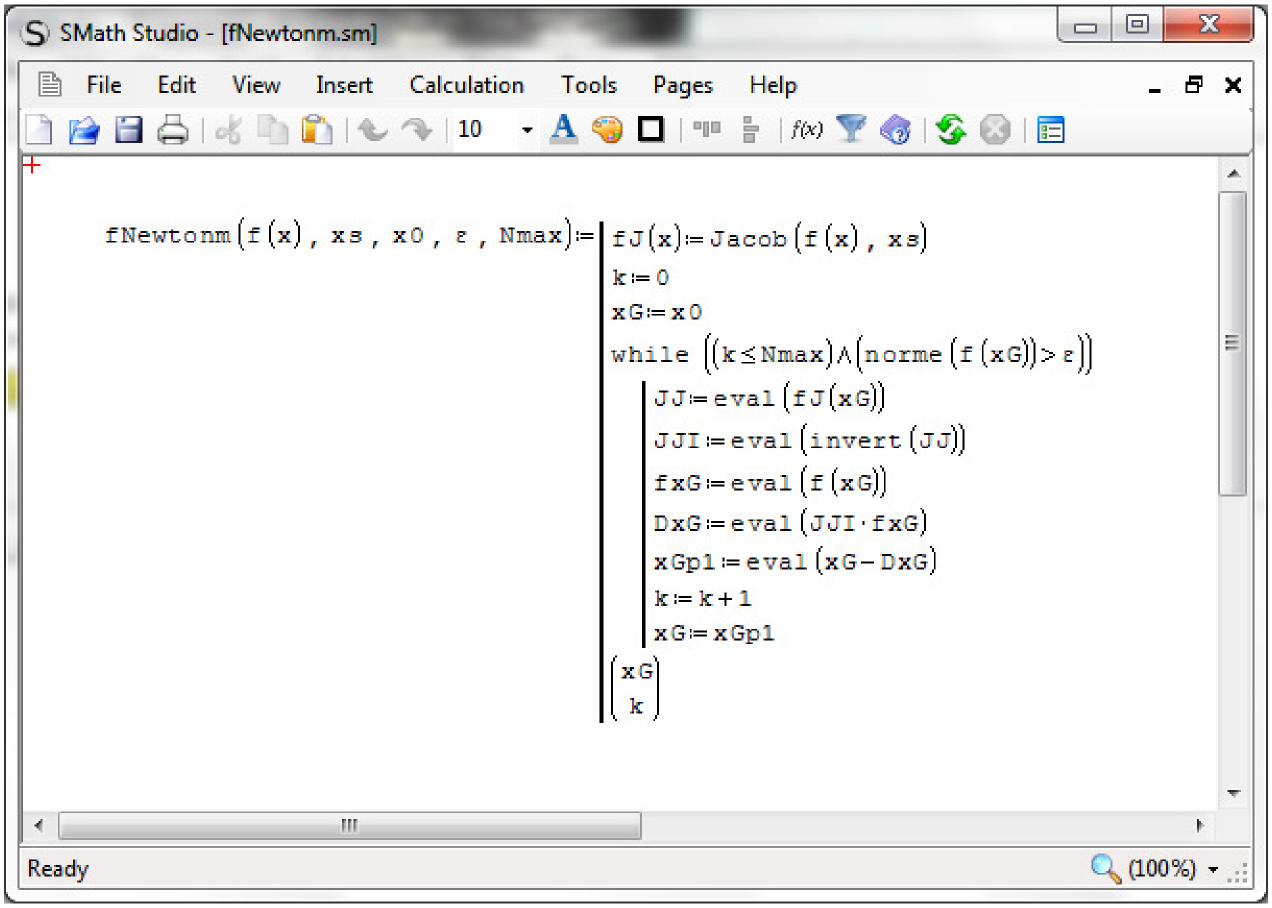
and,
$Q1 := xsol_1$ $Q2 := xsol_2$ $Q3 := xsol_3$ $HJ := xsol_4$

