Introduction to SciLab
Version 1.1

Peter Urban
additions by Kai Hagenburg

May 4, 2009

Contents

Chapter 1

Introduction

1.1 About this document

This short tutorial was written for students attending the *Numerical Algorithms in Computer Vision III* course at the Saarland University and will therefore only deal with a small subset of the SciLab features.

1.1.1 Typographic Conventions

Throughout the text some text styles are employed to characterize some words which have a special meaning.

Example	Utilization
LU decomposition	mathematical concepts
File/Close	buttons and menu entries in the graphical user interface
Page Up	keyboard and mouse keys
exp(rand(10,10))	source code

For functions and function arguments a special syntax similar to the one in used in the build-in help system is used. A dummy example of a function prototype is demofunc(<str>,[scalar,[mat]]). This function requires at least the string argument <str> to be specified and takes optionally the scalar argument scalar and a matrix or vector [mat].

1.2 What is SciLab

SciLab is an environment designed to solve numerical problems occurring in various application areas like science and engineering. It was developed at the I.N.R.I.A. (Institut National de Recherche en Informatique et Automatique) and is available as open source since 1994 under the CeCILL[1] license with modules licensed under the GPL[2]. SciLab is very similar to the MatLab and Octave environments but comes with several additions especially when it comes to graphics and integration of other languages like C/C++ and Fortran. Currently, the SciLab is in version 5.1.1, available for Linux and Windows environments. There also exist versions for MacOSX 10.5, but this is still an alpha version.

[1] A french open source license compatible to the GPL. The acronym stands for **Ce**A**C**nrs**I**NRIA**L**ogiciel**L**ibre.

[2] GPL stands for GNU Public License

1.2.1 Online documentation

Several free documentations in different languages can be found on the SciLab website next to the online-help. A well readable document is *Introduction à Scilab* written by Bruno Pinçon, which is also available in a good german translation. English readers are referred to F. Haugens *Master Scilab & Scicos*.

1.2.2 SciLab help

SciLab is equipped with a powerful and complete help system. It can be either accessed by clicking the menu entry `?/SciLab Help` or via the command `help()`. The function `help(key)` jumps directly to a topic and `apropos(key)` searches the help for a keyword given in the string variable `key`.

```
-->help("help") // display help for the help command
```

1.3 Design principles

The SciLab environment offers an interpreted programming language with a syntax designed for the formulation of numerical algorithms by staying close to mathematical notations in linear algebra. As in most scripting environments commands are either entered one after another in the shell or loaded from a file containing a script or a function. Although the language is focused on mathematical problems it is even possible to use it for complete graphical applications. Community and commercial contributors have released a wide range of plugins which extend the use of SciLab to application areas like image and video processing, optimization, real-time controlling or FEM simulations.

1.4 Getting started

After installation on your platform the SciLab environment can be started by entering `scilab` on your system terminal or clicking the corresponding icon in the applications menu. The main window contains the menu, the toolbar and the command line which shows the prompt `-->` to signalize readiness.

1.4.1 The command shell

The shell is the central point from where the environment is controlled. Expressions can be entered in the shell and are then evaluated as soon as the Enter key is pressed. After the evaluation of the last command is completed the prompt is displayed to show that SciLab is waiting for new user input. The Enter key inserts a newline character which finishes the current expression and prints the result and eventually other relevant information like errors in the command shell.

```
-->a=1
 a  =

    1.

-->b=a+2
```

