plotting

March 29, 2024

1 Plotting

2 Table of contents

1. Overview
2. 2D plot
3. Contour plots
4. Titles, axes and legends
5. Export

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.

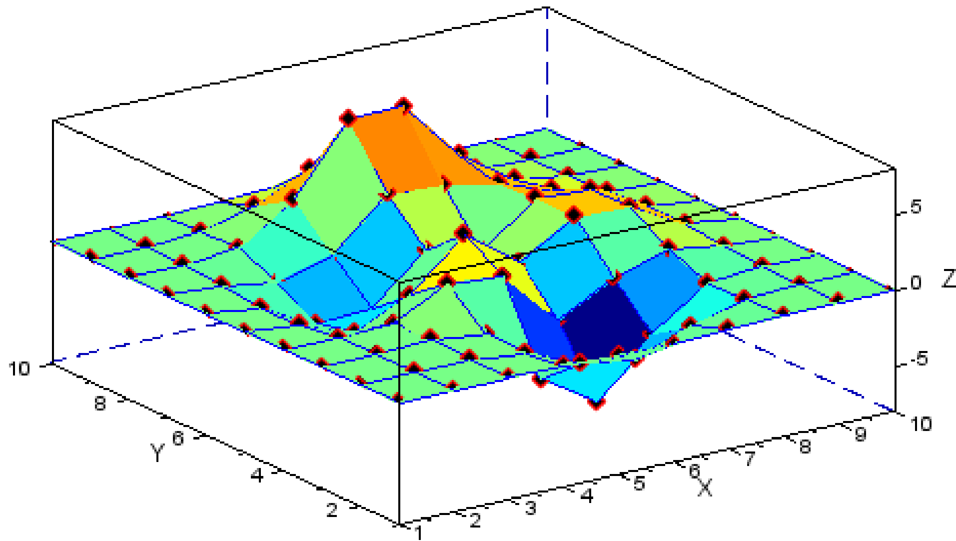
2.1 Overview

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:

- **plot**: 2D plot
- **surf**: 3D plot
- **contour**: contour plot
- **pie**: pie chart
- **histplot**: histogram
- **bar**: bar chart
- **barh**: horizontal bar chart
- **hist3d**: 3D histogram
- **polarplot**: plot polar coordinates
- **Matplot**: 2D plot of a matrix using colors
- **Sgrayplot**: smooth 2D plot of surface using colors
- **grayplot**: 2D plot of a surface using colors

In order to get an examaple of 3D plot, we can simplify type the statement **surf** in th Scilab console

[1]:
```
surf()
```



During the creation of a plot, we use several functions in order to create the data or to configure the plot:

- **linspace**: linearly spaced vector
- **feval**: evaluates a function on a grid
- **legend**: configure the legend of the current plot
- **title**: configure the title of the current plot
- **xtitle**: configure the title and the legends of the current plot

2.2 2D plot

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

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

[2]:
```
function f = myquadratic(x)
    f = x .^ 2
endfunction
delete(gcf())
```

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

[3]:
```
xdata = linspace(1, 10, 50);
xdata(1:10)
```

```
 ans  =
   1.   1.1836735   1.3673469   1.5510204   1.7346939   1.9183673   2.1020408
2.2857143   2.4693878   2.6530612
```

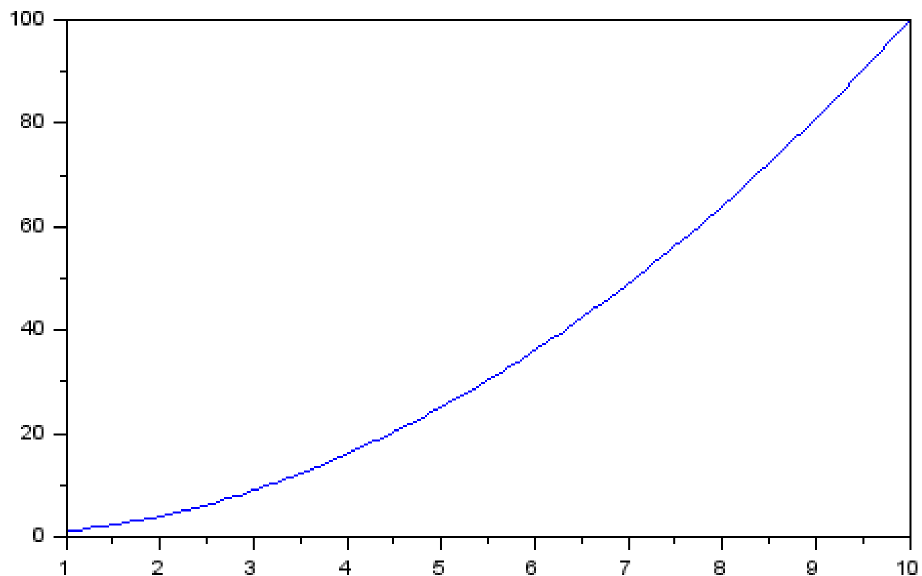
The **xdata** variable now contains a row vector with 50 entries, where the first value is equal to 1 and the last value is equal to 10. We can pass it to the **myquadratic** function and get the function value at the given points.