$Q1 = 0.5$ (cfs) $Q2 = 0.4$ (cfs) $Q3 = 0.3$ (cfs)

and, $HJ := xsol_4$, i.e., $HJ = 200$ (ft)

Code snippet for the "fNewtonm" function

The code snipped "fNewtonm.sm", shown below, contains the function 'fNewtonm', defined earlier:


SMath Studio - [fNewtonm.sm]
File Edit View Insert Calculation Tools Pages Help
10
Ready
(100%)

```
fNewtonm(f(x), xs, x0, ε, Nmax):= | fJ(x):= Jacob(f(x), xs)
                                  | k:= 0
                                  | xG:= x0
                                  | while ((k ≤ Nmax)∧(norme(f(xG)) > ε))
                                  |     JJ:= eval(fJ(xG))
                                  |     JJI:= eval(invert(JJ))
                                  |     fxG:= eval(f(xG))
                                  |     DxG:= eval(JJI·fxG)
                                  |     xGp1:= eval(xG − DxG)
                                  |     k:= k + 1
                                  |     xG:= xGp1
                                  | (xG)
                                  | (k )
```

This code snippet is available in my "snippets" folder.

Next, we insert the code snippet "fNewtonm.sm" and repeat the solution of the system of non-linear equations proposed earlier:

⊞— fNewtonm.sm

XSol:= fNewtonm(fK(x), xK, x0, ε, Nmax)

$$XSol=\begin{pmatrix}\begin{pmatrix}0.5\\0.4\\0.3\\200\end{pmatrix}\\0\end{pmatrix}$$

Extracting the solutions for x:

$xsol:= XSol_1$

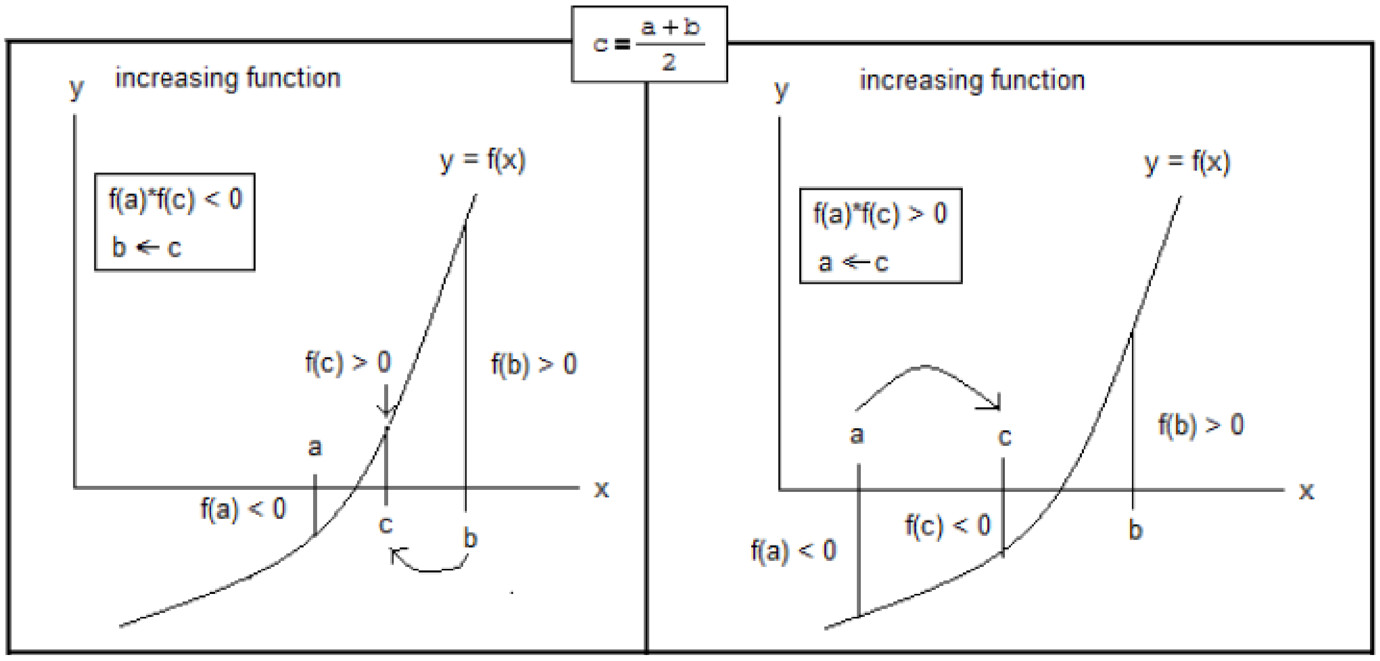
$$\boxed{xsol=\begin{pmatrix}0.5\\0.4\\0.3\\200\end{pmatrix}}$$