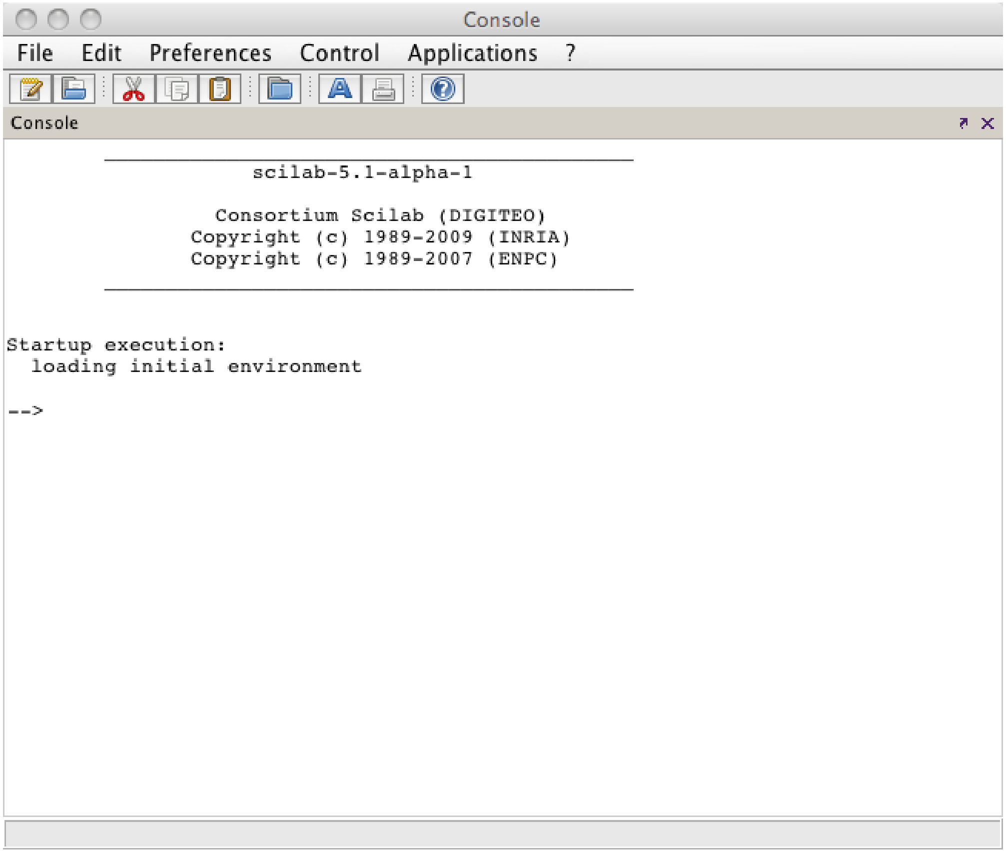

Figure 1.1: The SciLab window with the command prompt.

```
 b  =

    3.
```

To suppress the output of the last expression a semicolon can be appended at the end of a line:

```
-->b
 b  =

    3.

-->b;
```

If more than one statement per line is desired several statements can be separated with a comma. The expressions are evaluated from left to right.

```
a=1,b=a+1
 a  =

    1.
 b  =

    2.
```

If you like to extend an expression to the next line append `...` at the end of the current line.

```
-->b = 4 ...
-->   +5 ...
-->   -3 ...
-->   +1
 b  =

    7.
```

Command passed to SciLab are saved in a command history and can be recalled by pressing the Cursor Up and Cursor Down keys.

ans

In case the result of an expression is not assigned to a variable as done in the previous examples it will be assigned to a the special variable `ans` and can be read until it is overwritten by the result of the next non-assignment expression:

```
-->1
 ans  =

    1.
-->ans+1
 ans  =

    2.
```

Comments

In order to keep the source code readable comments can be introduced. All characters between // and the newline character are ignored by the interpreter.

```
-->n=5, n_fac=prod(1:n) // the factorial of n
 n  =

    5.
 n_fac  =

    120.
```

1.4.2 Scripts and functions

For programming and longer calculations it is beneficial to encapsulate the tasks in the form of scripts and functions which are then saved as text files readable with a normal text editor. If SciLab is build with Tcl/Tk support, the SciPad editor is included in the environment which can be launched with the `scipad()` or `scipad(file1[,...,fileN])` command or by clicking the `Applications/Editor` menu entry. It offers full support for SciLab like syntax highlighting, code completion and debugging. Many popular editors like emacs and vim also support the SciLab syntax. For more details on how to define your own scripts and functions see 3.4 and 3.3.

Chapter 2

Matrices and Linear Algebra

The most important data type in SciLab is a complex floating point matrix. It is not only used to represent matrices but also scalars by means of (1×1)-matrices and vectors as (1×n)- or (m×1)-matrices. This makes it easy to perform operations like accessing and assigning rows/columns of matrices because they only deal with one data type. In this text the term *matrix* can also refer to vectors and scalar unless stated otherwise.

2.1 Initialization

Variables which are used in the shell or in a script file always have the data type of the value used to (re-)initialize it so that it is not necessary to declare the data type explicitly. *Note:* The identifiers of variables consist of up to 24 characters. If a longer name is given only the first 24 characters are used.

2.1.1 Scalars

Scalars can be specified as normal floating point values or as complex numbers using the imaginary unit which is represented by the build-in variable `%i`

```
-->realScalar = 1  //scalars always have floating point precision ...
 realScalar  =

    1.

-->complexScalar = 1 + 1 * %i  //complex values use imaginary unit %i
 complexScalar  =

    1. + i

-->real(realScalar+complexScalar)  //the real part of the sum
 ans  =

    2.

-->imag(realScalar+complexScalar)  //the imaginary part
```

```
 ans  =

    1.
```

2.1.2 Vectors

Vectors are specified as a list of scalars, where the delimiters determine the layout of the vector. Elements in comma- or whitespace-separated lists form a row-vector:

```
-->rowVector = [1,2,3]
 rowVector  =

    1.    2.    3.

-->sameRowVector = [1 2 3]
```

whereas semicolons separate elements of column-vectors.

```
-->columnVector = [1.;2.;3.]
 columnVector  =

    1.
    2.
    3.
```

2.1.3 Matrices

The syntax to create a matrix is a straight forward extension of the vector syntax. The row vectors of the matrix are assigned to the elements of a column vector.

```
-->columnRowMatrix = [ [11,21,31] ; [12,22,32] ; [13,23,33] ]
 columnRowMatrix  =

    11.    21.    31.
    12.    22.    32.
    13.    23.    33.
```

which can be also written without the inner brackets and with whitespaces instead of commas.

```
-->sameColumnRowMatrix = [ 11,21,31 ; 12,22,32 ; 13,23,33 ]
-->equalColumnRowMatrix = [ 11 21 31 ; 12 22 32 ; 13 23 33 ]
```

Alternatively it is possible to construct the same matrix as a row vector containing column vectors.

```
-->rowColumnMatrix = [ [11;12;13] , [21;22;23] , [31;32;33] ]
```

Note: All of the above examples yield the same matrix. To exemplify that scalars, vectors and matrices are represented by the same data structure note that the vector notation is equivalent to that of (1×n)- or (m×n)-matrices. Similarly, a scalar can be specified as a vector or matrix with just one element.

```
-->scalar = [ 1 ] // scalar as vector with one element
 scalar  =

    1.

-->anotherScalar = [ [ 1 ] ] // or even as a matrix
 anotherScalar  =

    1.

-->scalar(1,1) + anotherScalar // scalars and vectors are matrices!
 ans  =

    0.
```

2.1.4 Creating an identity matrix

The frequently needed identity matrix is returned by `eye(m,n)` if the matrix dimensions `m` and `n` agree. In the general case the matrix $[\delta_{ij}]_{i=1...m,j=1...n}$ is created.

```
-->identityMatrix = eye(3,3)
 identityMatrix  =

    1.    0.    0.
    0.    1.    0.
    0.    0.    1.
```

Note: Instead of specifying the dimensions `m` and `n` explicitly it is possible to pass a matrix as an argument from which the size of the new matrix is taken. This applies to all functions introduced in this section.