[4]:
```
ydata = myquadratic(xdata);
ydata(1:10)
```

```
 ans  =
   1.   1.4010829   1.8696377   2.4056643   3.0091628   3.6801333   4.4185756
5.2244898   6.0978759   7.0387339
```

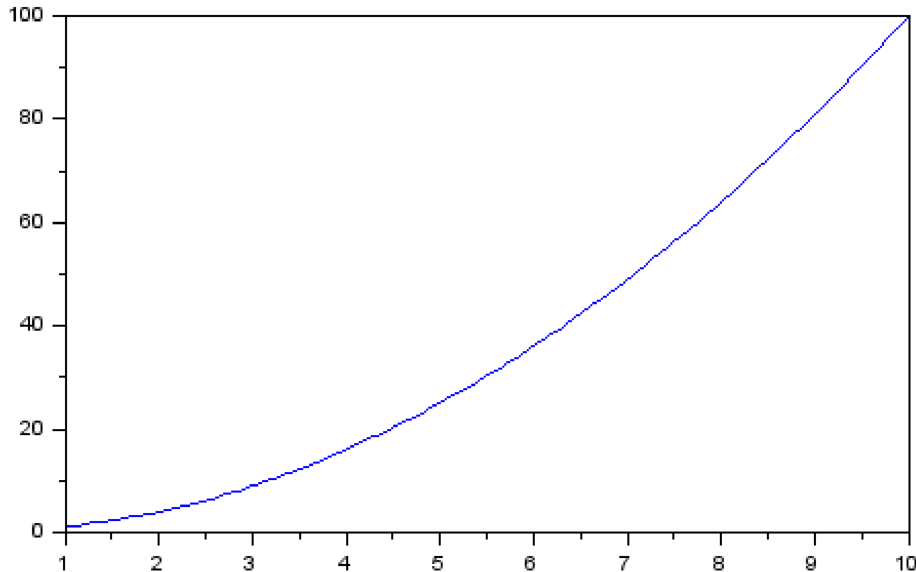
This produces the row vector **ydata**, which contains 50 entries. We finally use the **plot** function so that the data is displayed as an x-y plot.

[5]:
```
plot(xdata, ydata)
```



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.

[6]:
```
plot(xdata, myquadratic)
```



When the number of points to manage is large, using functions directly saves significant amount of memory space, since it avoids generating the intermediate vector **ydata**.