$$fK(xsol)=\begin{pmatrix}79.98\\90\\-50\\-78.8\end{pmatrix}$$

The solution is contained in the "xsol" vector shown above. The data for fK(x), xK, x0, ε, and Nmax, were defined earlier.

Programming Example No. 3 - The Bisection Method for solving equations

The solution to the equation f(x) = 0 is the point where the graph of y = f(x) crosses the x axis. The figure below illustrates such situation for the case of an increasing function near the solution. To "bracket" the solution we first identify a couple of values a and b, such that a<b and, in this case, f(a)<0 and f(b)>0. In such a situation we know that the solution, x, is located between a and b, i.e., a < x < b.



As a first approximation to the solution we calculate the midpoint of the interval [a,b], i.e., c = (a+b)/2. The figure above suggest two possibilities:

(i) f(c) > 0, in this case we can bracket the solution into a smaller interval by making b = c, and calculating a new c = (a+b)/2.

(ii) f(c) < 0, in this case we can bracket the solution into a smaller interval by making a = c, and calculating a new c = (a+b)/2.

We then repeat the process as detailed above until we find a value of c for which |f(c)| < ε, where ε is a small number (i.e., f(c) is as close to zero as we want it). Alternatively, we can stop the process after the number of iterations, say, k, exceeds a given value, say, Nmax.

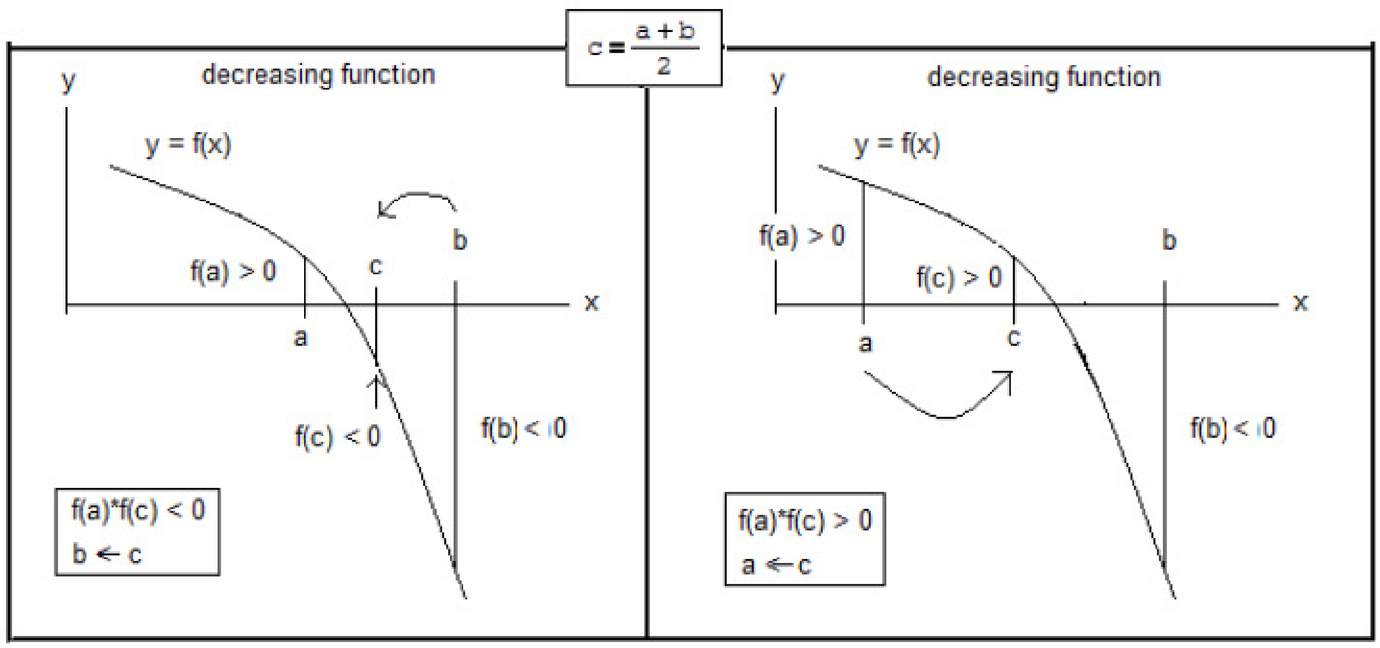
The process described above is a first draft of an algorithm for the so-called Bisection Method for solving a single equation of the form f(x) = 0, for the case in which y = f(x) is an increasing function.