2.1.5 Creating matrices containing zeroes and ones

Other useful functions are `zeros(m,n)` and `ones(m,n)` which create the matrices $[u_{ij} = 0]_{i=1...m,j=1...n}$ and $[u_{ij} = 1]_{i=1...m,j=1...n}$ respectively.

```
-->zeroMatrix = zeros(2,2)
 zeroMatrix  =

    0.    0.
    0.    0.
-->oneMatrix = ones(zeroMatrix) // size is taken from zeroMatrix
 oneMatrix  =

    1.    1.
    1.    1.
```

2.1.6 Creating diagonal matrices

Given a vector b of size n, the application of the function `diag(b)` gives a $n \times n$-Matrix with the diagonal entries $a_{ii} = b_i$. It should be noted that the function is also an overloaded and we will have a look at the second function in a later section.

```
-->A=diag([1 2])
 A  =

    1.    0.
    0.    2.
```

2.1.7 Creating matrices containing random numbers

A matrix containing random numbers is provided by the `rand(m,n)` function. The generated numbers are equally distributed in the closed interval $[0, 1]$. For a gaussian distributed elements `rand("normal")` is called, `rand("uniform")` switch back to uniform distribution. The current distribution is returned as a string by `rand("info")`.

```
-->uniformVector = rand(1,5)
 uniformVector  =

    0.5608486    0.6623569    0.7263507    0.1985144    0.5442573

-->rand("normal") // switch to N(0,1) distribution

-->normalVector = rand(1,5) // normal distributed (1x5)-matrix
 normalVector  =

  - 0.7414362  - 0.7437914  - 0.2589642    0.3501626    1.0478272

-->distribution = rand("info")  // get current distribution key
 distribution  =

 normal
```

Note: More advanced and configurable random number generators are provided by `grand(m,m,dist_type [,p1,...,pk])`. See the online help for details.

2.1.8 Interval sampling and the colon operator

This section deals with commands which generate vectors containing a sequence of equidistant numbers. The most frequently used sequences are subsets of the natural numbers.

$$[a, a + 1, ..., b - 1, b] \text{ where } a, b \in \mathbb{N}$$

Such sequence vectors are created by statements involving the "`:`"-operator of the form `a:b` with $a < b$ were `a` and `b` are integers. This expression returns a vector with all natural numbers in the interval $[a, b]$. In the most general case a *sampling distance* and arbitary interval bounds are specified. The SciLab expression then becomes `a:l:b`.

```
-->oneToTen = 1:10 // a vector with ten elements
 oneToTen  =

    1.    2.    3.    4.    5.    6.    7.    8.    9.    10.

-->oneToFive = 1:.5:5 // this result has nine elements
```

```
 oneToFive  =

    1.    1.5    2.    2.5    3.    3.5    4.    4.5    5.

-->oneToAlmost = 1:.51:5 // only eight elements here
 oneToAlmostFive  =

    1.    1.51    2.02    2.53    3.04    3.55    4.06    4.57
```

Note: If for the expression `a:l:b` we have $\frac{b-a}{l} = n \notin \mathbb{N}$ then b is not an element of the resulting vector. In this case the last element is $a + \lfloor n \rfloor l$.

The function `linspace(a,b[,n])` takes the points a,b of the closed interval $[a, b]$ and the number of sample points n as arguments and returns a vector containing a, b as well as $n - 2$ samples that subdivide $[a, b]$ into equally spaced intervals. The number of samples `n` is set to 100 if not specified.

```
--> linspace(0,10,7)
 ans  =

    0.    1.6666667    3.3333333    5.    6.6666667    8.3333333    10.
```

The similar function `logspace(a,b[,n])` samples with logarithmic sampling distance in the interval $[10^a, 10^b]$.

```
--> logspace(0,2,3)
 ans  =

    1.    10.    100.
```

2.2 Matrix Algebra

The elementary operations in linear algebra $+,-,\times,/$ and the power function are easily accessible in the SciLab language.

2.2.1 Addition

The sum of two matrices is computed with the “`+`”-operator if the matrix dimensions of the two operands agree.

```
-->a = [ 1 2 3; 1 2 3; 1 2 3] , b=a’
 a  =

    1.    2.    3.
    1.    2.    3.
    1.    2.    3.
 b  =

    1.    1.    1.
    2.    2.    2.
    3.    3.    3.
```

```
-->matrixSum = a + b
 matrixSum  =

    2.    3.    4.
    3.    4.    5.
    4.    5.    6.
```

2.2.2 Multiplication

Matrix multiplication is done with the "`*`"-operator:

```
-->matrixProduct = a * b
 matrixProduct  =

    14.    14.    14.
    14.    14.    14.
    14.    14.    14.

-->a(1,:) * a(1,:)'  // the norm of the first row of "a"
 ans  =

    14.

-->a(1,:)' * a(1,:)  // the outer product of the row vector a(1,:)
 ans  =

    1.    2.    3.
    2.    4.    6.
    3.    6.    9.
```

2.2.3 Powers

The "`^`"-operator computes the matrix power if the first operator is a square matrix and the second argument a scalar value. If the second argument `p` is an integer the matrix is multiplied `p` times with itself. If `p` is a floating-point number then diagonalization is used.

```
-->A = [ %pi 0 ; 0 2 ]^2 // requires one matrix multiplication
 A  =

    9.8696044    0.
    0.           4.

-->A^.5 // requires eigenvalue decomposition
 ans  =

    3.1415927    0.
    0.           2.
```

Note: In other cases the element-wise[1] power is computed.

