AMTH142 Lecture 10

Scilab — Graphs
Floating Point Arithmetic

April 20, 2007

Contents

10.1 Graphs in Scilab ........................................ 2
  10.1.1 Simple Graphs ..................................... 2
  10.1.2 Line Styles ....................................... 4
  10.1.3 Multiple Curves ................................... 4
  10.1.4 Multiple Plots .................................... 5
  10.1.5 Titles and Captions ............................... 6
  10.1.6 Other Features .................................... 7
  10.1.7 3D Curves ......................................... 7
  10.1.8 Histograms ........................................ 8

10.2 Floating Point Arithmetic ............................. 9
  10.2.1 Representation of Floating Point Numbers ...... 9
  10.2.2 Precision .......................................... 9
  10.2.3 Exceptional Values ............................... 10
  10.2.4 Rounding ......................................... 11
  10.2.5 Cancellation ...................................... 12
10.1 Graphs in Scilab

10.1.1 Simple Graphs

The simplest graph takes two vectors and plots one against the other:

\[
\begin{align*}
&\text{--> } x = (-20:0.01:20); \\
&\text{--> } y = \sin(x)/x; \\
&\text{--> } \text{plot2d}(x,y)
\end{align*}
\]

You should have noticed the use of the dot operator. To see why it is necessary first note that \( x \) is a vector. So is \( \sin(x) \) whose components are obtained by applying the \( \sin \) function element-by-element to the vector \( x \).

To construct the vector \( y \) whose components are obtained by applying the function \( \sin(x)/x \) element-by-element to the vector \( x \) we perform element-by-element division of the vector \( \sin(x) \) by the vector \( x \).

Scilab graphs joins points by straight lines which sometimes gives the graph a slight polygonal look. If you want a smooth looking graph you need to take a fairly dense of points, 1000 will usually do, for the \( x \)-coordinates.

We can also plot a single vector, whose components are plotted against 1, 2, \ldots, \( n \) where \( n \) is the length of the vector. First we use \texttt{clf} to clear the graphics window and then plot the components of the vector \( y \) used earlier:

\[
\begin{align*}
&\text{--> } \text{clf} \\
&\text{--> } \text{plot2d}(y)
\end{align*}
\]
Notice the different scaling on the $x$-axis.

Here is another example:

```matlab
-->x = linspace(-10, 10, 1000);
-->y = 2*sin(x) + 3*sin(2*x) + 7*sin(3*x) + 5*sin(4*x);
-->clf, plot2d(x,y)
```

In this example `linspace(-10,10,1000)` produces a vector which consists of 1000 evenly spaced points between -10 and 10. Also note that we can use commas or semicolons to separate two or more commands on the one line.
10.1.2 Line Styles

The graphs we have looked at so far were of continuous curves. Data can also be plotted as points by using the *style* option to the `plot2d` command. Negative values for *style* correspond to different types of points, positive values for *style* correspond to different colours. You can use the `xset()` command to bring up a menu to set different styles and colours as well as things like line thickness.

```plaintext
-->x = rand(1,100);
-->y = rand(1,100);
-->clf, plot2d(x,y,style = -1)
```

10.1.3 Multiple Curves

We can plot multiple curves, one on top of the other, by plotting them successively without clearing the screen. Alternatively, if *y* is a matrix, the command `plot2d(x, y)` plots each of the columns of the matrix *y* as a separate curve. In this case *x* has to be a column vector the same length as the columns of *y*. We could also plot the curves in different styles, by setting *style* to a vector of style numbers.

```plaintext
-->x = linspace(-10, 10, 1000)';
-->y1 = 2*sin(x); y2 = 3*sin(2*x);
-->y3 = 7*sin(3*x); y4 = 5*sin(4*x);
```
\[ y = y_1 + y_2 + y_3 + y_4; \]

\[ \text{clf, plot2d}(x,[y_1 \ y_2 \ y_3 \ y_4 \ y]) \]

Note that the use of the transpose \(^T\) to define \(x\) as a column vector. Then each of \(y_1, y_2, y_3, y_4,\) and \(y\) is also a column vector and \([y_1 \ y_2 \ y_3 \ y_4 \ y]\) is a 5 column matrix.

### 10.1.4 Multiple Plots

Multiple graphs can included in one figure using the `subplot` command.

\[ \text{clf} \]

\[ \text{subplot}(2,2,1) \]

\[ \text{plot2d}(x,y_1) \]

\[ \text{subplot}(2,2,2) \]

\[ \text{plot2d}(x,y_1+y_2) \]

\[ \text{subplot}(2,2,3) \]

\[ \text{plot2d}(x,y_1+y_2+y_3) \]

\[ \text{subplot}(2,2,4) \]

\[ \text{plot2d}(x,y_1+y_2+y_3+y_4) \]
10.1.5 Titles and Captions

Titles and captions can be added, after a graph has been drawn, using the `xtitle` command. It takes three arguments – the title for graph, the caption for the x-axis and the caption for the y-axis. If you don’t need a title just use a blank string, i.e. ""

```matlab
-->x = -20:0.01:20;

-->clf, plot2d(x, sin(x)./x)

-->xtitle('A TITLE', 'x', 'sin(x)/x')
```

![Graph with titles and captions added](image)
Note the unusual placement of the axis labels. There is nothing that I am aware of that you can do about this.