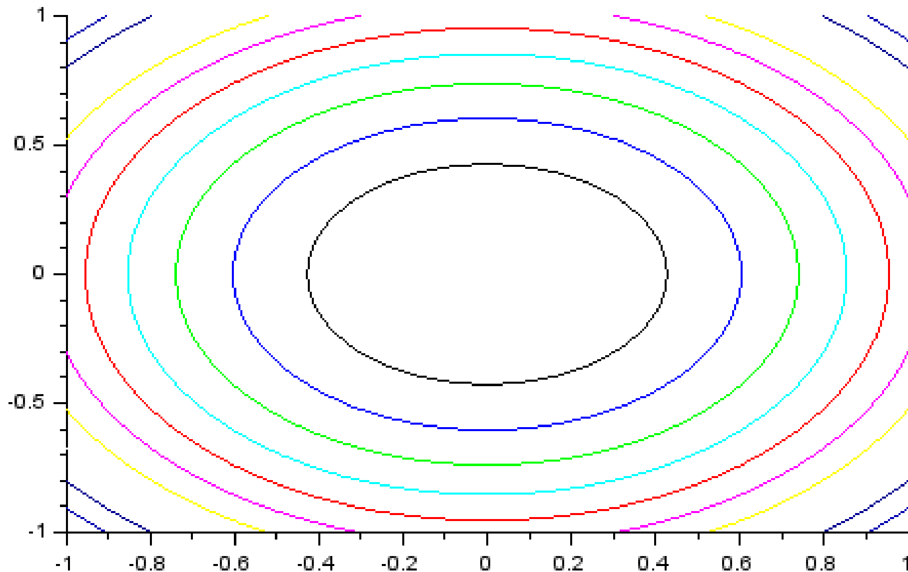
2.3 Contour plots

In this section, we present the contour plots of a multivariate function and make use of the **contour** function. This type of graphic is often used in the context of numerical optimization because it draws functions of two variables in a way that shows the location of the optimum.

The Scilab function **contour** plots contours of a function **f**. The **contour** function has the following syntax `contour(x, y, z, nz)` where - `x` (resp. `y`) is a row vector of **x** (resp. **y**) values with size **n1** (resp **n2**) - `z` is a real matrix of size **(n1,n2)**, containing the values of the function or a Scilab function which defines the surface `z = f(x, y)`, - `nz` the level values or the number of levels.

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 $[-1, 1]^2$.

[7]:
```
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)
```



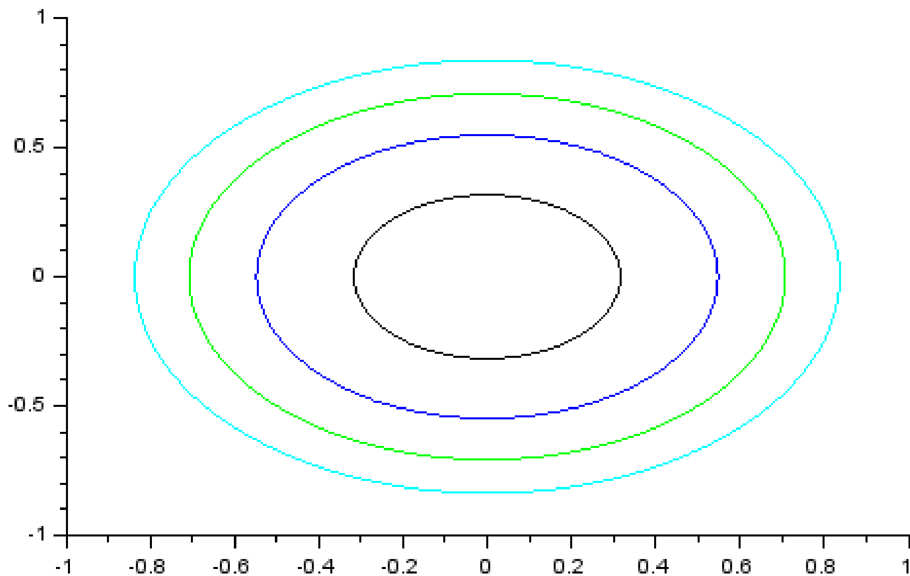
In practice, it may happen that our function has the header **z = myfunction (x)**, where the input variable **x** is a row vector. The problem is that there is only one single input argument, instead of the two arguments required by the **contour** function. There are two possibilities to solve this 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 function **myquadratic1arg**, which takes one vector as its single input argument. Then we perform two nested loops to compute the **zdata** matrix, which contains the **z** values. The **z** values are computed for all the combinations of points $(x(i), y(j)) \in \mathbb{R}^2$, for $i = 1, 2, \ldots, n_x$ and $j = 1, 2, \ldots, n_y$, where n_x and n_y are the number of points in the x and y coordinates. In the end, we call the **contour** function, with the list of required levels (instead of the previous number of levels). This gets exactly the levels we want, instead of letting Scilab compute the levels automatically.