[1]Element-wise operations are explained in section 2.3.

- For a non-square $(m \times n)$-matrix `[A]` the expression `[A]^b` and `[A].^b` returns $[A_{ij}^p]_{i=1\ldots m, j=1\ldots n}$.
- With a scalar `a` and a matrix `[B]` as operands `a^[B]` and `a.^[B]` yield the matrix $[a^{B_{ij}}]$.
- If `[A]` and `[B]` are matrices of the same size then the expression `[A]^[B]` returns $[A_{ij}^{B_{ij}}]$.

2.2.4 Transpose

The transpose is returned if a "`'`" follows the matrix expression.

```
-->[1 2 3]' // creating a column vector from a row vector
 ans  =

    1.
    2.
    3.

-->[1 2 ; 3+%i 4+%i]'
 ans  =

    1.    3. - i
    2.    4. - i
```

Note: For complex-valued matrices this yields the conjugate transpose. The non-conjugate transpose is returned by the "`.'`"-operator.

```
-->[ 1 2 ; 3+%i 4+%i].' //non-conjugate transpose
 ans  =

    1.    3. + i
    2.    4. + i
```

2.3 Element-wise operations

Although in classical linear algebra no element-wise operations except addition are defined, they can become handy for syntactical reasons. SciLab has versions of multiplication, division and the power operator that take two equally-sized matrix arguments or a matrix and a scalar and return a new matrix.

```
-->a = [0 1 ; 2 3] + 1 // addition works always element-wise:  a(i,j)+1
 a  =

    1.    2.
    3.    4.

-->a ./ a'  // element-wise division by the transpose of a:  a(i,j)/a(j,i)
 ans  =

    1.     0.6666667
    1.5    1.
```

```
-->a .* a  // element-wise multiplication:  a(i,j) * a(i,j)
 ans  =

    1.    4.
    9.    16.
```

Note: The expression `1./[A]` is not evaluated as an element-wise operation, instead `1.` is interpreted as a number and a *right division with feed back*[2] is performed. To get the matrix $[1/A_{ij}]$ the expression `1 ./[A]` or `(1.)./[A]` have to be used.

```
-->1./a  // the slash operator performs right division
 ans  =

  - 2.     1.
    1.5  - 0.5

-->1 ./a  // the element-wise operation: 1/a(i,j)
 ans  =

    1.           0.5
    0.3333333    0.25
```

2.4 Solving linear systems of equations

Most methods in engineering and physics lead to large linear systems of equations $Ax = b$. Although there are well established methods to solve such problems it is extremely time-consuming if done by hand and often specialized algorithms have to be used in order to get acceptable solutions. This is the reason why huge parts of SciLab are devoted to this problem.

2.4.1 The backslash operator

The standard way to solve linear equation systems with SciLab is to use the \-operator. This is the same as using the function `lusolve([A],[b])` which takes the non-singular square matrix `[A]` and the right hand side `[b]` as arguments. The equations are solved by *LU decomposition* that is especially useful for sparse matrices.

```
-->A=rand(3,3); b=rand(3,1);

-->x = A \ b  // solves A x = b
 x  =

    1.0057916
  - 2.2502357
    0.8202812

 -->norm(A*x-b) //  the residuum requires the vector norm
 ans  =
```

[2]see the online help for details

```
   1.241D-16
```

Note: If A is not a square matrix then the *least square solution* is returned. If A has full column rank the solution

$$\underset{x}{\operatorname{argmin}} \|Ax - b\|$$

is unique, if not the solution will in general not minimize $Ax - b$.

2.4.2 The inverse

Another way to solve a linear system of equations $Ax = b$ is to use the inverse A^{-1} of the square matrix A. Using the fact that $AA^{-1} = \mathbb{I}$ the solution x is obtained from $x = A^{-1}b$. The inverse is calculated with the `inv([A])` function.

```
-->Ainv = inv(A);

-->x1 = Ainv*b
 x1  =

   1.0057916
 - 2.2502357
   0.8202812

-->norm(A*x1-b)
 ans  =

   2.719D-16

-->x2 = Ainv*rand(3,1) // solution for another right hand side
 x2  =

   0.4542787
   2.1606781
 - 0.3947949

-->norm(A*x2-b)
 ans  =

   0.5435923
```

Note: The computation of the inverse is much more expensive since for a n-dimensional matrix A solving $AA^{-1} = \mathbb{I}$ requires the solutions for n right hand sides.
Note: For singular and non-square matrices the function `pinv([A])` returns the *pseudo inverse* (Moore-Penrose Inverse) of the matrix which is computed by using *singular value decomposition.*

2.5 Accessing matrices

SciLab provides a very powerful syntax for accessing elements and extracting sub-matrices.

