

In the following example, we compute the sum $s = 1 + 3 + 5 + 7 + 9 = 25$. The `modulo(i,2)` function returns 0 if the number i is even. In this situation, the script goes on to the next loop.

```
s = 0
i = 0
while ( i < 10 )
    i = i + 1
    if ( modulo ( i , 2 ) == 0 ) then
        continue
    end
    s = s + i
end
```

If the previous script is executed, the final values of the variables `i` and `s` are:

```
-->s
s =
    25.
-->i
i =
    10.
```

As an example of vectorized computation, the previous algorithm can be performed in one function call only. Indeed, the following script uses the `sum` function, combined with the colon operator `:` and produces same the result.

```
s = sum(1:2:10);
```

The previous script has two main advantages over the `while`-based algorithm.

1. The computation makes use of a higher-level language, which is easier to understand for human beings.
2. With large matrices, the `sum`-based computation will be much faster than the `while`-based algorithm.

This is why a careful analysis must be done before developing an algorithm based on a `while` loop.

6 Functions

In this section, we present Scilab functions. We analyze the way to define a new function and the method to load it into Scilab. We present how to create and load a *library*, which is a collection of functions. We also present how to manage input and output arguments. Finally, we present how to debug a function using the `pause` statement.

6.1 Overview

Gathering various steps into a reusable function is one of the most common tasks of a Scilab developer. The most simple calling sequence of a function is the following:

```
outvar = myfunction ( invar )
```

where the following list presents the various variables used in the syntax:

- `myfunction` is the name of the function,
- `invar` is the name of the input arguments,
- `outvar` is the name of the output arguments.

The values of the input arguments are not modified by the function, while the values of the output arguments are actually modified by the function.

We have in fact already met several functions in this document. The `sin` function, in the `y=sin(x)` statement, takes the input argument `x` and returns the result in the output argument `y`. In Scilab vocabulary, the input arguments are called the *right hand side* and the output arguments are called the *left hand side*.

Functions can have an arbitrary number of input and output arguments so that the complete syntax for a function which has a fixed number of arguments is the following:

```
[o1, ..., on] = myfunction ( i1, ..., in )
```

The input and output arguments are separated by commas `,`. Notice that the input arguments are surrounded by opening and closing parentheses, while the output arguments are surrounded by opening and closing *square* brackets.

In the following Scilab session, we show how to compute the *LU* decomposition of the Hilbert matrix. The following session shows how to create a matrix with the `testmatrix` function, which takes two input arguments, and returns one matrix. Then, we use the `lu` function, which takes one input argument and returns two or three arguments depending on the provided output variables. If the third argument `P` is provided, the permutation matrix is returned.

```
-->A = testmatrix("hilb",2)
A =
    4.   - 6.
   - 6.   12.
-->[L,U] = lu(A)
U =
   - 6.   12.
    0.    2.
L =
   - 0.6666667    1.
    1.           0.
-->[L,U,P] = lu(A)
P =
    0.    1.
    1.    0.
U =
   - 6.   12.
    0.    2.
L =
    1.           0.
   - 0.6666667    1.
```

Notice that the behavior of the `lu` function actually changes when three output arguments are provided: the two rows of the matrix `L` have been swapped. More

function	opens a function definition
endfunction	closes a function definition
argn	number of input/output arguments in a function call
varargin	variable numbers of arguments in an input argument list
varargout	variable numbers of arguments in an output argument list
fun2string	generates ASCII definition of a scilab function
get_function_path	get source file path of a library function
getd	getting all functions defined in a directory
head_comments	display Scilab function header comments
listfunctions	properties of all functions in the workspace
macrovar	variables of function

Figure 29: Scilab functions to manage functions.

specifically, when two output arguments are provided, the decomposition $A = LU$ is provided (the statement $A-L*U$ allows to check this). When three output arguments are provided, permutations are performed so that the decomposition $PA = LU$ is provided (the statement $P*A-L*U$ can be used to check this). In fact, when two output arguments are provided, the permutations are applied on the L matrix. This means that the `lu` function knows how many input and output arguments are provided to it, and changes its algorithm accordingly. We will not present in this document how to provide this feature, i.e. a variable number of input or output arguments. But we must keep in mind that this is possible in the Scilab language.