[8]:
```
function f = myquadratic1arg ( x )
  f = x(1)**2 + x(2)**2;
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
// Caution ! Two nested loops, this is bad.
for i = 1:length(xdata)
    for j = 1:length(ydata)
```

```
      x = [xdata(i) ydata(j)].';
      zdata ( i , j ) = myquadratic1arg ( x );
    end
end
contour ( xdata , ydata , zdata , [0.1 0.3 0.5 0.7])
```

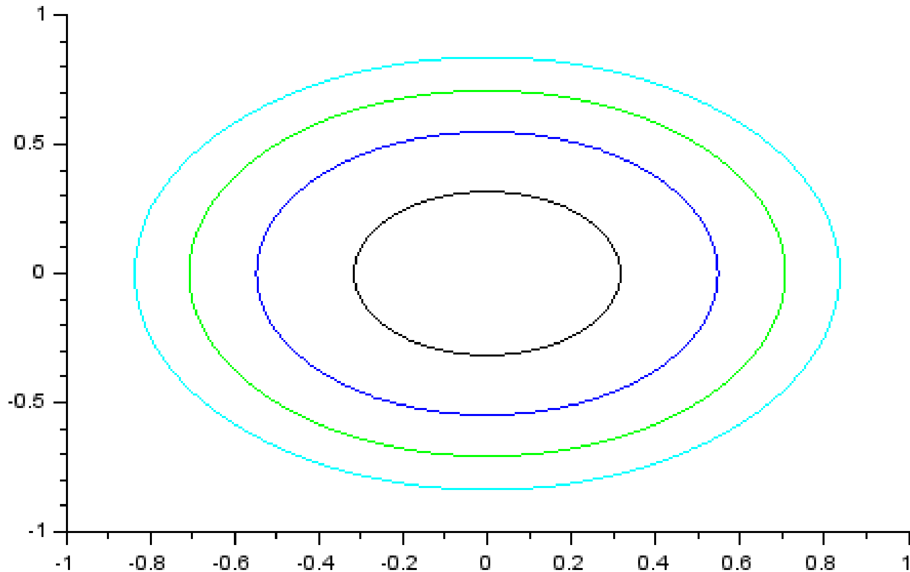


The previous script works perfectly. Still, it is not efficient because it uses two nested loops and this should be avoided in Scilab for performance reasons. Another issue is that we had to store the **zdata** matrix, which might consume a lot of memory space when the number of points is large. This method should be avoided since it is a bad use of the features provided by Scilab.

In the following script, we use the **feval** function, which evaluates a function on a grid of values and returns the computed data. The generated grid is made of all the combinations of points $(x(i), y(j)) \in \mathbb{R}^2$. We assume here that there is no possibility to modify the function **myquadratic1arg**, which takes one input argument. Therefore, we create an intermediate function **myquadratic3**, which takes 2 input arguments. Once done, we pass the **myquadratic3** argument to the **feval** function and generate the **zdata** matrix.

[9]:
```
function f = myquadratic1arg ( x )
  f = x(1)**2 + x(2)**2;
endfunction
function f = myquadratic3 ( x1 , x2 )
  f = myquadratic1arg ( [x1 x2] )
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
```

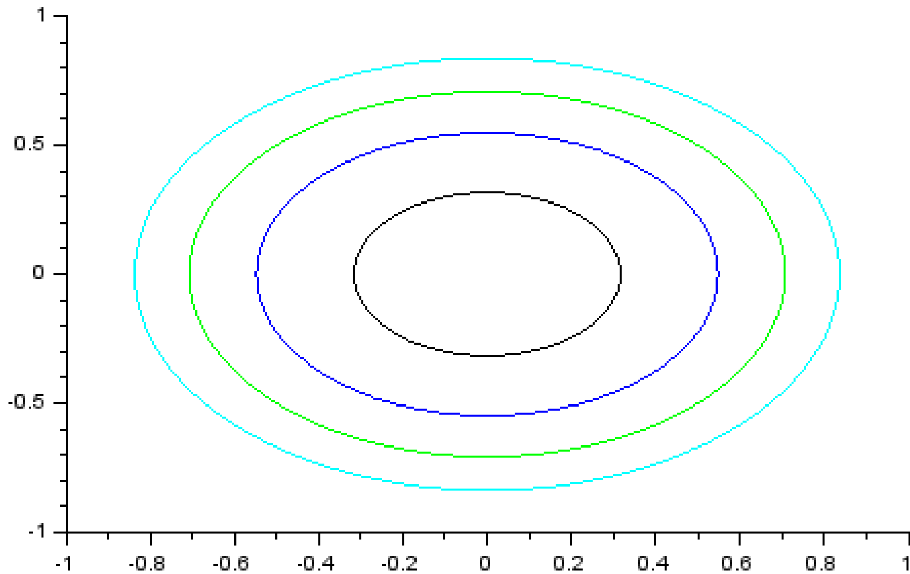
```
zdata = feval ( xdata , ydata , myquadratic3 );
contour ( xdata , ydata , zdata , [0.1 0.3 0.5 0.7])
```



