Introduction to Scilab for numerical simulations

CIMPA-UNESCO-MESR-MICINN-VENEZUELA 2012

Bruno Pinçon

INRIA (Corida team) and Institut Elie Cartan
Nancy University

with a little help from Jean-Paul, Séraphin, and Youcef.

April 2012

Outline

What is scilab ?

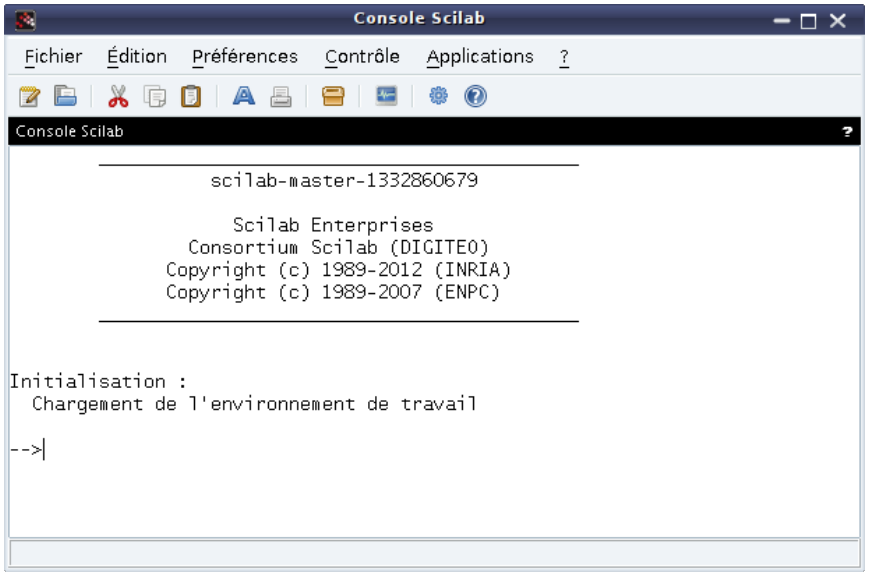
In few words, a free (CECILL) software which:

- handles vectors and matrices very easily ;
- has many mathematical/numerical functions ready to use: linear algebra (both full and sparse), solving ODE, optimization, etc...
- comes with an easy programming language ;
- has a large set of graphical features.

Scilab history:

- started in 1982 at INRIA;
- organized as a consortium since 2003 until June 2012;
- developped by a private corporation: Scilab Enterprises (but scilab should remain free software).

The scilab environment #1



enter any lines of scilab code in the command window $\Longleftrightarrow$ *write those lines in a file with the help of the SciNotes editor and then load this file in scilab.*

To load a script file in scilab:

- directly from SciNotes using an item menu or the $\triangleright$ icon;
- enter "`exec` *name_of_the_file*" in the command window.

The scilab environment #2

Scilab comes with a complete (but improvable) help:

- browse the help system from the menu ?
- or enter `help` *keyword* in the command window.

Exercise 1

1. Enter the following lines in the command window:

```
x = linspace(-%pi,%pi,11);  // mesh of [-pi,pi] using 11 pts.
y = cos(x)./(1+x.^2);       // (%pi is a scilab constant)
clf()                       // clear the (current) graphic window
plot(x,y,'b')
```

2. Now write them in a file (e.g. `exercise1.sce`) with `SciNotes` and try different methods to load it.
3. Modify your script by playing with the interval length and/or the discretisation parameter and/or the color of the plot line (`'b'`, `'g'`, `'r'`, `'k'`, `'c'`, `'m'` stand respectively for blue, green, red, black, cyan, magenta).

Various ways to define vectors and matrices I

Define a matrix from its coefficients

The basic objects in scilab are vectors and matrices.

- **scalars** are just 1x1 matrices
- **vectors** are matrices with one row or one column.

To define a matrix from its coefficients:

```
A = [1, 2, 3;  4, 5, 6]   //  the character ; introduce the next row
x = [0 ; 1; 0]  // a column vector
y = [exp(%pi), sin(%e)]  // a row vector
```

Remarks:

- row coefficients are separated by comma or blank;
- if we end such code line with a semicolon scilab will not display the object;
- coefficients can be got from an expression;

Various ways to define vectors and matrices II

- previous entered lines can be re-entered by playing with the $\uparrow$ and $\downarrow$ keyboard keys.

Define a matrix from others matrices

Same syntax works if scalars are replaced by already defined matrices or vectors:

- comma (or blank), means **right concatenation** and should involve matrices with the same number of rows
- semicolon, means **down concatenation** should involve matrices with the same number of columns.

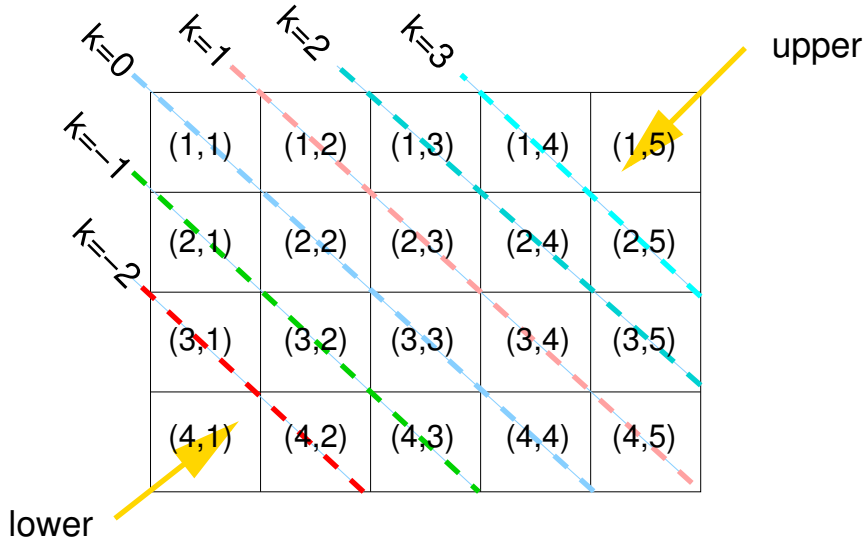
Try this:

```
B = [A ; y , 1]
C = [B, x]   // C can be built directly using [[A ; y, 1],x]
```

Various ways to define vectors and matrices III

Some matrix/vector constructors and operators

1. `linspace(a,b,n)` creates a uniform mesh of $[a, b]$ with n points
2. `zeros(m,n)` and `ones(m,n)` build $m \times n$ matrices of 0 and 1.
3. `eye(m,n)` builds the $m \times n$ identity like matrix.
4. If `x` is a **vector** `A=diag(x,k)` builds a diagonal like matrix by filling the k-th diagonal with the vector x.