For the case of a decreasing function, the figure below indicates that a solution can be bracketed for a<b if f(a)>0 and f(b)<0. The mid point of the interval [a,b] is calculated as before, namely, c = (a+b)/2. For this case, the figure suggests also two possibilities:

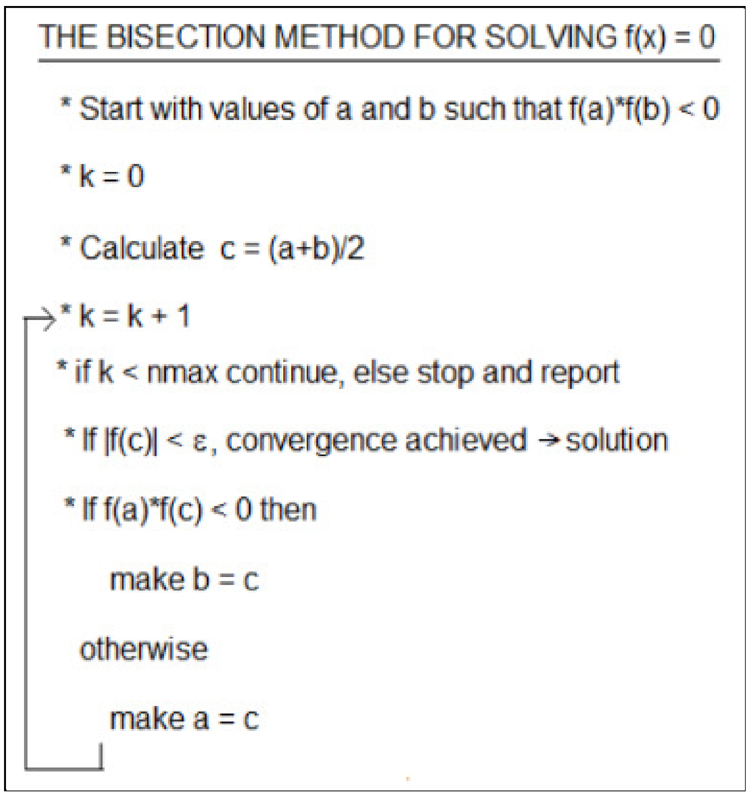
(i) f(c) < 0, in this case we can bracket the solution into a smaller interval by making b = c, and calculating a new c = (a+b)/2.

(ii) f(c) > 0, in this case we can bracket the solution into a smaller interval by making a = c, and calculating a new c = (a+b)/2.

Notice that, for both cases, the solution is in the interval [a,b] if f(a)*f(b) < 0, i.e., as long as f(a) and f(b) have opposite signs. The value of c = (a+b)/2, then replaces a or b depending on whether f(a)*f(c) > 0 or f(a)*f(c) < 0. A new value of c = (a+b)/2 is then calculated, and the process continued until |f(c)|<ε, or until k>Nmax.