2.5.1 Referencing matrix elements

To reference an element of a n-dimensional vector the index $i \in 1, ..., n$ is written in parenthesis after the name of the variable.

```
-->squares = (1:5)^2
 squares  =

    1.    4.    9.    16.    25.

-->squares(3)
 ans  =

    9.
```

For matrix elements a pair of indices has to be specified, the rows are addressed by the first and the columns by the second index.

```
-->mat=eye(3,3);

-->mat(1,3)=-1
 mat  =

    1.    0.  - 1.
    0.    1.    0.
    0.    0.    1.

-->mat(2,2)
 ans  =

    1.
```

Note: Internally matrices are stored as vectors such that the columns are aligned linear one after another. It is possible to access matrix elements by using only one index. To return the element a_{uv} from the matrix $[a_{ij}]_{i=1...m,j=1...n}$ the index $u + (v - 1) * n$ has to be used.

```
-->mat(7)
 ans  =

  - 1.
```

2.5.2 Referencing submatrices

In SciLab it is also possible to easily extract submatrices by using vectors of indices. For a pair of index vectors $\vec{i} = i_1, ..., i_p$ and $\vec{j} = j_1, ..., j_q$ used to address elements of the matrix A the result is a $(p \times q)$-matrix $[b_{uv} = a_{i_u j_v}]_{u=1...p,v=1...q}$.

```
-->mat([1 2 3],[1 3])  // this references the first and the last column
 ans  =

    1.  - 1.
    0.    0.
    0.    1.
```

The colon operator described in section 2.1.8 can be used to elegantly create index vectors containing index intervals. Whole rows and columns can be addressed by a single colon in the corresponding index slot.

```
-->mat(:,[1 3]);  // same as before

-->big = rand(100,100);

-->smaller = big(1:10,10:10:100) // the first ten row and every tenth column element
```

2.5.3 size and length

To find out the dimensions of a matrix or a vector you can use the `size([A][,dim])` function. It returns a vector containing the size of every dimension or the number of elements for a particular dimension specified by `dim`.

```
-->size(smaller) // "smaller" is a 10x10 matrix
 ans  =

    10.    10.

-->size(big,1) // size of the first dimension
 ans  =

    100.
```

The total number of elements is returned by `length([A])`

```
-->length(big)
 ans  =

    10000.
```

Note: `length` also determines the number of characters in strings (3.1.2) and the number of elements in a list (3.1.3).

2.5.4 Extracting the diagonal of a matrix

Next to the functionality explained in section 2.1.6 the `diag([M])` function extracts the diagonal if the argument is a matrix and returns it as a column vector.

```
-->diag(rand(3,3))'  // the transposed diagonal of a 3x3 random-number matrix
 ans  =

    0.8596608    0.5111992    0.2596145
```

2.5.5 Extracting triangle matrices

The upper and lower triangle (including the diagonal) of a matrix can be separated with the commands `triu([M])` and `tril([M])` which return a matrix of the same size as the input matrix `[M]` with all elements not belonging to the respective triangle matrix set to zero.

```
-->numbers = ones(10,10);

-->numbers(1:100)=1:100;

-->tril(numbers)
 ans  =

    1.     0.     0.     0.     0.     0.     0.     0.     0.     0.
    2.     12.    0.     0.     0.     0.     0.     0.     0.     0.
    3.     13.    23.    0.     0.     0.     0.     0.     0.     0.
    4.     14.    24.    34.    0.     0.     0.     0.     0.     0.
    5.     15.    25.    35.    45.    0.     0.     0.     0.     0.
    6.     16.    26.    36.    46.    56.    0.     0.     0.     0.
    7.     17.    27.    37.    47.    57.    67.    0.     0.     0.
    8.     18.    28.    38.    48.    58.    68.    78.    0.     0.
    9.     19.    29.    39.    49.    59.    69.    79.    89.    0.
    10.    20.    30.    40.    50.    60.    70.    80.    90.    100.

-->triu(numbers)
 ans  =

    1.    11.    21.    31.    41.    51.    61.    71.    81.    91.
    0.    12.    22.    32.    42.    52.    62.    72.    82.    92.
    0.    0.     23.    33.    43.    53.    63.    73.    83.    93.
    0.    0.     0.     34.    44.    54.    64.    74.    84.    94.
    0.    0.     0.     0.     45.    55.    65.    75.    85.    95.
    0.    0.     0.     0.     0.     56.    66.    76.    86.    96.
    0.    0.     0.     0.     0.     0.     67.    77.    87.    97.
    0.    0.     0.     0.     0.     0.     0.     78.    88.    98.
    0.    0.     0.     0.     0.     0.     0.     0.     89.    99.
    0.    0.     0.     0.     0.     0.     0.     0.     0.     100.
```

Chapter 3

Programming with SciLab

All constructs described so far allow you to use SciLab as a matrix calculator. In order to implement numerical algorithms other language features like loops, branches and functions are necessary.