The previous script produces, of course, exactly the same plot as previously. This method should be avoided when possible, since it requires storing the **zdata** matrix, which has size 100×100.

Finally, there is a third way of creating the plot. In the following Scilab session, we use the same intermediate function **myquadratic3** as previously, but we pass it directly to the **contour** function.

[10]:
```
function f = myquadratic1arg ( x )
  f = x(1)**2 + x(2)**2;
endfunction
function f = myquadratic3 ( x1 , x2 )
  f = myquadratic1arg ( [x1 x2] )
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
contour ( xdata , ydata , myquadratic3 , [0.1 0.3 0.5 0.7])
```



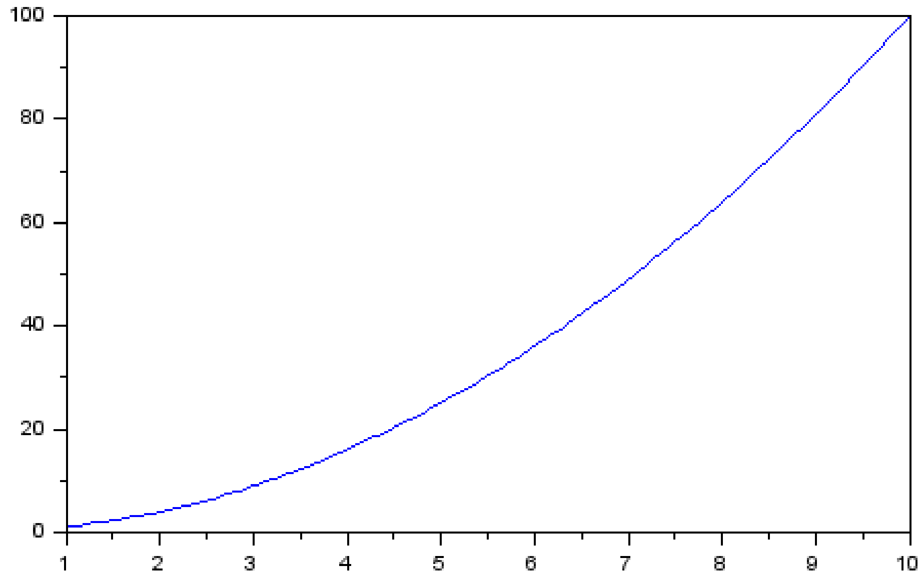
The previous script produces, of course, exactly the same plot as previously. The major advantage is that we did not produce the **zdata** matrix.

We have briefly outlined how to produce simple 2D plots. We are now interested in the configuration of the plot, so that the titles, axis and legends corresponds to our data.

2.4 Titles, axes and legends

In this section, we present the Scilab graphics features which configure the title, axes and legends of an x-y plot.

[11]:
```
function f = myquadratic ( x )
  f = x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata )
```