Various ways to define vectors and matrices IV

5. if `A` is a **matrix** but not a vector `diag(A,k)` extracts the diagonal number `k` as a **column vector**.
6. `A*B` is the **matrix product** of the matrices A and B (or product between a scalar and a vector or matrix).
7. `A'` performs the **transposition** of matrix `A`.
8. If A is a square invertible matrix **you can solve the linear system** $Ax = b$ using `x = A\b` (a $PA = LU$ factorization of the matrix, followed by an estimation of its condition number, and finally by solving the 2 triangular systems, are done in a transparent manner).

exercise 2: write a scilab script file `exercise2.sce` which:

1. give a value to the variable n then define the $n \times n$ matrix :

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

2. Define a vector $b \in \mathbb{R}^n$ using `rand(n,1)`, compute the solution of $Ax = b$. Compute the relative residual $\|Ax - b\|/\|b\|$ using the function `norm`.
3. compute $E = \frac{1}{2}x^\top Ax - b^\top x$ Define another vector $y \in \mathbb{R}^n$ using `rand`, compute $F = \frac{1}{2}y^\top Ay - b^\top y$ and verify that $E < F$.
4. build the following matrix: $B = \left(\begin{array}{c|c} A & I_n \\ \hline I_n & A \end{array}\right)$

exercise 2: solution

```
n = 7;
v = -ones(1,n-1);
A = diag(v,-1) + 2*eye(n,n) + diag(v,1)
// another solution
//  A =  diag(v,-1) + diag(2*ones(1,n)) + diag(v,1)


b = rand(n,1);
x = A\b
res = norm(A*x-b)/norm(b)

E = 0.5*x'*A*x - b'*x
y = rand(n,1);
F = 0.5*y'*A*y - b'*y
E < F

B = [ A        , eye(n,n) ;...
      eye(n,n), A        ]
```

Assignments and extractions I

The basic assign scilab syntax takes the form:

var_name = expression

where the variable *var_name* is created (or completly redefined if it already exists)

Rmk: if you don't provide the left part, i.e. with just:

expression

the result of *expression* is assigned to a default variable named `ans`.

It is possible to change only a subpart of a given matrix. Be aware that like in usual linear algebra text math, indexing in scilab begin at 1 and not 0:

- the first element of a vector `x` is `x(1)` not `x(0)`
- nor `x[1]` i.e. parenthesis are used and not brackets !

Assignments and extractions II

Try:

```
A = rand(3,4)  // create a matrix
A(2,2) = -1    // change coef (2,2) of A
c = A(2,3)     // extract coef (2,3) of A and assign it to variable c
A(2,:)         // extract row 2 of A (it is assigned to ans)
A(2,:) = ones(1,4) // change row 2 of A
A(:;3) = 0     // change column 3 of A
B = A([1,3],[1 2]) // extract submat (1,3)x(1,2) and assign it to B
A([1,3],[1 2]) = [-10,-20;-30,-40] // change the same sub-matrix
```

The more complete assign/extraction syntaxes are:

```
A(row_ind,col_ind)= RHS     // assign
var = A(row_ind,col_ind)    // extract (and assign to var)
```

where you specify for each dimension a **vector of indices**. Useful shortcut: the colon **:** stands for the complete row or column range.

Assignments and extractions III

Another very useful vector constructor In the command window try the following expressions:

```
I = 1:5
J = 1:2:6   // try also J = 1:2:7 which give the same vector
K = 10:-1:5
II = 1:0     // this create an empty matrix
```

The syntax is `init_val:inc:lim` and this builds a row vector with `init_val` as the first coefficient, the others components being obtained from the previous one by adding it `inc` until `lim` is not overtaken. Remarks:

- if `inc` is not given it is assumed to be 1;
- when `inc` is positive and `init_val > lim` or when `inc` is negative and `init_val < lim` the resulting vector is an **empty matrix**.

This kind of vector will be useful for extraction and/or assignment of matrices and for loop control.

exercise 3

1. Copy-paste your previous script `exercise2.sce` in a new file `exercise3.sce` and use a small value for n (e.g. $n = 5$) (*Rmk: we need only the part of the code which defines A and B: you can remove unuseful lines of code*).
2. Continue the script by creating the following new matrices:
   1. C such that $C_{i,j} = A_{i,n+1-j}$ i.e. by reversing the column order of A;
   2. D such that $C_{i,j} = A_{2i-1,j}$ i.e. taking one row over two of matrix A;
   3. E the matrix formed by the B submatrix of rows and columns $n-2, n-1, n, n+1, n+2, n+3$.

exercise 3: solution

```
n = 5;
v = -ones(1,n-1);
A = diag(v,-1) + 2*eye(n,n) + diag(v,1)
B = [ A        , eye(n,n) ;...
      eye(n,n), A        ]


C = A(:,n:-1:1)

D = A(1:2:n,:)

E = B(n-2:n+3,n-2:n+3)
```

The component-wise algebra and the plot function I

3 useful operators `.*`, `./` and `.^` :

1. `x` and `y` matrices with the *same dimensions*:
   - `z=x.*y` is the component-wise product, i.e. $z_{i,j} = x_{i,j}y_{i,j}$
   - `z=x./y` is the component-wise division, i.e. $z_{i,j} = x_{i,j}/y_{i,j}$.
   - Useful shortcut: if `s` is a scalar, `z=s./y` gives $z_{i,j} = s/y_{i,j}$ **but** `z=1./y` doesn't work as expected ! (use `z=1 ./y`).
2. `x` matrix and `p` scalar:
   - `z=x.^p` is the component-wise power: $z_{i,j} = x_{i,j}^p$.
   - and `=p.^x`, gives $z_{i,j} = p^{x_{i,j}}$.