3.1 Other data types

3.1.1 Boolean expressions

Boolean values are represented by the build-in variables `%t` or `%T` for true and `%f` or `%F` for false statements. Matrices containing boolean elements are defined and accessed the same way as floating-point matrices

```
-->boolMatrix = [%t %t %t; %t %f %f; %t %t %t]
 boolMatrix  =

  T T T
  T F F
  T T T

-->boolMatrix(2,:)
 ans  =

  T F F
```

The well known boolean operators work element-wise with two scalar, vector or matrix operands of the same size

`&`	logical *AND*
`|`	logical *OR*
`~`	logical negation

```
-->boolMatrix & boolMatrix'
 ans  =

  T T T
```

```
  T F F
  T F T

-->boolMatrix | boolMatrix'
 ans  =

  T T T
  T F T
  T T T

-->~boolMatrix
 ans  =

  F F F
  F T T
  F F F
```

3.1.2 Character Strings

String literals are defined by enclosing the text between two single or double quotes.

```
-->a = "typical character string"
 a  =

 typical character string
```

String matrices are created using the usual syntax.and have the + operator defined which concatenates the character strings element-wise.

```
-->a + 's can be concatenated easily.'
 ans  =

 typical character strings can be concatenated easily.
```

Extraction of characters and sub-strings is achieved by the `part(string,[indices])` function.

```
-->part(a, [ 6 , 8 ,19:25 ] )
 ans  =

 a string
```

3.1.3 Lists

The list data type in SciLab is used to define variables representing ordered collections of objects which can be of different types like floating-point, boolean, string matrices or even other lists. To create a list the `list(element1,element2,...)` function with the desired elements is called. It creates a empty list if no arguments are passed.

```
aList = list([%t,%f],["bunch","of","objects"],eye(2,2))
 aList  =
```

```
       aList(1)

  T F

       aList(2)

!bunch  of  objects  !

       aList(3)

   1.    0.
   0.    1.
```

To read out all elements at the same time the same number of variables as list elements is given in row vector notation (2.1.2) as the left hand side and the list as the right hand side of the assignment.

```
-->[totally,different,things] = aList(1:3)
 things  =

   1.    0.
   0.    1.
 different  =

!bunch  of  objects  !
 totally  =

  T F
```

Individual elements are referenced the same way as vector elements (2.5.1).

```
-->aList(2)
 ans  =

!bunch  of  objects  !
```

3.2 Entering multi-line statements

Programming constructs like functions, conditions and loops extend over multiple lines. There are several ways to enter such statements.

- It is possible to define a script or multiple functions per file in order to reuse the saved code. Before you can call these scripts and functions they have to be loaded into the SciLab environment by prepending the `getf(path)` or `exec(path)` command.

  ```
  -->getf("func.sci") // define function(s) in file
  -->exec("script.sce") // execute file (also defines contained functions)
  ```

  Note: As when working with the shell the newline character inserted by the ENTER key as well as "," and ";" separate commands.

- Multi-line statements are usually enclosed in keywords (like `function ... endfunction`). After the first keyword is processed by the shell all further commands are deferred until the second keyword closes the statement.

  ```
  -->if rand()<.1
  -->  "10% probability to see this"
  -->else
  -->  "90% probability to see this"
   ans  =

   90% probability to see this
  ```

- Instead of using multiple lines the commands can be written in one line and separated by "," or ";".

  ```
  -->function result = myFactorial(num), result=prod(1:num) , endfunction
  ```

3.3 Functions

User-defined functions encapsulate lengthy computations and are saved to files with the extension `.sci` for later use. They take *input arguments* when called and return the computed *output arguments*. The definition of a user-defined function follows the schema:

```
function [result1,result2] = newFunction(argument1,argument2)

\\some computations are done here

result1 = expression(argument1,argument2)  // result one is assigned
result2 = expression(argument1,argument2)  // the second output argument is assigned

endfunction
```

The function header starts with the `function` keyword. An arbitrary number of output arguments `result1`, `result2` ... are defined as a list followed by the `=` sign, function name and input arguments given in parenthesis. The body of the function contains the actual commands and the assignment of the results to the output arguments. The definition of the function is completed with the `endfunction` keyword.
Note: If only one output argument is used the brackets of the list can be omitted.
The same syntax as for the extraction of list elements is used to assign the output arguments to variables in the SciLab environment.

```
-->rand("normal");

-->function [ statMean , statVariance ] = stats(data)
-->
-->statMean        = sum(data)/length(data);
-->statVariance    = sum((data-statMean).^2)/(length(data)-1);
-->
-->endfunction
```

```
-->randVec = rand(1000,1);

-->[randMean,randVar] = stats(randVec)
 randVar  =

    0.9603543
 randMean  =

  - 0.0169296

-->stats(randVec)  // ans only gets the value of the first output variable
 ans  =

  - 0.0169296
```

3.4 Scripts

Long command sequences or programs which are frequently used can be saved in a file to execute them from the shell whenever it is needed. Scripts can be written with your favorite text editor and are usually saved under the extension `.sce` to associate it with the SciLab executable.

To execute the commands saved in a script it is either possible to select the `File/Execute...` menu entry or to use the `exec(path)` which takes the filename as a string argument (enclosed in quotes or double quotes).

3.4.1 User input

In scripts limited user interaction is realized by `input` which works fully transparent to the interpreter. The function prompts the user for input that is then evaluated as an ordinary SciLab expression.

```
-->input("What is the ...?")
What is the ...? ["question","matter","result"]
 ans  =

!question  matter  result  !
```

3.5 Conditional execution

A basic concept in procedural programing is *conditional execution* were a piece of code is executed only if a given boolean expression (condition) returns *true* (the value `%T`).