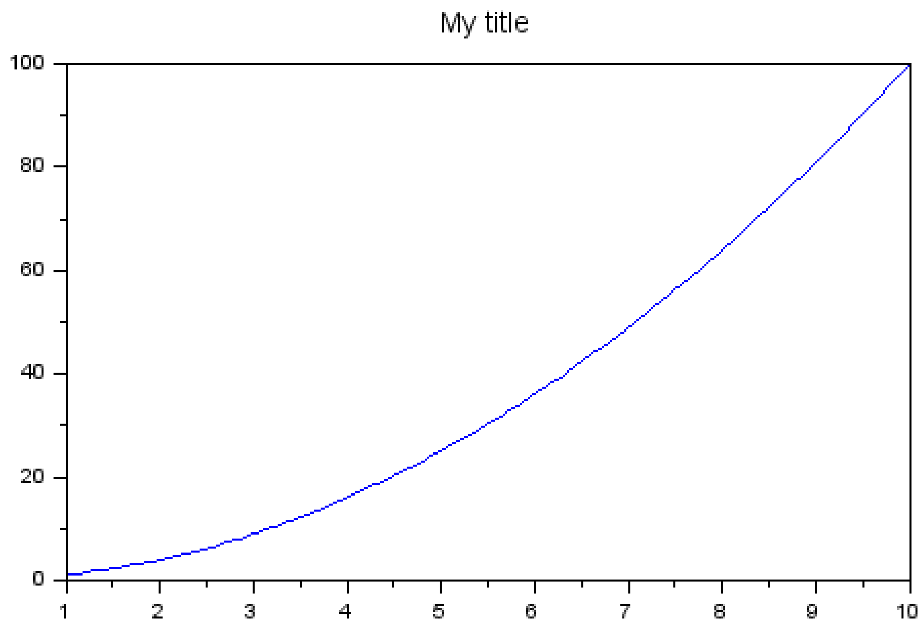


Scilab graphics system is based on **graphics handles**. The graphics handles provide an object-oriented access to the fields of a graphics entity. The graphics layout is decomposed into sub-objects such as the line associated with the curve, the x and y axes, the title, the legends, and so forth. Each object can be in turn decomposed into other objects if required. Each graphics object is associated with a collection of properties, such as the width or color of the line of the curve. These properties can be queried and configured simply by getting or setting their values, like any other Scilab variables. Managing handles is easy and very efficient.

But most basic plot configurations can be done by simple function calls and, in this section, we will focus in these basic features.

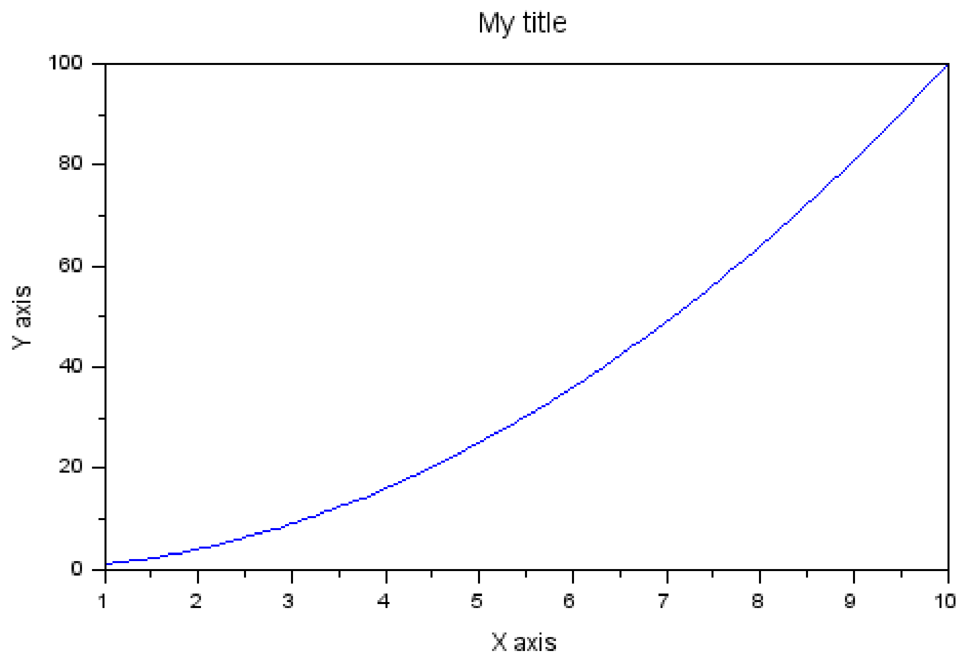
In the following script, we use the **title** function in order to configure the title of our plot.