The `plot` function can be used to draw one or several curves:

```
plot(x1,y1[,style1],x2,y2[,style2], ....)
```

with `style` is an optional string to define color, line type, or symbol. strings in scilab are delimited by simple or double quote.

The component-wise algebra and the plot function II

```
x = linspace(0,2*%pi,31);
y1 = sin(x); y2 = cos(x);
scf(0); // select graphic window 0 to be the default graphic window
clf(); // clear the graphic window
plot(x,y1,"b-",x,y2,"r--");  // only lines
scf(1); // select graphic window 1 to be the default graphic window
clf(); // clear the graphic window
subplot(2,1,1); // split the graphic window and use subpart 1
plot(x,y1,"ro",x,y2,"bx");  // only symbols
subplot(2,1,2); // split the graphic window and use subpart 2
plot(x,y1,"r--o",x,y2,"g-x");  // both lines and symbols
```

colors

k	black	c	cyan
b	blue	m	magenta
r	red	y	yellow
g	green	w	white

line types

-	solid
--	dashed
:	dotted
-.	dashdot

symbols

+	+	d	◇
x	×	v	▽
o	◯	s	□
*	*	^	△

The component-wise algebra and the plot function III

playing with several graphic windows

`scf(num)`	select window `num` as current graphic window
`show_window()`	raise current graphic window
`clf()`	clear current graphic window
`xdel(num)`	delete the graphic window `num`

Graphic annotations:

- a title with `title(string_title)`.
- x and y labels with `xlabel(string_xlabel)` and `ylabel(string_ylabel)`
- a legend for the curves with `legend(curve1_leg, curve2_leg)`
- latex math notation is available by enclosing the string between $

exercise 4 I

1. In graphic window 0, draw $f(x) = \frac{1}{1+x^2}$ in blue on $[-4, 4]$ and $g(x) = cos(\pi x)e^{-|x|}$ in red on $[-5, 5]$. Hint: the absolute value function is abs. Example of use of latex for the legend:

```
legend('$\frac{1}{1+x^2}$', '$\cos(\pi x) e^{-|x|}$');
hleg = gce(); hleg.font_size=4;
```

2. In graphic window 1, draw $x^2 + y^2 = 1$ and $(x/2)^2 + (y/0.9)^2 = 1$. Add a legend. For an isometric scale play with graphic handles like this:

```
haxes = gca();  //  get identifier of current axes system
haxes.isoview = "on";  // was "off" by default
```

exercise 4: solution I

```
// curves 1 and 2 in graphic window 0

x1 = linspace(-4,4,101);
y1 = 1 ./(1+x1.^2);

x2 = linspace(-5,5,101);
y2 = cos(%pi*x2).*exp(-abs(x2));

scf(0);
clf();
plot(x1,y1,'b',x2,y2,'r');

legend('$\frac{1}{1+x^2}$', '$\cos(\pi x) e^{-|x|}$');
hleg = gce(); hleg.font_size=4;
title("some curves","FontSize",5);
```

exercise 4: solution II

```
// circle and ellipse in graphic window 1

theta = linspace(0,2*%pi,200);
x3 = cos(theta);
y3 = sin(theta);

x4 = 2*cos(theta);
y4 = 0.9*sin(theta);

scf(1);
clf();
plot(x3,y3,'b',x4,y4,'r')
legend('circle', 'ellipse')
hleg = gce(); hleg.font_size=3;
haxes = gca();
haxes.isoview = "on";
```

functions

In scilab a function definition takes the form:

```
function [y1,y2,...,yn] = function_name(x1,x2,..xm)
   // the body of the function define the output arguments y1,...,yn
   // in function of the input arguments x1,...,xm
        ..............
endfunction
```

Such a definition can be written in a script (before the part of the script which uses it) or better in another file (with a name traditionaly ending with `.sci`). You can write any number of functions in a file. In this case you have to **load** the file in scilab before we can use them.

Programming tools II

An example (write it with SciNotes in a file e.g. `quad.sci`):

```
function [x1,x2] = solve_quad(a,b,c)
   // solve a x^2 + b x + c = 0
   delta = b^2 - 4*a*c
   x1 = (-b + sqrt(delta))/(2*a);
   x2 = (-b - sqrt(delta))/(2*a);
endfunction
```

Now load the file (with one of methods seen previously) and try in the command window:

```
[r1,r2] = solve_quad(1,0,1)   // scilab use complex arithmetic as well
[r1,r2] = solve_quad(0,1,1)   // not a quadratic equation
[r1,r2] = solve_quad(0,1,1)   // not a quadratic equation
solve_quad(1,0,-1)            // ans stores the first result
                              // but second root is lost !
[r1,r2] = solve_quad(1,0,-1)  // now get the 2 roots
```

`if` tests

They permit to execute different blocks of code depending on boolean expressions:

```
if bool_expression then
    // block executed when bool_expression is TRUE
    .....
else
    // block executed when bool_expression is FALSE
    .....
end
```

Example:

```
x = rand()
if x < 0.5 then
   y = -1;
else
   y = 1;
end
```

`for` loop

One general form is:

```
for i = row_vector
   // body of the loop
   .....
end
```

- the number of iterations equal the number of components of the row vector
- at iteration k the loop variable `i` is equal to `row_vector(k)`.
- Very often the row vector is of the form `first_val:inc:lim`.

Try:

```
n = 5; fact_n = 1;
for i = 1:n, fact_n = i*fact_n, end
```

Programming tools V

It is possible to exit prematurely a `for` loop using the `break` statement:

```
for i = 1:n
    .....
    if special_condition_test then, break, end  // may be add a message
    ....
end
```

`while` loop

A `while` loop allows to repeat a block of code while a boolean expression is true:

```
while bool_expression
    // block
    ....
end
```

Programming tools VI

Try:

```
x = 1;
while x < 1000, x = 2*x, end
```

It is also possible to exit prematurely a `while` loop with the `break` statement.