The figure below summarizes the algorithm of the Bisection Method for solving equations of the form f(x) = 0, based on the information obtained earlier for both increasing and decreasing functions.



THE BISECTION METHOD FOR SOLVING f(x) = 0

```
 * Start with values of a and b such that f(a)*f(b) < 0
 * k = 0
 * Calculate  c = (a+b)/2
→* k = k + 1
 * if k < nmax continue, else stop and report
 * If |f(c)| < ε, convergence achieved → solution
 * If f(a)*f(c) < 0 then
      make b = c
   otherwise
      make a = c
```

An implementation of this algorithm is shown below. In this program we use "strings" to report a couple of non-solving outcomes:

(i) If the solution is not within [a,b], the initial guesses a and b are wrong, therefore, the program indicates that the product "f(a)*f(b) must be < 0"
(ii) If there is no convergence after Nmax iterations, the program reports "no convergence"

```
Bisect(f, a, b, ε, Nmax) := if f(a)·f(b) > 0
                                "f(a)*f(b) must be < 0"
                            else
                                k := 0
                                c := $\frac{a+b}{2}$
                                while (k < Nmax) ∧ (|f(c)| > ε)
                                    c := $\frac{a+b}{2}$
                                    k := k + 1
                                    if f(a)·f(c) < 0
                                        b := c
                                    else
                                        a := c
                                if k > Nmax
                                    "no convergence"
                                else
                                    c
```

Next we test the program for the function: $f(x) := x^3 - 4\cdot x^2 - 27\cdot x + 85$

with parameters: $\varepsilon := 10^{-5}$ $\quad$ $Nmax := 10$

Various selections of a and b are shown below:

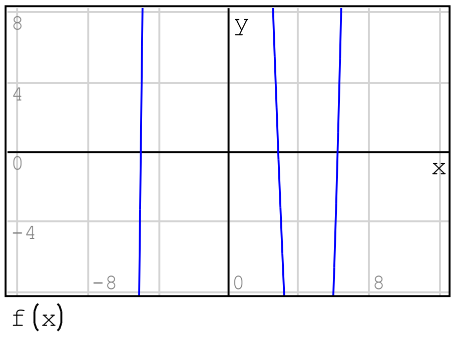
$a := -8$ $\quad$ $b := -5$ $\quad$ Bisect(f, a, b, ε, Nmax) = "f(a)*f(b) must be < 0"

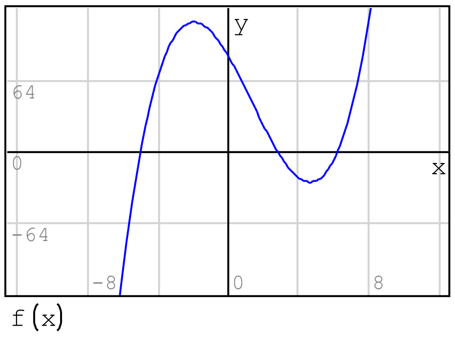
$a := -8$ $\quad$ $b := -2$ $\quad$ Bisect(f, a, b, ε, Nmax) = -4.9473

$a := 1$ $\quad$ $b := 4$ $\quad$ Bisect(f, a, b, ε, Nmax) = 2.8018

$a := 4$ $\quad$ $b := 8$ $\quad$ Bisect(f, a, b, ε, Nmax) = 6.1445

A plot of the function f(x) used in this case is shown below. To produce this graph, click in a location in your worksheet, then use "Insert>Plot>2D". In the placeholder on the lower left corner of the resulting graph type "f(x)". At first you get the graph to the left:





The figure shows the three locations where the graph (blue line) crosses the x axis. These are the three values found above (-4.9473, 2.8018, and 6.1445). You can adjust the vertical scale by fist clicking on the "Scale" button (second button) in the "Plot" palette, then click on the graph, and hold the "Cntl" key, and use the wheel in your mouse. Roll the wheel down to increase the range in the vertical scale. The graph to the right, above, shows the result of adjusting the vertical scale. NOTE: To adjust the horizontal scale, use a similar procedure, but hold down the "Shift" key instead of the "Cntl" key.