The commands provided by Scilab to manage functions are presented in figure 29. In the next sections, we will present some of the most commonly used commands.

6.2 Defining a function

To define a new function, we use the **function** and **endfunction** Scilab keywords. In the following example, we define the function **myfunction**, which takes the input argument **x**, multiplies it by 2, and returns the value in the output argument **y**.

```
function y = myfunction ( x )
    y = 2 * x
endfunction
```

The statement **function y = myfunction (x)** is the *header* of the function while the *body* of the function is made of the statement **y = 2 * x**. The body of a function may contain one, two or more statements.

There are at least three possibilities to define the previous function in Scilab.

- The first solution is to type the script directly into the console in an interactive mode. Notice that, once the "function y = myfunction (x)" statement has been written and the enter key is typed in, Scilab creates a new line in the console, waiting for the body of the function. When the "endfunction" statement is typed in the console, Scilab returns back to its normal edition mode.

- Another solution is available when the source code of the function is provided in a file. This is the most common case, since functions are generally quite long and complicated. We can simply copy and paste the function definition into the console. When the function definition is short (typically, a dozen lines of source code), this way is very convenient. With the editor, this is very easy, thanks to the *Load into Scilab* feature.
- We can also use the `exec` function. Let us consider a Windows system where the previous function is written in the file "C:\myscripts\examples-functions.sce". The following session gives an example of the use of `exec` to load the previous function.

```
-->exec("C:\myscripts\examples-functions.sce")
-->function y = myfunction ( x )
-->  y = 2 * x
-->endfunction
```

The `exec` function executes the content of the file as if it were written interactively in the console and displays the various Scilab statements, line after line. The file may contain a lot of source code so that the output may be very long and useless. In these situations, we add the semicolon character ";" at the end of the line. This is what is performed by the *Execute file into Scilab* feature of the editor.

```
-->exec("C:\myscripts\examples-functions.sce" );
```

Once a function is defined, it can be used as if it was any other Scilab function.

```
-->exec("C:\myscripts\examples-functions.sce");
-->y = myfunction ( 3 )
y
=
6.
```

Notice that the previous function sets the value of the output argument `y`, with the statement `y=2*x`. This is mandatory. In order to see it, we define in the following script a function which sets the variable `z`, but not the output argument `y`.

```
function y = myfunction ( x )
  z = 2 * x
endfunction
```

In the following session, we try to use our function with the input argument `x=1`.

```
-->myfunction ( 1 )
!--error 4
Undefined variable: y
at line      4 of function myfunction called by :
myfunction ( 1 )
```

Indeed, the interpreter tells us that the output variable `y` has not been defined.

When we make a computation, we often need more than one function in order to perform all the steps of the algorithm. For example, consider the situation where we need to optimize a system. In this case, we might use an algorithm provided by Scilab, say `optim` for example. First, we define the cost function which is to be optimized, according to the format expected by `optim`. Second, we define a driver,

which calls the `optim` function with the required arguments. At least two functions are used in this simple scheme. In practice, a complete computation often requires a dozen of functions, or more. In this case, we may collect our functions in a library and this is the topic of the next section.

6.3 Function libraries

A function library is a collection of functions defined in the Scilab language and stored in a set of files.

When a set of functions is simple and does not contain any help or any source code in a compiled language like C/C++ or Fortran, a library is a very efficient way to proceed. Instead, when we design a Scilab component with unit tests, help pages and demonstration scripts, we develop a *module*. Developing a module is both easy and efficient, but requires a more advanced knowledge of Scilab. Moreover, modules are based on function libraries, so that understanding the former allows to master the latter. Modules will not be described in this document. Still, in many practical situations, function libraries allow to efficiently manage simple collections of functions and this is why we describe this system here.