Problem 1 I

Find a function $u : [0,1] \to \mathbb{R}$, solution of:

$$\begin{cases} -u''(x) + c(x)u(x) = f(x), & x \in (0,1) \\ u(0) = u_l, \quad u(1) = u_r \end{cases}$$

where u_l, u_r and the functions f and c are given. The problem can be solved approximately by a finite difference method:

- n being given, discretize the interval $[0,1]$ by a regular mesh using n intervals: $x_i = ih, i = 0, \ldots, n$ with $h = 1/n$;
- "solve" the equation only on the internal mesh points:

$$-u''(x_i) + c(x_i)u(x_i) = f(x_i), \quad i = 1, \ldots, n-1$$

Problem 1 II

- use the following development for the second derivative:

$$u''(x_i) = \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{h^2} + O(h^2)$$

(valid if u is smooth enough), so that the previous equation reads:

$$\frac{-u(x_{i-1})+2u(x_i)-u(x_{i+1})}{h^2} + O(h^2) + c(x_i)u(x_i) = f(x_i)$$
$$i = 1, \ldots, n-1$$

- finally neglect the $O(h^2)$ term: noting $U_i \simeq u(x_i)$ we get the equations:

$$\frac{-U_{i-1} + (2 + h^2 c(x_i))U_i - U_{i+1}}{h^2} = f(x_i), \quad i = 1, \ldots, n-1$$

Problem 1 III

This is a linear system $\mathbb{A}U = \mathbb{F}$ with $n-1$ equations and $n-1$ unknowns (as $U_0 = u_l$ and $U_n = u_r$ are given). Denoting $C_i = c(x_i)$ and $F_i = f(x_i)$, $\mathbb{A}$ and $\mathbb{F}$ read:

$$\mathbb{A} = \begin{bmatrix} 2+h^2C_1 & -1 & 0 & \cdots & 0 \\ -1 & \ddots & 2+h^2C_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2+h^2C_{n-2} & -1 \\ 0 & \cdots & 0 & -1 & 2+h^2C_{n-1} \end{bmatrix}$$

and:

$$\mathbb{F} = [h^2F_1 + u_l, h^2F_2, \ldots, h^2F_{n-2}, h^2F_{n-1} + u_r]^\top$$

$$U = [U_1, U_2, \ldots, U_{n-1}]^\top$$

Problem 1 IV

We will use the following datas:

$$
\begin{aligned}
&c(x) = 2\pi^4 x^2, \\
&f(x) = 6\pi^2 x(\sin(y) + y\cos(y)), \quad \text{with } y = (\pi x)^2, \\
&u_l = 0, \\
&u_r = \cos(\pi^2)
\end{aligned}
$$

which correspond to the exact solution $u(x) = x\cos(y)$ (with $y = (\pi x)^2$).

To solve the approximate problem with scilab you can organize the code in 2 files, one, e.g. `problem1.sci` (use the provided skeleton) to write the various needed scilab functions, and another, e.g. `problem1.sce`, to write the top level script.

Problem 1 V

1. Write the 3 functions corresponding to c, f and the exact solution (name them `c_func1`, `f_func1` and `u_exact1` for instance). Code them using `.*` and `.^` operators.
2. Write a function with header `function [A,F,h,x] = set_bvp(n,ul,ur,c,f)` which, from the input parameters:
   - number of intervals n,
   - the left and right boundary conditions u_l and u_r
   - and the two functions c and f.

   builds the matrix $\mathbb{A}$, the right hand side $\mathbb{F}$ and also outputs the mesh size $h = 1/n$ and the mesh points $x_i, i = 0, \ldots, n$.
   Rmk: a scilab function is also a kind of scilab variable and can be passed as an argument to another function. When you will call `set_bvp` in the top level script, you will have to put the real function names, here `c_func1` and `f_func1` corresponding to our specific problem.

Problem 1 VI

Hint: to build the main part of the right hand side $\mathbb{F}$ easily, you will need to transpose the mesh points got with `linspace`:

```
h = 1/n;
x = linspace(0,1,n+1)';
xi = x(2:n);  // internal mesh points
F = h^2*f(xi); // main part of F
```

3. Finally write the top level script, which should give a value to n, set the boundary conditions, call your `set_bvp` scilab function, solve the linear system, and plot the approximate and the exact solutions. Try several values for n (e.g. $n = 10, 20, 40, 80$).

Problem 1: solution (functions file)

```
function y = u_exact(x)
   y = x.*cos((%pi*x).^2);
endfunction
function y = c_func(x)
   y = 2*%pi^4*x.^2;
endfunction
function y = f_func(x)
   pix2 = (%pi*x).^2
   y = 6*%pi^2*x.*(sin(pix2)+pix2.*cos(pix2));
endfunction
function [A,F,h,x] = set_bvp(n,ul,ur,c,f)
   x = linspace(0,1,n+1)';
   xx = x(2:n);  // internal points
   h = 1/n;
   sd = -ones(n-2,1);
   d = 2*ones(n-1,1) + h^2*c(xx);
   A = diag(sd,-1) + diag(d) + diag(sd,1);
   F = h^2 * f(xx);
   F(1) = F(1) + ul; F(n-1) = F(n-1)+ ur;
endfunction
```

Problem 1: solution (top level script)

```
n = 20;

ul = u_exact(0);
ur = u_exact(1);

[A,F,h,x] = set_bvp(n,ul,ur,c_func,f_func);

U = A\F;   // solve linear system

UU = [ul;U;ur];
uu =  u_exact(x);

// plot
clf();
plot(xx,uu,'b',xx,UU,'r');
legend('exact sol','approx sol');
```

Problem 1 bis I

The error between the exact solution and the approximate verifies:

$$E(h) := \max_{1 \leq i \leq n-1} |U_i - u(x_i)| \leq Ch^2$$

and in most cases $E(h) \sim Ch^2$. Write another script file to compute (and recover) the convergence rate numerically. To this aim define a vector of "n" for instance:

```
n = [10, 20, 40, 80, 160, 320, 640];
```

then compute for each value n_k (so for each mesh $\mathcal{X}^{(k)}$, and $h_k = 1/n_k$):

- the vector $U^{(k)}$ (the approximate solution)
- the associated error $E(h_k) = \max_{1 \leq i \leq n_k - 1} |U_i^{(k)} - u(x_i^{(k)})|$