3.5.1 Relations and conditions

Conditions are mostly formulated by means of the *binary relations*

==	equal
>	greater than
<	smaller than
>	greater or equal
<	smaller or equal
~= or <>	not equal

which also take two matrices or a matrix and a scalar as operands.

```
-->A=rand(2,3),B=rand(2,3)
 A  =

    0.9258019    0.0203203    0.0785347
    0.8723090    0.3337130    0.2945725
 B  =

    0.5671068    0.3854331    0.0589854
    0.1304469    0.1535310    0.1093552

-->A<B
 ans  =

  F T F
  F F F

-->A>.3
 ans  =

  T F F
  T T F
```

The `find([M])` function is used to return the indices of those matrix elements which match a certain criterion, namely those which are equal `%T` (true).

```
-->A=rand(3,3)
 A  =

    0.9208180    0.5974132    0.7683759
    0.3971987    0.7759346    0.3647073
    0.0748480    0.7841938    0.2541607

-->find(A<.5)  // returns (vector-)indices of all elements A(i)<0.5
 ans  =

    2.    3.    8.    9.
```

3.5.2 If-then-else

The most simple instance of this idea executes a code block only if a condition is satisfied, using also the `else` keyword enables the user to specify alternative code that is interpreted if the condition does not hold.

```
-->a=rand(), if a<.5 then, "head", else, "tail", end;
 a  =

    0.2113249
 ans  =

 head
```

3.5.3 Select case

While the *if-then-else* construct differentiates at most two cases the *select-case* statement can be used to decide between more branches depending on the value of a certain variable.

```
-->person = "tom";

-->select person;
-->case "peter", age=25; profession="student";
-->case "katrin", age=22; profession="student";
-->case "jones", age=30; profession="scientist";
-->case "olivia", age=27; profession="teacher";
-->else; age=0; profession="unknown"; "WARNING: "+person+" unknown", end
 ans  =

 WARNING: tom unknown
```

3.6 Loops

The possibility for parallel access described in section 2.5.2 makes it obsolete to use loops to assign and read matrix elements in most cases. It is strongly advised to use matrix expressions wherever possible since they are much faster. However for many algorithms loops are necessary.

3.6.1 For Loops

For loops in SciLab iterate over the elements of a vector. The number of iterations is given by the number of vector elements.
Note: In other programming languages *for loops* iterate over an integer that is incremented until a limit is reached. This is easily achieved in SciLab by creating a vector using the colon operator from section 2.1.8

```
-->n=6; res=ones(n,1);

-->for i=1:n, res(i)=sin(i*%pi/n); end

-->res'
 ans  =

    0.5    0.8660254    1.    0.8660254    0.5    1.225D-16
```

3.6.2 While Loops

The syntax of a *while loop* which iterates until a certain condition is violated, is straight forward.

```
-->i=0;

-->while rand()<.99
-->i=i+1;
-->end;

-->i
 i  =

    89.
```

3.7 Graphics

Numeric algorithms usually process and output vast amounts of data which is hard to interpret without the support of a suitable visualization.

3.7.1 Window management

The SciLab drawing system has the capability to work with multiple windows which are identified by a number or a handle. Drawing commands only influence the *current figure*. The following table lists the commands required for basic window management

scf(fig)	set current figure to fig (number or handle). Or creates a new one.
gcf()	returns the handle to the current figure. Or creates a new one.
clf(fig)	clears the windows with the id or handle fig
delete([h])	destroys the windows identified by the handles in the vector [h]

The properties of the figures can be controlled by clicking the Edit/Figure Properties menu entry in the corresponding window or by creating a figure explicitly with the figure(<PropertyName1>,PropertyValu

3.7.2 The plot command

The easiest way to get a simple figure is the command
plot([x],[y][,<LineSpec>,<GlobalPropertyName1>,GlobalPropertyValue1 ...])
where [x] and [y] contain the plot coordinates. Both [x] and [y] can be either vectors, matrices or a function defined as a macro or primitive. The following cases are possible:

[x]	[y]	
omitted	vector	[y] is plotted versus 1:length(y).
omitted	matrix	every column of [y] is plotted versus 1:size(y,1).
vector	function	plots y(x) against [x].
vector	vector	[y] is plotted against [x].
vector	matrix	every column of [y] is plotted versus [x].
matrix	function	plots y(x(:,i)) against [x(:,i)] for all i in 1,...,size(x,2).
matrix	matrix	every column of [y] is plotted versus the corresponding column in [x].

The <LineSpec> string argument is used to influence the appearance of the newly created plot. This is done by simply combining the specifiers from figure 3.3 in arbitrary order.

```
-->x=logspace(-1,0,100);

-->plot(x,sin(1. ./x))

-->plot(x,exp(x),"g")  // use green line color
```

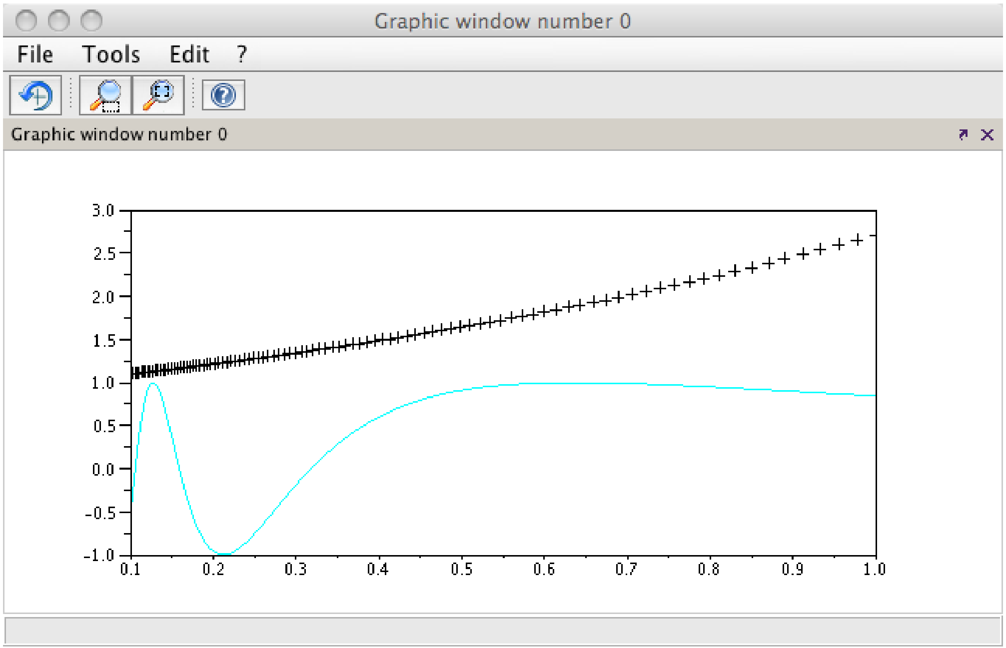

Figure 3.1: Two plots in one figure.