In this section, we describe a very simple library and show how to load it automatically at Scilab startup.

Let us make a short outline of the process of creating and using a library. We assume that we are given a set of `.sci` files containing functions.

1. We create a binary version of the scripts containing the functions. The `genlib` function generates binary versions of the scripts, as well as additional indexing files.
2. We load the library into Scilab. The `lib` function allows to load a library stored in a particular directory.

Before analyzing an example, let us consider some general rules which must be followed when we design a function library. These rules will then be reviewed in the next example.

The file names containing function definitions should end with the `.sci` extension. This is not mandatory, but helps in identifying the Scilab scripts on a hard drive.

Several functions may be stored in each `.sci` file, but only the first one will be available from outside the file. Indeed, the first function of the file is considered to be the only public function, while the other functions are (implicitly) private functions.

The name of the `.sci` file must be the same as the name of the first function in the file. For example, if the function is to be named `myfun`, then the file containing this function must be `myfun.sci`. This is mandatory in order to make the `genlib` function work properly.

The functions which allow to manage libraries are presented in figure [30](#).

We shall now give a small example of a particular library and give some details about how to actually proceed.

Assume that we use a Windows system and that the `samplelib` directory contains two files:

genlib	build library from functions in a given directory
lib	library definition

Figure 30: Scilab commands to manage functions.

- `C:/samplelib/function1.sci`:

```
function y = function1 ( x )
    y = 1 * function1_support ( x )
endfunction
function y = function1_support ( x )
    y = 3 * x
endfunction
```

- `C:/samplelib/function2.sci`:

```
function y = function2 ( x )
    y = 2 * x
endfunction
```

In the following session, we generate the binary files with the **genlib** function, which takes as its first argument a string associated with the library name, and takes as its second argument the name of the directory containing the files. Notice that only the functions **function1** and **function2** are publicly available: the **function1_support** function can be used inside the library, but cannot be used outside.

```
-->genlib("mylibrary","C:/samplelib")
-->mylibrary
mylibrary =
Functions files location : C:\samplelib\
function1          function2
```

The **genlib** function generates the following files in the directory *"C:/samplelib"*:

- **function1.bin**: the binary version of the **function1.sci** script,
- **function2.bin**: the binary version of the **function2.sci** script,
- **lib**: a binary version of the library,
- **names**: a text file containing the list of functions in the library.

The binary files ***.bin** and the **lib** file are cross-platform in the sense that they work equally well under Windows, Linux or Mac.

Once the **genlib** function has been executed, the two functions are immediately available, as detailed in the following example.

```
-->function1(3)
ans =
    9.
-->function2(3)
ans =
    6.
```

In practical situations, though, we would not generate the library everytime it is needed. Once the library is ready, we would like to load the library directly. This is done with the `lib` function, which takes as its first argument the name of the directory containing the library and returns the library, as in the following session.

```
-->mylibrary = lib("C:\samplelib")
ans =
Functions files location : C:\samplelib\
function1          function2
```

If there are many libraries, it might be inconvenient to load manually all libraries at startup. In practice, the `lib` statement can be written once for all, in Scilab startup file, so that the library is immediately available at startup. The startup directory associated with a particular Scilab installation is stored in the variable `SCIHOME`, as presented in the following session, for example on Windows.

```
-->SCIHOME
SCIHOME =
C:\Users\username\AppData\Roaming\Scilab\scilab-5.2.0
```

In the directory associated with the `SCIHOME` variable, the startup file is `.scilab`. The startup file is automatically read by Scilab at startup. It must be a regular Scilab script (it can contain valid comments). To make our library available at startup, we simply write the following lines in our `.scilab` file.

```
// Load my favorite library.
mylibrary = lib("C:/samplelib/")
```

With this startup file, the functions defined in the library are available directly at Scilab startup.