Problem 1 bis II

then plot the errors $E(h_k)$ as function of the mesh sizes h_k in loglog scale. When $E(h) \sim Ch^2$, for h small enough we have $E(h) \simeq Ch^2$ and: $\log(E(h)) \simeq log(C) + 2\log(h)$ so a line (with slope 2) should appear. Hints:

- when vector `n` is defined, `K = size(n,"*")` gives its number of components;
- initialize the vectors which will contain mesh sizes h_k and errors $E(h_k)$ using `zeros`.
- use a for loop
- `norm( x , "inf")` compute infinite norm
- to get a log-log plot with scilab, you have to play with the axes handle this way:

```
plot(h,E,'bo');
haxes = gca();
haxes.log_flags="ll";  // by default it is "nn"
```

Problem 1 bis: solution

```
ul = u_exact(0); ur = u_exact(1);

n = [10, 20, 40, 80, 160, 320, 640];
K = size(n,"*");
E = zeros(1,K); hh = zeros(1,K);

for k = 1:K
    [A,F,h,x] = set_bvp(n(k),ul,ur,c_func,f_func);
    U = A\F;
    uu =  u_exact(x(2:n(k)));
    E(k) = norm(U - uu,"inf");
    hh(k) = h;
end

clf()
plot(hh,E,'b-o');
haxes = gca(); haxes.log_flags="ll";
E./hh.^2
diff(log(E))./diff(log(hh))
```

Solving an ordinary differential equation I

Scilab has a powerful primitive to solve **ode**:

$$\begin{cases} u' = f(t,u) \\ u(t_0) = u_0 \end{cases}$$

where: $u(t) \in \mathbb{R}^n$, $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, and $u_0 \in \mathbb{R}^n$ is the inital condition and we look for the solution $u(t)$ on some interval $[t_0, T]$.
Solving an ode is quite easy with scilab:

1. write the function f as a scilab function;
2. select the time steps t_k you want to get the (approximate) solution vectors $u(t_k)$,
3. define the vector of initial condition
4. then call `u=ode(u0, t0, t, rhs_function)`

Solving an ordinary differential equation II

Here is a complete example to solve the Van der Pol equation:

$$y'' = c(1 - y^2)y' - y, \quad y(0) = y_0,\ y'(0) = y_1$$

which is a second order differential equation. First we reformulate it as a first order differential system, denoting $u_1(t) = y(t)$ et $u_2(t) = y'(t)$ we get:

$$\frac{d}{dt}\begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} = \begin{bmatrix} u_2(t) \\ c(1 - u_1^2(t))u_2(t) - u_1(t) \end{bmatrix}$$

First step is to write the rhs function as a scilab function:

```
function [f] = VanDerPol(t,u)
  // right hand side fct for Van der Pol equation (we use c = 0.4)
  f =  [ u(2) ;
         0.4*(1 - u(1)^2)*u(2) - u(1) ]
endfunction
```

Solving an ordinary differential equation III

Then a small script take the form (try it):

```
T = 30 ;  // we compute solution from 0 to 30
t = linspace(0,T,500); // time instants at which we want the solution
u0 = [-2.5 ; 2.5];     // initial condition vector
u = ode(u0, 0, t, VanDerPol); // ode call
clf();
plot(u(1,:),u(2,:),'b')  // plot the solution in the phase space
haxes = gca(); haxes.isoview = "on"; // set iso scale
```

The ode primitive has many options, in particular it is possible to use different integration methods and to precise the error tolerance for each time step, etc. . .

Exercise 5 I

Here, as well as the ode function, we will see the **xclick** function which allows **to get the current mouse position**. We will use it to choose any new starting point we want, by clicking a choosen position in the phase plane. Just after the click we will solve the Van Der Pol equation from the new initial condition and display the associated trajectory in the phase plane.
[c_i,c_x,c_y]=xclick(); returns the three following informations:

- the button number `c_i` which have been used: 0 for left, 1 for middle and 2 for right button;
- `c_x` and `c_y` which are the mouse coordinates in the current scale.

Your task is to understand the following script and to complete it (`exercise5.sce`).

Exercise 5 II

```
// set scale of the plot
xmin = -6; xmax = +6; ymin = -4; ymax = 4;
clf();
show_window();
haxes = gca();
haxes.data_bounds = [xmin,ymin;xmax,ymax];
haxes.isoview = "on";
title(["Van Der Pol: left click to select u0";
       "             right click to quit";
       "click should be somewhat longer enough"]);
plot(xmin,ymin);  // a plot to set tics easily

// define the instant times
T = 30;
t = linspace(0,T,500);
colors = ["k", "b", "r", "g", "c", "m", "y"];

current_color_num = 1;
while %t
   [c_i,c_x,c_y]=xclick();
```

Exercise 5 III

```
    if c_i == 0 then  // left bouton has been clicked
       u0 = [c_x;c_y];  // the new initial condition
       // draw a symbol (for instance a square)
       plot(c_x,c_y,"s"+colors(current_color_num))
       // then solve using ode
       ....   // to complete
       // then plot the new trajectory
       ....   // to complete
       // update current_color_num
       ....   // to complete

    elseif c_i == 2 then  // right mouse button has been clicked
                          // => exit the while loop
       break
    end
end
```