[12]:
```
function f = myquadratic ( x )
  f = x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata )
title ( "My title" );
```



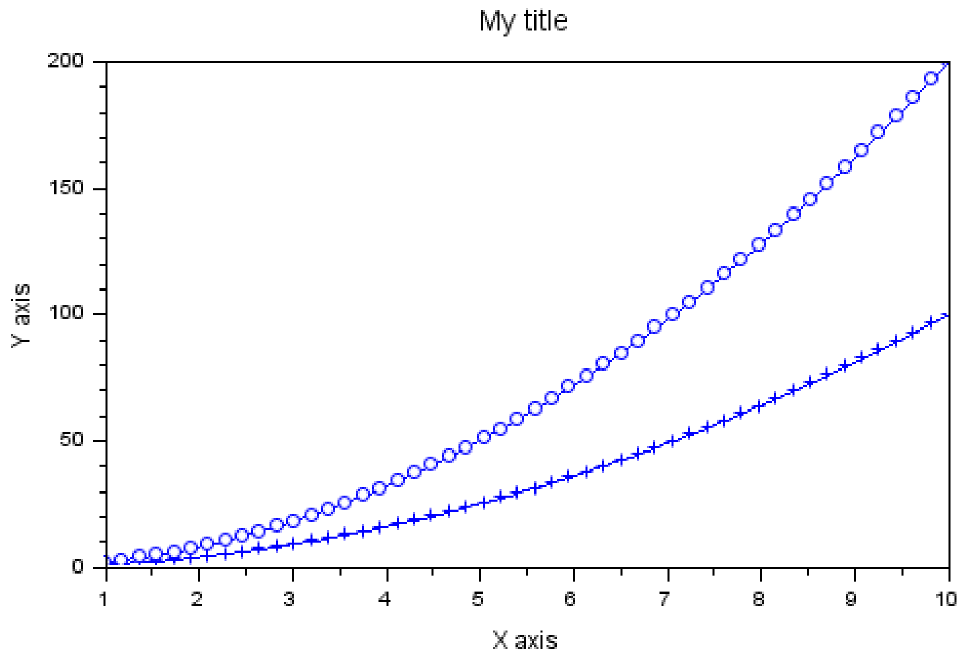
We may want to configure the axes of our plot as well. For this purpose, we use the **xtitle** function in the following script.

[13]:
```
function f = myquadratic ( x )
  f = x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata )
xtitle ( "My title" , "X axis" , "Y axis" );
```



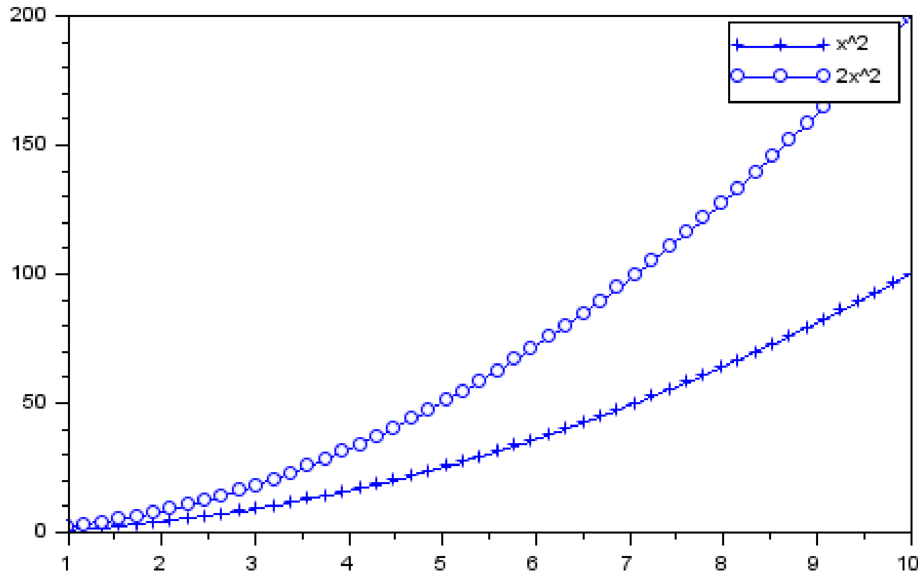
It may happen that we want to compare two sets of data in the same 2D plot, that is, one set of x data and two sets of y data.
In the following script, we define the two functions $f(x) = x^2$ and $f(x) = 2x^2$ and plot the data on the same x-y plot. We additionally use the **"+-"** and **"o-"** options of the **plot** function, so that we can distinguish the two curves $f(x) = x^2$ and $f(x) = 2x^2$.