6.4 Managing output arguments

In this section, we present the various ways to manage output arguments. A function may have zero or more input and/or output arguments. In the most simple case, the number of input and output arguments is pre-defined and using such a function is easy. But, as we are going to see, even such a simple function can be called in various ways.

Assume that the function `simplef` is defined with 2 input arguments and 2 output arguments, as following.

```
function [y1 , y2] = simplef ( x1, x2 )
    y1 = 2 * x1
    y2 = 3 * x2
endfunction
```

In fact, the number of output arguments of such a function can be 0, 1 or 2. When there is no output argument, the value of the first output argument is stored in the `ans` variable. We may also set the variable `y1` only. Finally, we may use all the output arguments, as expected. The following session presents all these calling sequences.

```
-->simplef ( 1 , 2 )
ans =
2.
```

whereami	display current instruction calling tree
where	get current instruction calling tree

Figure 31: Scilab commands associated with the call stack.

```
-->y1 = simplef ( 1 , 2 )
y1 =
    2.
-->[y1,y2] = simplef ( 1 , 2 )
y2 =
    6.
y1 =
    2.
```

We have seen that the most basic way of defining functions already allows to manage a variable number of output arguments. There is an even more flexible way of managing a variable number of input and output arguments, based on the **argn**, **varargin** and **varargout** variables. This more advanced topic will not be detailed in this document.

6.5 Levels in the call stack

Obviously, function calls can be nested, i.e. a function **f** can call a function **g**, which in turn calls a function **h** and so forth. When Scilab starts, the variables which are defined are at the *global* scope. When we are in a function which is called from the global scope, we are one level down in the call stack. When nested function calls occur, the current level in the call stack is equal to the number of previously nested calls. The functions presented in figure 31 allows to inquire about the state of the call stack.

In the following session, we define 3 functions which are calling one another and we use the function **whereami** to display the current instruction calling tree.

```
function y = fmain ( x )
    y = 2 * flevel1 ( x )
endfunction
function y = flevel1 ( x )
    y = 2 * flevel2 ( x )
endfunction
function y = flevel2 ( x )
    y = 2 * x
    whereami()
endfunction
```

When we call the function **fmain**, the following output is produced. As we can see, the 3 levels in the call stack are displayed, associated with the corresponding function.

```
-->fmain(1)
whereami called at line 3 of macro flevel2
flevel2  called at line 2 of macro flevel1
flevel1  called at line 2 of macro fmain
ans =
```

8.

In the previous example, the various calling levels are the following:

- level 0 : the global level,
- level -1 : the body of the `fmain` function,
- level -2 : the body of the `flevel1` function,
- level -3 : the body of the `flevel2` function.

These calling levels are displayed in the prompt of the console when we interactively debug a function with the `pause` statement or with breakpoints.

6.6 The return statement

Inside the body of a function, the `return` statement allows to immediately return, i.e. it immediately quits the current function. This statement can be used in cases where the remaining of the algorithm is not necessary.

The following function computes the sum of integers from `istart` to `iend`. In regular situations, it uses the `sum` function to perform its job. But if the `istart` variable is negative or if the `istart<=iend` condition is not satisfied, the output variable `y` is set to 0 and the function immediately returns.

```
function y = mysum ( istart , iend )
    if ( istart < 0 ) then
        y = 0
        return
    end
    if ( iend < istart ) then
        y = 0
        return
    end
    y = sum ( istart : iend )
endfunction
```

The following session allows to check that the `return` statement is correctly used by the `mysum` function.

```
-->mysum ( 1 , 5 )
ans =
    15.
-->mysum ( -1 , 5 )
ans =
     0.
-->mysum ( 2 , 1 )
ans =
     0.
```

Some developers state that using several `return` statements in a function is generally a bad practice. Indeed, we must take into account the increased difficulty of debugging such a function, because the algorithm may suddenly quit the body of the function. The user may get confused about what exactly caused the function to return.

pause	wait for interactive user input
resume	resume execution and copy some local variables
abort	interrupt evaluation

Figure 32: Scilab functions to debug manually a function.