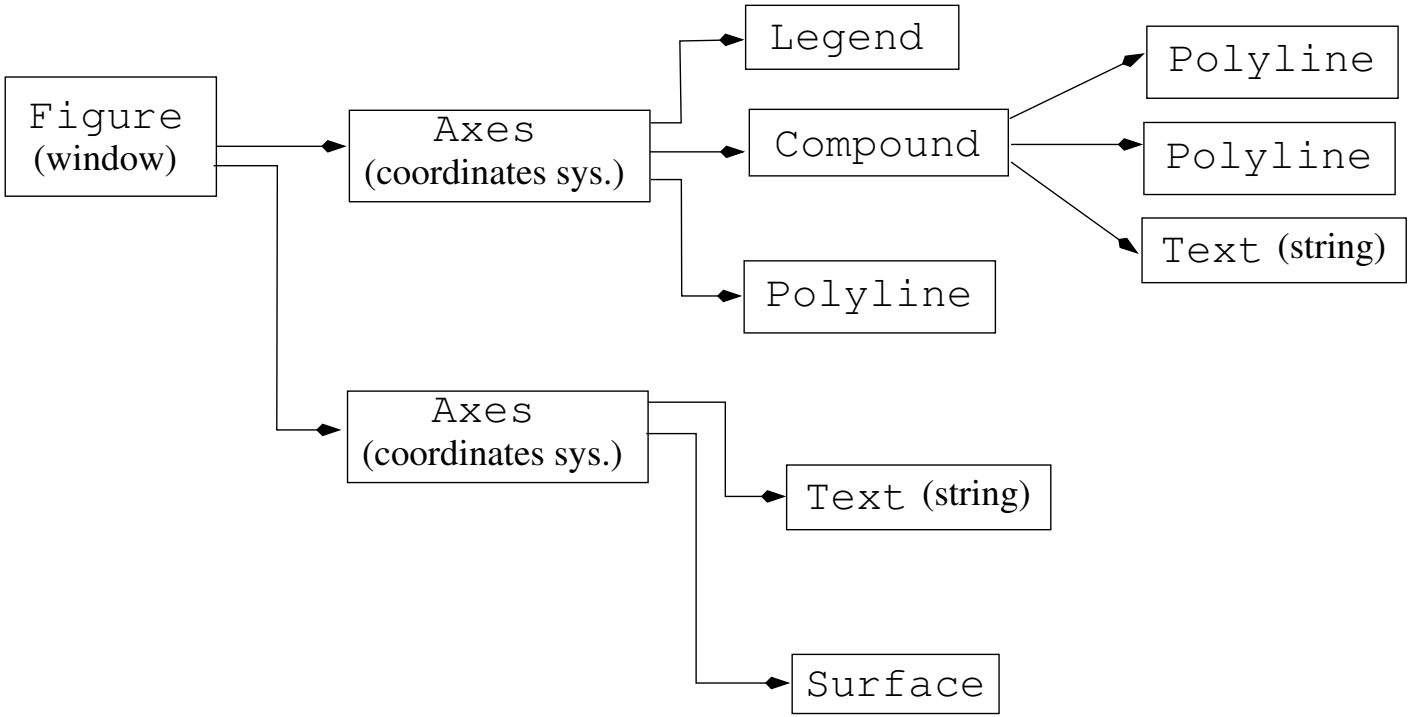
Exercise 5: solution

```
// then solve using ode
u = ode(u0, 0, t, VanDerPol);

// then plot the new trajectory
plot(u(1,:),u(2,:),colors(current_color_num));

// update current_color_num
current_color_num = modulo(current_color_num,7)+1;
```

The graphic object system



- "Figure" level is top level of a graphic hierarchy; the handle of a figure is retrieved with: `hf = gcf();`

More about graphics II

- "Axes" level: "coordinates + subpart of a graphic window"; the handle of the current "Axes" is got with: `ha = gca();`

```
ha = gca();
ha.data_bounds = [xmin,ymin;xmax,ymax]; // set the visu rectangle
ha.isoview = "on"|"off";                // set/unset isoscale
ha.log_flags="nn"|"nl"|"ln"|"ll";       // set logscale or not
```

- Last level is composed by basic graphic entities such "Polyline", "Surface", "Text", "Legend", etc... or by "Compound", which encapsulates several basic graphic entities (and other "Compounds"). For instance **plot** gathers the curves (i.e. "Polylines") in a "Compound" object. To get the handle of any basic graphic object: `he = gce()` just after the drawing function.

More about graphics III

An example:

```
clf();
x = linspace(0,2*%pi,81);
y1 = cos(x);
y2 = sin(x);
plot(x,y1,"b",x,y2,"r"); // plot cos in blue and sine in red
he = gce();    // the handle of the compound created by plot
hsin = he.children(1); // handle of the second curve (inverse order !)

// a few properties of a polyline object
hsin.thickness = 4;  // the sine in red  with a thicker line
hsin.foreground = 6; // we change color (colors indices on a colormap)
//
hcos = he.children(2); // handle onto the cos curve (inverse order !)
hcos.line_style = 2;   // we get now a dashed line
```

More about graphics IV

How make a simple animation with scilab

The **double buffering** *technique consists in "computing" first the picture to be displayed in memory (pixmap) then send it to the screen.*

In scilab it works like this:

```
f = gcf();              // get the handle of the current graphic window
f.pixmap = "on";        // set the graphic window in double buffer mode
for i=1:nb_pictures // a loop on all the pictures (the movie)
   clf()                // clear previous graphic objects
   .......
   .......              // build the i th picture in the pixmap
   .......
   show_pixmap()        // send the pixmap to the screen
end
f.pixmap = "off";       // reset the graphic window in the usual mode
```

More about graphics V

Or better like this:

```
clf();
show_window()       // raise the current graphic window
f = gcf();          // handle of the current graphic window
...........         // build first plot (in usual mode)
e1 = ......         // handle of the moving basic graphic entities
e2 = ......         // (here e1 et e2)
xclick()            // an xclick to launch the animation
                    // (put a warning using the title of the plot)
f.pixmap = "on";    // set the graphic window in double buffer
for i=2:nb_pictures
   e1.data = ..... // modify datas of graphic object e1
   e2.data = ..... // modify datas of graphic object e2
   show_pixmap()   // send to the screen
end
f.pixmap = "off";   // reset the graphic window in usual mode
```

Problem 2: the wave equation I

The 1d space wave equation with initial and boundary conditions:

$$\begin{cases} u_{tt} = cu_{xx}, & x \in (0,1), \quad t > 0, \\ u(x,0) = u_0(x), & u_t(x,0) = u_1(x), \quad x \in (0,1), \\ u(0,t) = u_l, & u(1,t) = u_r, \quad t > 0. \end{cases}$$

solved (approximatively) from $t = 0$ until $t = T > 0$ using another simple finite difference based on:

$$u_{tt}(x_i, t_j) = \frac{u(x_i, t_{j-1}) - 2u(x_i, t_j) + u(x_i, t_{j+1})}{\Delta t^2} + O(\Delta t^2)$$

$$u_{xx}(x_i, t_j) = \frac{u(x_{i-1}, t_j) - 2u(x_i, t_j) + u(x_{i+1}, t_j)}{\Delta x^2} + O(\Delta x^2)$$

We discretize:

- $[0,1]$ into n intervals, set $\Delta x = 1/n$, $x_i = i\Delta x$, $i = 0, \ldots n$;

Problem 2: the wave equation II

- $[0,T]$ into m intervals, set $\Delta t = T/m$, $t_j = j\Delta t, j = 0,\ldots,m$.

