MatDeck Programing

1. General

MatDeck allows the integration of text editing, script language, the ability to generate GUIs, flowcharts, virtual instrumentation, data visualization, programing, and parallel processing on multiple computers all to be comprehensively done within its documents. Expressing programming ideas in MatDeck takes usually less lines of code compared to others. The main objectives of this manual are: give general tips and suggestions about how to program in MatDeck, teach enough MatDeck script that it is easy to do most common data manipulations, analyzing, and comparing, as well as to a provide firm knowledge foundation so that learning more advanced MatDeck techniques is possible.

2. Editing code

There are three ways to edit script code and programming in MatDeck: using Math objects within a canvas, Code Editor in text mode, and Script Editing Document. The first two options are used within a regular MatDeck document and the last, Script Editor Document is a dedicated code editing document.

For short and simple mathematical and programing calculations in a MatDeck document you should utilize the Canvas for Math objects (see Insert and Math tab).

```
edit_code( )
{
  print("Hello world")
}
```

Make programm

edit_code( ) Call function

For more complex programing within a MatDeck document, MatDeck provides the Code Editor in text mode. The Code Editor is enabled by clicking Text/Code icon in the Math Tab, or using **ctrl + i**. The code lines are numbered as seen here:

```
1  // Edit code here
2  varaiable := 5
```

In order to get back to text editing you must first click the Text/Code icon in the Math Tab, or use **ctrl + i**.

For more complex programing MatDeck provides the alternate option, Script Editing Document. The document is generated by using the File-New and then using the drop down tool bar next to it select the Script option. The Script Editing Document is dedicated for programming and it can contain only code in text mode as in the example above.

MatDeck documents are evaluated every time after a **=** or a new line key is pressed and you will get results immediately after. Also you can click on the document, use **ctrl + e** or the Evaluate button to explicitly evaluate and execute codes within the document at any point in time.

MatDeck documents and scripts can be compiled into executable applications. Behavior of such applications will be the same as in the source script but it will run much faster. For compiling processes you should use the Build And Run Exe button from the tool bar. You will be asked to download and set our software development kit (see Math Settings) if it is not present.

3. Syntax

MatDeck Script is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

Also keep in mind that script is executed from the left to the right and from the top to the bottom.

4. Data types

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the types of values that can be represented and manipulated in a programming language.

MatDeck Script allows you to work with these data types:

- boolean
- integer
- double
- complex
- symbolic value
- string
- symbolic function
- unit
- vector
- matrix
- expression
- equation
- interval
- object

MatDeck Script also defines trivial data type **undefined** (void).

Composite data types like vector, matrix, equation and interval are composed of primitive data types so you should enter its keyword before you can enter its data.

$$\text{type}(\text{true}) = \text{"boolean"}$$
$$\text{type}(3) = \text{"integer"}$$
$$\text{type}(3.5) = \text{"double"}$$
$$\text{type}(4 + 7i) = \text{"complex"}$$

$$\text{type}(a) = \text{"symbolic value"}$$
$$\text{type}(\text{"a"}) = \text{"string"}$$
$$\text{type}(\sin(x)) = \text{"symbolic function"}$$
$$\text{type}(a + b) = \text{"expression"}$$

$$\text{type}\left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right) = \text{"matrix"}$$
$$\text{type}\left(\begin{bmatrix} 3 \\ 8 \end{bmatrix}\right) = \text{"vector"}$$

$$\text{type}(\text{void}) = \text{"undefined"}$$
$$\text{type}() = \text{"undefined"}$$

$$\text{type}(m) = \text{"symbolic value"}$$
$$\text{type}(x == 3) = \text{"equation"}$$

$$\text{type}((\ 3\ ,\ 5\ ]) = \text{"interval"}$$

5. Variables

Like many other programming languages, MatDeck Script has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the

container.

Variable declaration and initialization

Before you use a variable you must declare it. Unlike many other languages, you don't have to tell MatDeck Script during variable declaration what type of value the variable will hold. Variable can hold a value of any data type. Variable will get its initial value type during variable declaration and it can be changed during the execution of a program.

Variables are declared with the **:=** operator as follows.

```
a := 7
name := "Your name"
type(a) = "integer"
type(name) = "string"
```

Storing a value in a variable is called **variable initialization**. You should do variable initialization at the time of variable creation. Also you can change variable value later with variable assignment operator **v=** (or with **=** in functions).

```
name = "My name"
name = "My name"      <-- when entered variable assignment operator v= looks like = but bold
name = "My name"
```

You can re-declare same variable again but it is good practice to use variable assignment operator like in the example above.

Variable scope

The scope of a variable is the region of your code or document in which it is defined. MatDeck Script variables have two scopes:

- Global scope − A global variable has global scope which means it can be defined anywhere in your code or document.
- Local scope − A local variable will be visible only within a function where it is defined. Function arguments are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function argument with the same name as a global variable, you effectively hide the global variable. Take a look into the following example.

```
var := "global variable"
fn( )
{
  1  var := "local variable"
  2  return(var)
}
```

fn( ) = "local variable"

Variable names

While naming your variables in MatDeck Script, keep the following rules in mind:

- You should not use any of the MatDeck Script reserved keywords as a variable name.
- Variable names should not start with a numeral (0-9). They must begin with a letter.
- Variable names are case-sensitive. For example, **Name** and **name** are two different variables.