10.1.6 Other Features

There a lot more Scilab can do with graphs, for example

```scilab
-->clf, plot2d(x, sin(x)./x, rect=[-20 -0.5 20 1.5], ...
-->axesflag=5)
```

![Graph example](image)

In this example, the `rect` option sets the bounds on the \( x \) and \( y \) axes and the `axesflag=5` places the axes so that they cross at the origin \((0, 0)\).

The latest version (3.1.1) of Scilab seems to have much enhanced graphics facilities. Try Scilab’s help if you want more information.

10.1.7 3D Curves

Curves in 3 dimensional space can be plotted using `param3d`. It takes three vectors containing the values the \( x \), \( y \) and \( z \) coordinates of the points on the curve. By clicking on the 3D Rot button on the graphics window and playing around with the mouse you can alter the orientation of the graph.

Here is a 3-D spiral:

```scilab
-->z = 0:0.01:10;

-->clf, param3d(z.*sin(5*z), z.*cos(5*z), z)
```
10.1.8 Histograms

Histograms can be plotted with the `histplot(n, data)` command. Here `n` is the number of bins in the histogram and `data` is the vector of data for which we want to draw the histogram. The following example draws a histogram from a vector of normally distributed random numbers.

```matlab
--> rr = rand(1,100000,'normal');

--> clf, histplot(100, rr)
```
10.2 Floating Point Arithmetic

10.2.1 Representation of Floating Point Numbers

The arithmetic used by Scilab is called floating point arithmetic. Scientific notation is used to express floating point numbers; the number \(1.2345 \times 10^{-6}\) is written in Scilab as \(1.2345e-6\).

Internally floating point numbers are stored in a binary format known as IEEE double precision arithmetic. Each floating point number occupies 64 bits, which contain the sign, digits and exponent of the number. The details of the binary representation are not usually important in practical applications, but the finite precision of floating point numbers has some important implications:

1. Each floating point number is capable of representing about 16 decimal digits (actually 53 bits).

2. There is a limit on the range of exponents; from about \(10^{-308}\) to \(10^{+308}\) (actually \(2^{-1022}\) to \(2^{+1023}\)).

The most important consequence of this is that it is impossible to calculate with more than 16 digits precision. This, in itself, is not a problem since it is rare in practical applications to require more accuracy than this. However it can happen that much less accuracy is attained in actual calculations due to rounding errors.

10.2.2 Precision

The usual measure of precision in floating point arithmetic is the number machine epsilon, written \(\varepsilon_{\text{mach}}\), and defined to be difference between the number 1 and the next largest floating point number. In IEEE arithmetic the number 1 has the representation

\[
\begin{array}{c}
1 \cdot 0 \ 0 \ 0 \ \ldots \ 0 \ 0 \ 0 \\
52\text{bits}
\end{array}
\]

and the next largest floating point number is

\[
\begin{array}{c}
1 \cdot 0 \ 0 \ 0 \ \ldots \ 0 \ 0 \ 1 \\
52\text{bits}
\end{array}
\]

Thus

\[
\varepsilon_{\text{mach}} = 2^{-52} \approx 2.220 \times 10^{-16}
\]
In Scilab $\varepsilon_{\text{mach}}$ is denoted by $\%\text{eps}$.

```plaintext
-->%eps
%eps  =
2.220E-16
```

To check the defining property of $\varepsilon_{\text{mach}}$, note that

```plaintext
--> (1 + %eps) - 1
ans  =
2.220E-16
```

```plaintext
--> (1 + %eps/2) - 1
ans  =
0.
```

In floating point arithmetic if we add $\varepsilon_{\text{mach}}$ to 1 we get a number different to 1, while if add half this much the result is still 1.

### 10.2.3 Exceptional Values

Besides the usual floating point numbers, IEEE arithmetic has two special numbers $\text{Inf}$, for ‘infinity’, and $\text{Nan}$, for ‘not-a-number’. $\text{Inf}$ typically occurs when we try try to produce a number whose exponent is greater than the maximum exponent:

```plaintext
--> z = 1e300
z  =
1.000+300
```

```plaintext
--> w = z*z
w  =
Inf
```

This is called overflow. It is easy to recognize when overflow has occurred since an $\text{Inf}$ always results.