If we write the equation in (x_i, t_j) and drop the $O(\Delta t^2)$ and $O(\Delta x^2)$ we get:

$$\frac{U_{i,j-1} - 2U_{i,j} + U_{i,j+1}}{\Delta t^2} = c\frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{\Delta x^2}, \quad i = 1,\ldots,n-1,$$

where $U_{i,j}$ is thought to be an approximation of $u(x_i, t_j)$. If we denote $\alpha = c(\Delta t/\Delta x)^2$ the last expression reads:

$$U_{i,j+1} = (\alpha U_{i-1,j} + 2(1-\alpha)U_{i,j} + \alpha U_{i+1,j}) - U_{i,j-1}, \quad i = 1,\ldots,n-1,$$

So the scheme works if we know the time level j and $j-1$. At starting time we have $U_{i,0} = u_0(x_i)$, and the level 1 is defined by the approximation:

$$U_{i,1} = u_0(x_i) + \Delta t u_1(x_i), \quad i = 1,\ldots,n-1$$

Problem 2: the wave equation III

It can be proved that this explicit scheme is **second order accurate** both in time and space under the **stability condition**:

$$\Delta t \leq \frac{\Delta x}{\sqrt{c}}$$

If we denote $U^{(j)} = [U_{0,j}, U_{1,j}, \ldots, U_{n,j}]^\top$, the scheme can be written:

$$U^{(j+1)} = AU^{(j)} + BU^{(j-1)}, j = 2, \ldots, m-1$$

with matrices $(n+1) \times (n+1)$:

$$A = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \alpha & 2(1-\alpha) & \alpha & 0 & \cdots & 0 \\ 0 & \alpha & 2(1-\alpha) & \alpha & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \alpha & 2(1-\alpha) & \alpha \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}$$

Problem 2: the wave equation IV

and:

$$B = \begin{bmatrix} 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & -1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & -1 & 0 \\ 0 & \cdots & \cdots & 0 & 0 & 0 \end{bmatrix}$$

and with:

$$U^{(0)} = [u_0(0), u_0(x_1), \ldots, u_0(x_{n-1}), u_0(1)]^\top$$
$$U^{(1)} = U^{(0)} + \Delta t[u_1(0), u_1(x_1), \ldots, u_1(x_{n-1}), u_1(1))]^\top$$

For our simulation we will use the following datas: $u_l = 0$, $u_r = 0$, $u_0(x) = e^{-(12*(x-0.5))^2}$ and $u_1(x) = 0$ $(0 < x < 1)$ and a velocity of $c = 1$.

Problem 2: the wave equation V

The scilab programming task can be organized with two files:

- One file, `problem2.sci` with the two following functions:

```
function [u0,u1] = condinit(x)
   // initial conditions (this function is complete)
   u0 = exp(-(12*(x-0.5)).^2);
   u1 = zeros(x);  // build a zero matrix of same size than x
endfunction

function [A,B] = mat_wave(n,dt,c)
   // compute the two matrices A and B
   dx = 1/n;
   alpha = c*(dt/dx)^2;
   // to complete
   ..............
   // once A and B are formed:
   A = sparse(A); B = sparse(B);
endfunction
```

Problem 2: the wave equation VI

- Another file, `problem2.sce`, which will be our top level script. After some initializations, it computes and stores all the solution in the array `U` of size $(n+1) \times (m+1)$ (your task) then does the animation. Here is a nearly complete script.

```
ul = 0; ur = 0;    // boundary conditions
c = 1;             // velocity
T = 4;             // final integration time

n = 50;            // space discretization
x = linspace(0,1,n+1)';  // space mesh
dx = 1/n;          // space step size
dt = dx/sqrt(c);   // we use the max possible time step

[u0,u1] =  condinit(x);
[A,B] = mat_wave(n,dt,c);

// reserve memory
m = floor(T/dt);
```

Problem 2: the wave equation VII

```
U = zeros(n+1,m+1);

// set the two first time level
U(:,1) = u0;
U(:,2) = u0+dt*u1;

// compute other time level with the recurrence relation
for j=3:m+1
   ........  // to complete
end

// animation
f = gcf();                          // get "Figure" handle
clf();                              // clean the window
show_window();                      // raise it
drawlater();                        //
plot(x,U(:,1),"b");
title("Click to start the animation");
e = gce(); e = e.children(1); // get the handle of the curve
e.thickness = 3;                    // modify its thickness
```

Problem 2: the wave equation VIII

```
a = gca();                          // get the Axes handle
a.data_bounds = [0,-1;1,1];         // set the visualisation rectangle for al
drawnow();                          // draw the first picture
xclick();                           // wait for user click to start the anima
f.pixmap = "on";
for j=2:m
    e.data = [x,U(:,j)];            // modify data of graphic object e
    show_pixmap()                   // send pixmap to screen
end
f.pixmap = "off";
```

When it will work, change the n parameter to see that the scheme becomes more accurate as n becomes bigger (try $n = 100$ then $n = 200$, for this last version we see no visible difference between the first and last picture of the movie). You can also try a value of Δt slightly upper the stability limit to see that the numerical solution becomes quickly completly false.