Similarly <GlobalProperty> influences the visual properties of the drawings. It is in fact a list of the form <PropertyName1>,Value1,...,<PropertyNameN>,ValueN. The properties accessible via <LineSpec> can be also specified. To see all possible atributes type help GlobalProperty on the command line. In case conflicting values are passed to one plot command the <GlobalProperty> has precedence. In figures containing several plots with different styles the alternative prototype:

```
plot([x1],[y1],<LineSpec1>,...,[xN],[yN],<LineSpecN>,<GlobalProperty>)
```

can be used to define the properties per-graph in the <LineSpec> strings and global attributes for all graphs via <GlobalProperty>.

```
-->clf // clears the current figure

-->plot(x,x^2,"ro",x,sin(1../x),"b+","MarkerSize",2)
```

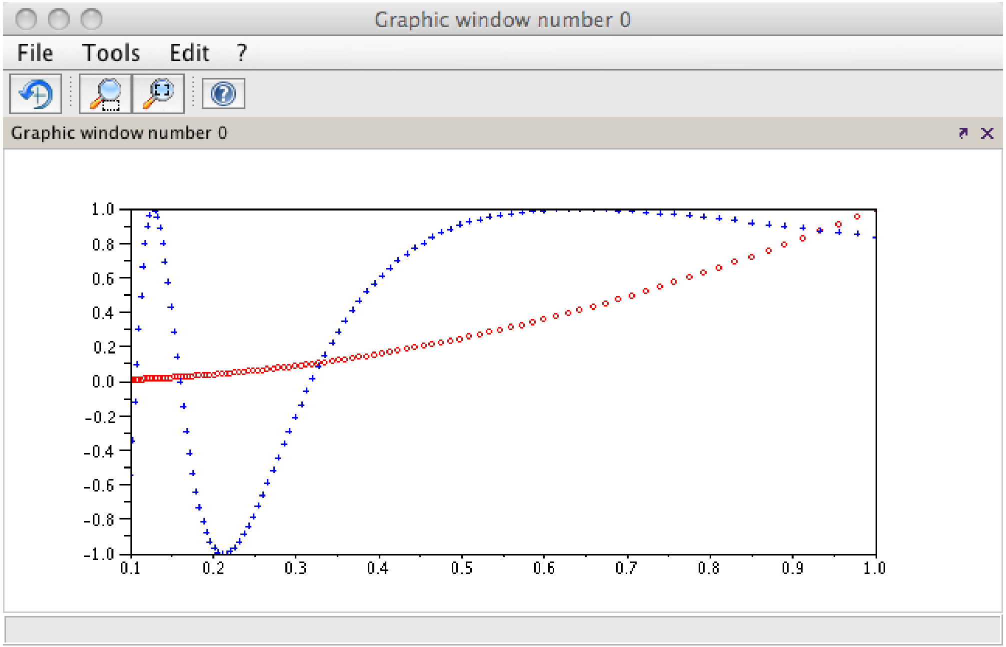

Figure 3.2: Two plots with modified marker size.

specifier	style
`-`	solid line (default)
`--`	dashed line
`:`	dotted line
`-.`	dash-dotted line

(a) line style

specifier	color
`r`	red
`g`	green
`b`	blue
`c`	cyan
`m`	red
`y`	green
`k`	black
`w`	white

(b) line color

specifier	marker symbol
`+`	plus sign
`o`	circle
`*`	asterisk
`.`	point
`x`	cross
`square` or `s`	square
`diamond` or `d`	diamond
`^`	upward-pointing triangle
`v`	downward-pointing triangle
>	right-pointing triangle
<	left-pointing triangle
`pentagram`	five-pointed star
`none`	no marker (default)

(c) marker style

Figure 3.3: Possible substrings for the `<LineSpec>` argument of the `plot` function

3.7.3 The plot2d command

The easiest way to get a simple figure is the `plot2d()` function. Given a vector b, SciLab plots automatically the vector according to the position in the vector, i.e. entry 4 in the vector is being plotted at position $(4, b_4)$. By giving a second vector, it is possible to overwrite the x-axis. We will see that in the tutorial.

3.8 Outlook

This introduction should help you to get started on some of the basic stuff of SciLab. We are going to have some programming exercises in the NAVC III lecture and we will be using SciLab as our basic tool. You can install SciLab on your computer or you can use SciLab in the programming labs, where SciLab should be installed by default. If you have problems getting SciLab ready-to-work, please feel free to contact us. This tutorial will be accompanied by an introductory session to SciLab on March 27th, in which we will show basic routines and some complete functions and applications.

3.9 References

- Several documentations
- *Introduction à Scilab* written by Bruno Pinçon,
- German translation.
- *Master Scilab & Scicos*.