6. Operators

An operator is a symbol that tells the MatDeck to perform specific mathematical or logical manipulations. MatDeck Script operators can also perform operations between different data types if it is possible.

MatDeck Script supports the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operator
- Compound Assignment Operators
- Subscript Operator

Arithmetic Operators

Following arithmetic operators are supported by MatDeck Script:

- **+** Adds two operands
- **-** Subtract second operant from the first
- * Multiplies both operands
- **/** Divides numerator by de-numerator

$2 \cdot 5 = 10$
$\text{"I"} + \text{"am"} = \text{"I am"}$
$\text{"Cat"} - \text{"t"} = \text{undefined}$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 2 = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Relational Operators

Following relational operators are supported by MatDeck Script:

- **==** Checks if the values of two operands are equal or not, if yes then condition becomes true.
- **!=** Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
- **>** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
- **<** Checks if the value of left operand is less than the value of right operand, if yes then condition

becomes true.

- **>=** Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
- **<=** Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

x > y = false

x < y = false

7 > 5 = true

true == 1 = true

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} == \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \text{true} \qquad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} > \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} = \text{false}$$

Logical Operators

Following logical operators are supported by MatDeck Script:

- **&&** Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
- **||** Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
- **!** Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.

Assignment Operator

v= or in functions just **=** will assign value to the existing variable.

Compound Assignment Operators

- **+=** Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
- **-=** Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
- ***=** Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
- **/=** Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.

Subscript Operator

[ ] gives access to the vector or matrix element

$$a := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

a[0] = 1

a[0] = 5

a[0] = 5

$$a = \begin{bmatrix} 5 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 5 \\ a \end{bmatrix}[1] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 5 \\ a \end{bmatrix}[1] = \begin{bmatrix} 5 & 2 \\ 3 & 4 \end{bmatrix}$$

Operators Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

Operators with the highest precedence appear at the top of the list, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

- Subscript Operator **[ ]**
- Arithmetic Operators *** / + -**
- Relational Operators **< <= > >= == !=**
- Logical Operators **&& ||**
- Compound Assignment Operators **+= -= *= /=**
- Assignment Operator **v=**

7. Control statements if, else if, else

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.

MatDeck Script supports conditional statements which are used to perform different actions based on different conditions. These statements are:

- **if** statement is the fundamental control statement that allows MatDeck Script to make decisions and execute statements conditionally.
- **else if** (can be used only after **if**)
- **else** (can be used only after **if** or **else if**)

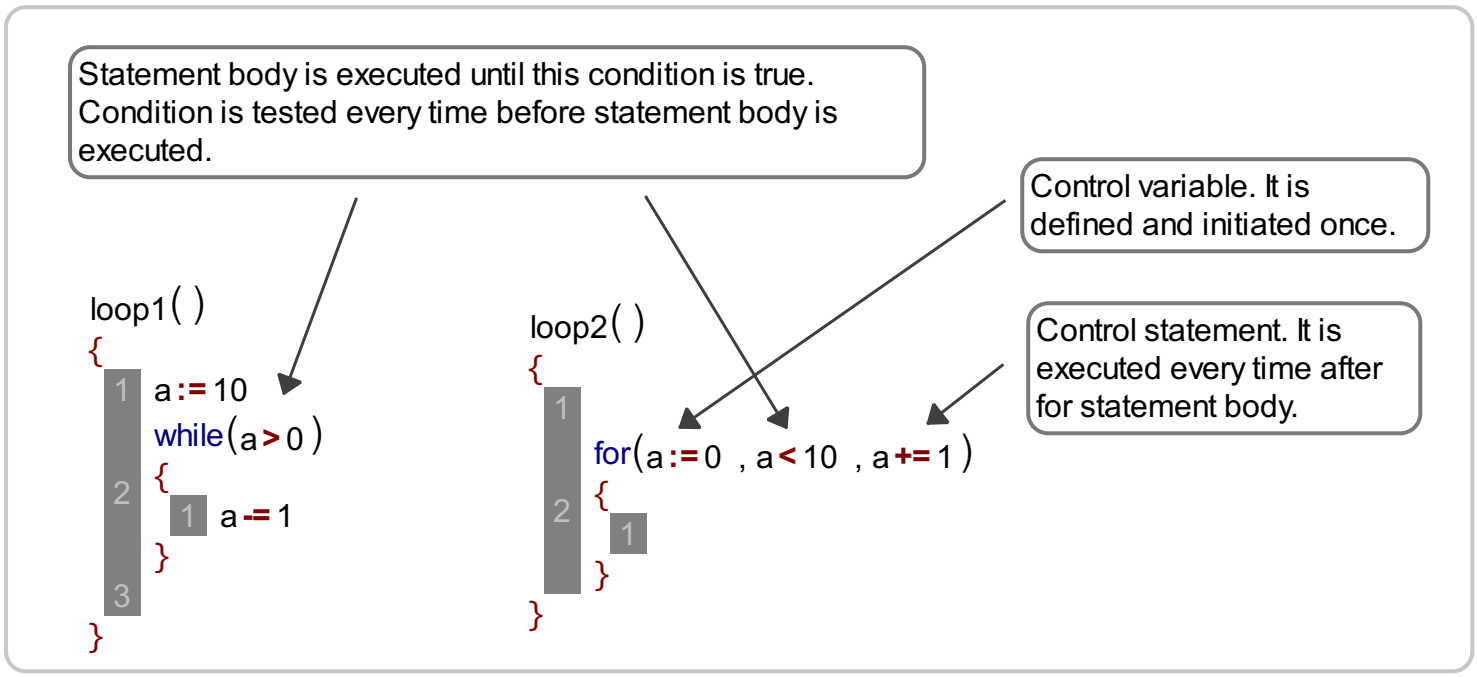
```
condition(arg)
{
    if(arg == 0 )                          condition(1) = "arg is 1"
    {
1       return("arg is 0")                 this statement is executed
    }
2
    else if(arg == 1 )
    {
3       return("arg is 1")
    }
4
    else
    {
5       return("arg is not 0 and not 1")
    }
}
```

8. Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially, the first statement is executed first, followed by the second, and so on.

MatDeck Script provides the following type of loops to handle looping requirements:

- **while** loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
- **for** loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.



9. Loop control

There may be a situation when you need to come out of a loop without reaching its bottom. There may also be a situation when you want to skip a part of your code block and start the next iteration of the loop.

To handle all such situations, MatDeck Script provides **break** and **continue** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

- **break** statement is used to exit a loop early, breaking out of the enclosing curly braces.
- **continue** statement starts the next iteration of the loop and skip the remaining code block. When a **continue** statement is encountered, the program flow moves to the loop check expression immediately and if the condition remains true, then it starts the next iteration, otherwise the control comes out of the loop.

```
3   // Sum of numbers from 10 to 20
4   loopcontol()
5   {
6     a := 0
7     for(n := 0; n < 30; n += 1)
8     {
9       if(n < 10)
10      {
```

```
11
12          continue
13        }
14        a += n
15        if(n >= 20)
16        {
17          break
18        }
19      }
20      return(a)
21    }
```

10. Functions

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Also control and loop statements are available only in functions.

Defining a function

Before we use a function, we need to define it. The most common way to define a function in MatDeck Script is by using the **function** keyword, and enter a unique function name in the red square.

Second way is to enter function name in the new line and enter **(** .

After this move the cursor inside the brackets with right arrow key and add function arguments. After each argument you can use "**,**" key to add place for the next argument, or backspace to remove previous argument. You can skip this step for functions without arguments.

Next step is to move a cursor to the right of the bracket and create function body with "**{**" key.

Calling a function

To invoke a function somewhere later in the script, you would simply need to write the name of that function as shown in the following code.

Return statement

A MatDeck Script function can have an optional **return** statement. This is required if you want to return a value from a function. Functions without return statement will return **void** (undefined).

11. Classes

Classes and are often called user-defined types. A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

Class definition

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces.

Define a object

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. In that process class constructor (main function with the same name as class) is executed.

The public data members of objects of a class can be accessed using the direct member access operator (.)

```
class rectangle
{
  wid := 0
  hei := 0

  rectangle(w , h)
  {
    wid = w
    hei = h
  }

  surface( )
  {
    return(wid hei)
  }
}
```

```
r1 := rectangle(3 , 4)
r2 := rectangle(2 , 2)
r1.surface( ) = 12
r2.surface( ) = 4
r1.wid = 3
r1.hei = 4
r2.wid = 2
r2.hei = 2
```

Special programing statements and functions

Include statement

When document or script is finished and saved you can reuse it later in the new document or script. MatDeck will include all functions, classes and variables from it for use in the new document.

You can dedicate one directory for that purpose. Default include directory is **include** directory under the MatDeck installation directory.

```
include("somefile.mdd")
```

Plug-in name statement

Plug-in naming. For more see plug-in user manual.

```
plugin name("my pugin")
```

Console input and output

All MatDeck scripts can be compiled into executable files. You can use the following functions to print to the console and get user input from the console.

```
print("some text")
a:=2
print(to string(a))

c:=getc(1)   <- expected input type is integer
s:=getc(a)   <- expected input type is symbolic value
```

In the document or script **getc()** will return its type argument as return value and **print()** will do nothing.

Programing examples

- Minimum and maximum of vector (.mdd), (.pdf)
- Recursion (.mdd), (.pdf)
- Newton - Raphson method (.mdd), (.pdf)
- Secant method (.mdd), (.pdf)
- Class (.mdd), (.pdf)