[14]:
```
function f = myquadratic ( x )
  f = x.^2
endfunction
function f = myquadratic2 ( x )
  f = 2 * x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata , "+-" )
ydata2 = myquadratic2 ( xdata );
plot ( xdata , ydata2 , "o-" )
xtitle ( "My title" , "X axis" , "Y axis" );
```



Moreover, we must configure a legend so that we can know what curve is associated with $f(x) = x^2$ and what curve is associated with $f(x) = 2x^2$. For this purpose, we use the **legend** function in order to print the legend associated with each curve.

[15]:
```
function f = myquadratic ( x )
  f = x.^2
endfunction
function f = myquadratic2 ( x )
  f = 2 * x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata , "+-" )
ydata2 = myquadratic2 ( xdata );
plot ( xdata , ydata2 , "o-" )
legend ( "x^2" , "2x^2" );
```



We now know how to create a graphics plot and how to configure it. If the plot is sufficiently interesting, it may be worth putting it into a report. To do so, we can export the plot into a file, which is the subject of the next section.

2.5 Export

In this section, we present ways of exporting plots into files, either interactively or automatically with Scilab functions.

Scilab can export any graphics into the vectorial and bitmap formats presented below. Once a plot is produced, we can export its content into a file, by interactively using the **File > Export to...** menu of the graphics window. We can then set the name of the file and its type.

Vectorial format functions are:

- **xs2png**: export into PNG
- **xs2pdf**: export into PDF
- **xs2svg**: export into SVG
- **xs2eps**: export into Encapsulated Postscript
- **xs2ps**: export into Postscript
- **xs2emf**: export into EMF (only for Windows)

Bitmap format functions are:

- **xs2fig**: export into FIG
- **xs2gif**: export into GIF
- **xs2jpg**: export into JPG
- **xs2bmp**: export into BMP

- **xs2ppm**: export into PPM

We can alternatively use the **xs2*** functions. All these functions are based on the same calling sequence:

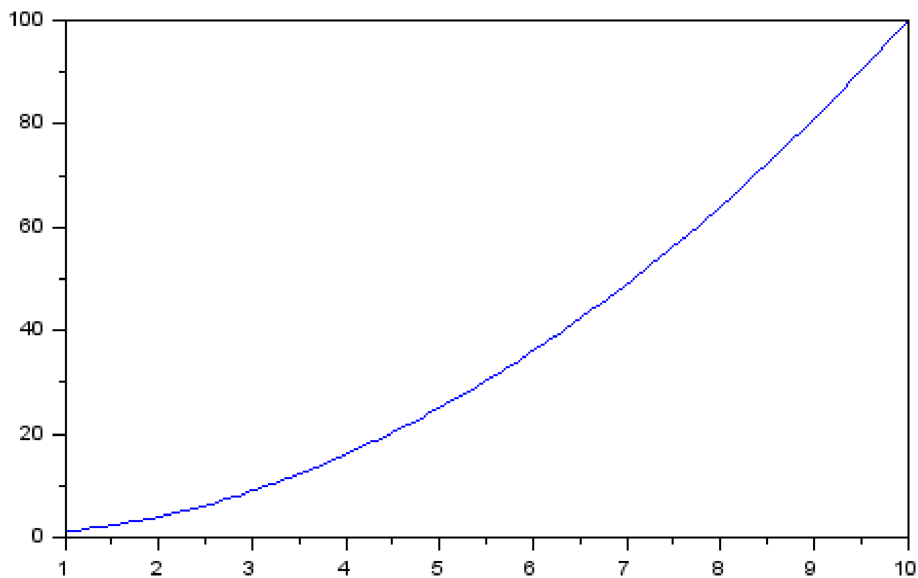
```
xs2png ( window_number , filename )
```

where:

- **window_number** is the number of the graphics window
- and **filename** is the name of the file to export.

For example, the following session exports the plot which is in the graphics window number 0, which is the default graphics window,
into the file **foo.png**.

[16]:
```
function f = myquadratic ( x )
  f = x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata )
xs2png ( 0 , "foo.png" )
```



If we want to produce higher quality documents, the vectorial formats are preferred. For example, **LaTeX** documents may use Scilab plots exported into PDF files to improve their readability, whatever the size of the document.