This is why, in practice, the **return** statement should be used with care, and certainly not in every function. The rule to follow is that the function should return only at its very last line. Still, in particular situations, using **return** can actually greatly simplify the algorithm, while avoiding **return** would require writing a lot of unnecessary source code.

6.7 Debugging functions with pause

In this section, we present simple debugging methods which allow to fix most simple bugs in a convenient and efficient way. More specifically, we present the **pause**, **resume** and **abort** statements, which are presented in figure 32.

A Scilab session usually consists in the definition of new algorithms by the creation of new functions. It often happens that a syntax error or an error in the algorithm produces a wrong result.

Consider the problem computation of the sum of then integers from **istart** to **iend**. Again, this simple example is chosen for demonstration purposes, since the **sum** function performs it directly.

The following function **mysum** contains a bug: the second argument "foo" passed to the **sum** function has no meaning in this context.

```
function y = mysum ( istart , iend )
    y = sum ( iend : istart , "foo" )
endfunction
```

The following session shows what happens when we use the **mysum** function.

```
-->mysum ( 1 , 10 )
!--error 44
Wrong argument 2.
at line      2 of function mysum called by :
mysum ( 1 , 10 )
```

In order to interactively find the problem, we place a **pause** statement inside the body of the function.

```
function y = mysum ( istart , iend )
    pause
    y = sum ( iend : istart , "foo" )
endfunction
```

We now call the function **mysum** again with the same input arguments.

```
-->mysum ( 1 , 10 )
Type 'resume' or 'abort' to return to standard level prompt.
-1->
```

We are now interactively located *in the body* of the `mysum` function. The prompt `"-1->"` indicates that the current call stack is at level -1. We can check the value of the variables `istart` and `iend` by simply typing their names in the console.

```
-1->istart
istart =
    1.
-1->iend
iend =
   10.
```

In order to progress in our function, we can copy and paste the statements and see what happens interactively, as in the following session.

```
-1->y = sum ( iend : istart , "foo" )
y = sum ( iend : istart , "foo" )
                                           !--error 44
Wrong argument 2.
```

We can see that the call to the `sum` function does not behave how we might expect. The `"foo"` input argument is definitely a bug: we remove it.

```
-1->y = sum ( iend : istart )
y =
    0.
```

After the first revision, the call to the `sum` function is now syntactically correct. But the result is still wrong, since the expected result in this case is 55. We see that the `istart` and `iend` variables have been *swapped*. We correct the function call and check that the fixed version behaves as expected

```
-1->y = sum ( istart : iend )
y =
   55.
```

The result is now correct. In order to get back to the zero level, we now use the `abort` statement, which interrupts the sequence and immediately returns to the global level.

```
-1->abort
-->
```

The `-->` prompt confirms that we are now back at the zero level in the call stack.

We fix the function definition, which becomes:

```
function y = mysum ( istart , iend )
    pause
    y = sum ( istart : iend )
endfunction
```

In order to check our bugfix, we call the function again.

```
-->mysum ( 1 , 10 )
Type 'resume' or 'abort' to return to standard level prompt.
-1->
```

We are now confident about our code, so that we use the `resume` statement, which lets Scilab execute the code as usual.

```
-->mysum ( 1 , 10 )
-1->resume
ans =
    55.
```

The result is correct. All we have to do is to remove the `pause` statement from the function definition.

```
function y = mysum ( istart , iend )
    y = sum ( istart : iend )
endfunction
```

In this section, we have seen that, used in combination, the `pause`, `resume` and `abort` statements are a very effective way to interactively debug a function. In fact, our example is very simple and the method we presented may appear to be too simple to be convenient. This is not the case. In practice, the `pause` statement has proven to be a very fast way to find and fix bugs, even in very complex situations.