If the result of a calculation has exponent less than the smallest exponent, the result is set to zero. This is called underflow and, while it is usually harmless it can sometimes cause unexpected errors.
\[ z = 1e^{-300} \]
\[ w = z \times z \]

The following shows some ways \texttt{Nan}'s can arise:

\[ z = (1e300)^2 \]
\[ w = 0 \times z \]
\[ w = z - z \]

10.2.4 Rounding

When arithmetic operations are performed on floating point numbers the exact result will not, in general, be representable as a floating point number. The exact result will be rounded to the nearest floating point number, a process called, obviously enough, \textbf{rounding}. In IEEE arithmetic all results of floating point operations are correctly rounded, something that was rarely done on early computers.

We will look at an example to give an idea of the rounding process using base 10 arithmetic (although as pointed out above computers use binary arithmetic). Suppose we are doing arithmetic with, say, 5 decimal digits precision. Let

\[ x = 1.2345 \times 10^6 \]
\[ y = 6.7890 \times 10^3 \]

\(^1\)In the case of a tie, the round to even rule is used.
then, in exact arithmetic,

\[
\begin{align*}
  x + y &= 1.241289 \times 10^6 \\
  x - y &= 1.227711 \times 10^6 \\
  x \times y &= 8.3810205 \times 10^9
\end{align*}
\]

and these are rounded to

\[
\begin{align*}
  x + y &= 1.2413 \times 10^6 \\
  x - y &= 1.2277 \times 10^6 \\
  x \times y &= 8.3810 \times 10^9
\end{align*}
\]

respectively.

Rounding errors are unavoidable in numerical computation due to the finite precision of floating point arithmetic. The relative error due to rounding is less than \(1/2\varepsilon_{\text{mach}} \approx 1.1 \times 10^{-16}\), but this can be expected to occur in every floating point operation, including binary/decimal conversion on reading/writing floating point numbers.

10.2.5 Cancellation

Cancellation error occurs when two nearly equal numbers are subtracted. Consider the calculation

\[
\begin{align*}
  \text{-->} x &= 3.141592653589793 \\
  x &= 3.1415927 \\
  \text{-->} y &= 3.141592653585682 \\
  y &= 3.1415927 \\
  \text{-->} x-y \\
  \text{ans} &= 4.111\text{E-12}
\end{align*}
\]

Here \(x\) is a 16 digit approximation to \(\pi\) while \(y\) agrees with \(\pi\) to 12 digits. What is the error in the calculation?

We know the relative error due to rounding is less than \(1/2\varepsilon_{\text{mach}} \approx 1.1 \times 10^{-16}\). The problem is that in converting \(x\) and \(y\) from decimal to
binary, that is approximating $x$ and $y$ by floating point numbers, we have relative errors of the same magnitude. Thus the absolute error in $x$ is

$$E_{\text{abs}} \approx 1/2\varepsilon_{\text{mach}}x \approx 3.5 \times 10^{-16}$$

with the same size error in $y$. We would expect the same size error in their difference, giving a relative error of about

$$E_{\text{rel}} \approx 3.5 \times 10^{-16}/4.1 \times 10^{-12} = 8.5 \times 10^{-5}.$$  

Any number that is not an exact floating point number, and this includes the results of almost all calculations, will have a relative error of the order of $1/2\varepsilon_{\text{mach}}$. Whenever two such numbers which are nearly equal are subtracted, cancellation error will occur giving a result with less precision than the original numbers.

**Example**

An instructive example of cancellation error occurs when we approximate the derivative of function by a formula such as

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

As $h$ gets smaller and smaller the approximation gets better until in the limit as $h \to 0$ we get the definition of the derivative.

Computing with this formula has the numerical problem that as $h$ gets smaller, $f(x + h)$ and $f(x)$ get closer together and cancellation error will occur. In fact if $h$ is small enough, $f(x + h)$ and $f(x)$ will be represented by the same floating point numbers giving the approximation $f'(x) \approx 0$.

We will take $f(x) = \sin x$ and approximate the derivative at $x = 1$ using values of $h$ from $10^{-2}$ down to $10^{-16}$:

--->n = -2:-2:-16
n =


--->h = 10 .^n
h =

1.0E-08 *
Note (1) the use of the dot operator and (2) how Scilab puts a scale factor outside of the array.

```scilab
-->approx = (sin(1+h) - sin(1))./h
approx =

column 1 to 4
! 0.5360860  0.5402602  0.5403019  0.5403023 !

column 5 to 8
! 0.5403022  0.5403455  0.5440093  0.     
```

The error in the approximation is:

```scilab
-->err = approx - cos(1)
err =

column 1 to 4
! -0.0042163 - 0.0000421 - 0.0000004 - 2.970E-09 !

column 5 to 8
! -5.848E-08  0.0000432  0.0037070 - 0.5403023 !
```

At first the main cause of error is the approximation itself and the error decreases as \( h \) decreases. At \( h = 10^{-10} \) the error begins to increase. The main cause of error now is cancellation and the error increases rapidly until at \( h = 10^{-16} \) the approximate derivative is zero and totally useless.