7 Plotting

Producing plots and graphics is a very common task for analysing data and creating reports. Scilab offers many ways to create and customize various types of plots and charts. In this section, we present how to create 2D plots and contour plots. Then we customize the title and the legend of our graphics. We finally export the plots so that we can use it in a report.

7.1 Overview

Scilab can produce many types of 2D and 3D plots. It can create x-y plots with the `plot` function, contour plots with the `contour` function, 3D plots with the `surf` function, histograms with the `histplot` function and many other types of plots. The most commonly used plot functions are presented in figure 33.

In order to get an example of a 3D plot, we can simply type the statement `surf()` in the Scilab console.

```
-->surf()
```

During the creation of a plot, we use several functions in order to create the data or to configure the plot. The functions which are presented in figure 34 will be used in the examples of this section.

7.2 2D plot

In this section, we present how to produce a simple x-y plot. We emphasize the use of vectorized functions, which allow to produce matrices of data in one function call.

We begin by defining the function which is to be plotted. The function `myquadratic` squares the input argument `x` with the `^` operator.

```
function f = myquadratic ( x )
    f = x^2
endfunction
```

<code>plot</code>	2D plot
<code>surf</code>	3D plot
<code>contour</code>	contour plot
<code>pie</code>	pie chart
<code>histplot</code>	histogram
<code>bar</code>	bar chart
<code>barh</code>	horizontal bar chart
<code>hist3d</code>	3D histogram
<code>polarplot</code>	plot polar coordinates
<code>Matplot</code>	2D plot of a matrix using colors
<code>Sgrayplot</code>	smooth 2D plot of a surface using colors
<code>grayplot</code>	2D plot of a surface using colors

Figure 33: Scilab plot functions

<code>linspace</code>	linearly spaced vector
<code>feval</code>	evaluates a function on a grid
<code>legend</code>	configure the legend of the current plot
<code>title</code>	configure the title of the current plot
<code>xtitle</code>	configure the title and the legends of the current plot

Figure 34: Scilab functions used when creating a plot.

We can use the `linspace` function in order to produce 50 values in the interval $[1, 10]$.

```
xdata = linspace ( 1 , 10 , 50 );
```

The `xdata` variable now contains a row vector with 50 entries, where the first value is equal to 1 and the last value is equal to 10. We can pass it to the `myquadratic` function and get the function value at the given points.

```
ydata = myquadratic ( xdata );
```

This produces the row vector `ydata`, which contains 50 entries. We finally use the `plot` function so that the data is displayed as a x-y plot.

```
plot ( xdata , ydata )
```

Figure 35 presents the associated x-y plot.

Notice that we could have produced the same plot without generating the intermediate array `ydata`. Indeed, the second input argument of the `plot` function can be a function, as in the following session.

```
plot ( xdata , myquadratic )
```

When the number of points to manage is large, using directly the function allow to save significant amount of memory space, since it avoids to generate the intermediate vector `ydata`.

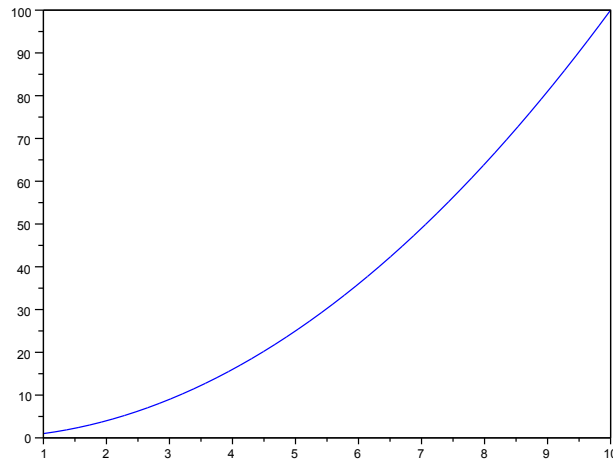


Figure 35: A simple x-y plot.

7.3 Contour plots

In this section, we present the contour plots of a multivariate function and make use of the `contour` function. This type of graphics is often used in the context of numerical optimization as they allow to draw functions of two variables in a way that makes apparent the location of the optimum.

Assume that we are given function f with n variables $f(\mathbf{x}) = f(x_1, \dots, x_n)$ and $\mathbf{x} \in \mathbb{R}^n$. For a given $\alpha \in \mathbb{R}$, the equation

$$f(\mathbf{x}) = \alpha, \quad (2)$$

defines a surface in the $(n + 1)$ -dimensional space \mathbb{R}^{n+1} .

When $n = 2$, the points $z = f(x_1, x_2)$ represent a surface in the three-dimensional space $(x_1, x_2, z) \in \mathbb{R}^3$. This allows to draw *contour* plots of the cost function, as we are going to see. For $n > 3$, though, these plots are not available. One possible solution in this case is to select two significant parameters and to draw a contour plot with these parameters varying (only).

The Scilab function `contour` allows to plot contours of a function f . The `contour` function has the following syntax

```
contour(x,y,z,nz)
```

where

- \mathbf{x} (resp. \mathbf{y}) is a row vector of x (resp. y) values with size `n1` (resp. `n2`),
- \mathbf{z} is a real matrix of size `(n1,n2)`, containing the values of the function or a Scilab function which defines the surface $\mathbf{z}=\mathbf{f}(\mathbf{x},\mathbf{y})$,
- `nz` the level values or the number of levels.

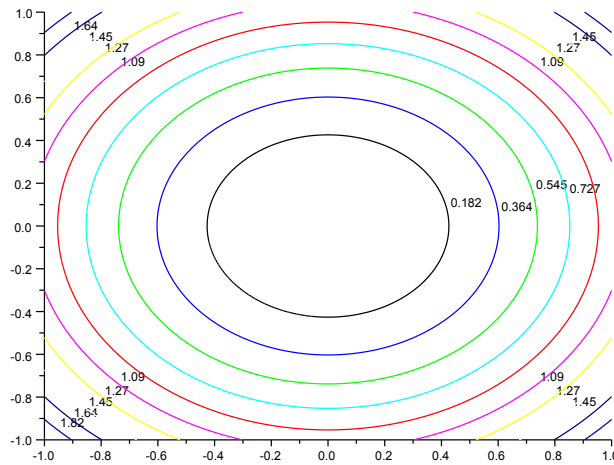


Figure 36: Contour plot of the function $f(x_1, x_2) = x_1^2 + x_2^2$.

In the following Scilab session, we use a simple form of the `contour` function, where the function `myquadratic` is passed as an input argument. The `myquadratic` function takes two input arguments x_1 and x_2 and returns $f(x_1, x_2) = x_1^2 + x_2^2$. The `linspace` function is used to generate vectors of data so that the function is analyzed in the range $[-1, 1]^2$.

```
function f = myquadratic2arg ( x1 , x2 )
    f = x1**2 + x2**2;
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
contour ( xdata , ydata , myquadratic2arg , 10)
```

This produces the contour plot presented in figure 36.

In practice, it may happen that our function has the header `z = myfunction (x)`, where the input variable `x` is a row vector. The problem is that there is only one single input argument, instead of the two arguments required by the `contour` function. There are two possibilities to solve this little problem:

- provide the data to the `contour` function by making two nested loops,
- provide the data to the `contour` function by using `feval`,
- define a new function which calls the first one.

These three solutions are presented in this section. The first goal is to let the reader choose the method which best fits the situation. The second goal is to show that performances issues can be avoided if a consistent use of the functions provided by Scilab is done.

In the following Scilab naive session, we define the quadratic function `myquadratic1arg`, which takes one vector as its single input argument. Then we perform two nested