Problem 2: solution

```
function [A,B] = mat_wave(n,dt,c)
   dx = 1/n;
   alpha = c*(dt/dx)^2;
   D = [1,2*(1-alpha)*ones(1,n-1),1];
   subD = [alpha*ones(1,n-1),0];
   uppD = [0,alpha*ones(1,n-1)];
   A = diag(D)+diag(subD,-1)+diag(uppD,1)
   B = -diag([0,ones(1,n-1),0])
   A = sparse(A);
   B = sparse(B);
endfunction


// compute
for j=3:m+1
   U(:,j) = A*U(:,j-1) + B*U(:,j-2);
end
```

Sparse matrices in scilab I

Howto build sparse matrices

- initialize a matrix with `spzeros` then build it by assigning coefficients or small (full or sparse) submatrices:

```
A = spzeros(4,4)              // create a sparse 6x6 matrix
A(1,1) = 3
A(2,2)= -1
A(3:4,3:4) = [2,-1;-1,2] // assign a submatrix (with a full one)
full(A)                       // convert to full matrix (for display)
```

- we can build the matrix as a full and use `A = sparse(A)...`
- a better way is to use the **sparse** function from the list of non zeros coefficients which are provided to the function by 2 arrays (plus the dimensions):

```
A = sparse(ij, val, matrix_dims)
```

with:

- `ij` a $nb\_elems \times 2$ array
- `val` $nb\_elems \times 1$ array

Sparse matrices in scilab II

each row of `ij` gives a row and column indices of a coefficient, its value being given by the corresponding row of the `val`. Here is the same example than our first:

```
ij = [1,1; 2,2; 3,3; 3,4; 4,3; 4,4];
val = [3;-1;2;-1;-1;2];
B = sparse(ij,val,[4,4])
full(B)     // for display purpose
```

The sparse function has a curious but useful feature: in case you define several same (i, j) couples of row, column indices, then the corresponding values are **added**. This is useful for building fastly matrices arising in the **finite element method**.

As it can be cumbersome to define sparse matrix here is a function to build easily some sparse matrices by filling given diagonals:

Sparse matrices in scilab III

```
function A = my_spdiags(dims,varargin)
   // usage: (each val_diags could be a scalar)
   // A = my_spdiags([m,n], num_diag1, val_diag1, num_diag2, val_diag2,
   m = dims(1); n = dims(2);
   M = length(varargin)
   if modulo(M,2) ~= 0 then, error("bad entries..."), end
   nd = M/2;   // number of provided diagonals

   ij = []; val = [];      // init ij and val arrays
   for i = 1:2:M-1         // loop onto the provided diagonals
      k = varargin(i);           // diagonal number
      values =  varargin(i+1);  // corresponding values
      if k < -(m-1) | k > (n-1) then
         error(msprintf("argument %d is not a good diag number",i+1))
      end
      if k >= 0 then
         nbelem = min(m,n-k); i = (1:nbelem)'; j = i+k;
      else
         nbelem = min(m+k,n); j = (1:nbelem)'; i = j-k;
      end
```

Sparse matrices in scilab IV

```
        nv = length(values)
        if nv==1 then
           values = values*ones(nbelem,1);
        elseif nv == nbelem then
           values = values(:)
        else
           error(msprintf("argument %d has not the good size",i+2));
        end
        ij = [ij;[i,j]]; val = [val;values];
    end
    A = sparse(ij,val,dims)
endfunction
```

For instance 1d laplacian matrix (problem 1) is got using:

```
A = my_spdiags([5,5],  -1,-1,  0,2,  1,-1)
full(A)   // for purpose display
```

Sparse matrices in scilab V

Solving sparse linear systems

The backslash operator is also available for sparse matrix, i.e. `x = A\b;` works if A is a square invertible sparse matrix. But it relies on a old sparse code outperformed by more modern ones. But two good sparse solvers are available within scilab. Here is how they work:

```
x = umfpack(A,"\",b);          // replace A\b
LUptr = umf_lufact(A);         // create an LU factorization
x = umf_lusolve(LUptr, b);     // solve Ax=b with the factorization
x = umf_lusolve(LUptr, b, A); // the same with iterative refinement
umf_ludel(LUptr);              // destroy the factorization

Cptr = taucs_chfact(A);        // create a Cholesky factorization
x = taucs_chsolve(Cptr, b);    // solve Ax=b with the factorization
x = taucs_chsolve(Cptr, b, A);// the same with iterative refinement
taucs_chdel(Cptr);             // destroy the factorization
```

Problem 3 I

To test the various scilab solvers, we are going to solve the **Poisson equation** on the unit square:

$$\left\{ \begin{array}{rcll} -\Delta u(x,y) & = & f(x,y) & (x,y) \in \Omega =]0,1[\times]0,1[\\ u(x,y) & = & 0 & (x,y) \in \partial\Omega \end{array} \right.$$

by a finite difference method always based on:

$$\frac{\partial^2 u}{\partial x^2}(x,y) = \frac{u(x-h,y) - 2u(x,y) + u(x+h,y)}{h^2} + O(h^2)$$

$$\frac{\partial^2 u}{\partial y^2}(x,y) = \frac{u(x,y-h) - 2u(x,y) + u(x,y+h)}{h^2} + O(h^2)$$

with h the space step (we use the same h both in "x" and "y"). So:

$$-\Delta u(x,y) = \frac{-u(x-h,y)-u(x,y-h)+4u(x,y)-u(x+h,y)-u(x,y+h)}{h^2} + O(h^2)$$

Problem 3 II

Given a number of intervals n:

- we define $h = 1/n$ and a grid of points: $(x_i, y_j) = (ih, jh), (i, j) \in [\![0, n]\!] \times [\![0, n]\!]$.
- and write the equation at each internal point of the grid, i.e. at (x_i, y_j) with $(i, j) \in [\![1, n-1]\!] \times [\![1, n-1]\!]$:

$$-\Delta u(x_i, y_j) = f(x_i, y_j) \iff \frac{-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1}}{h^2} + O(h^2) = f_{i,j}$$

(we denote $u_{i,j} = u(x_i, y_j)$ and $f_{i,j} = f(x_i, y_j)$).

- as usual we drop the $O(h^2)$ term, this leads to the following $(n-1)^2$ equations :

$$\frac{-U_{i-1,j} - U_{i,j-1} + 4U_{i,j} - U_{i+1,j} - U_{i,j+1}}{h^2} = f_{i,j} \quad (i, j) \in [\![1, n-1]\!] \times [\![1, n-1]\!]$$

where the quantities $U_{i,j}$ are thought to be approximations of the $u_{i,j}$.

Problem 3 III

Due to the Dirichlet boundary condition the $U_{i,j}$ are known when $i = 0$ or $j = 0$ so that we have $(n-1)^2$ unknowns and $(n-1)^2$ equations, i.e. this is a linear system of the form $\mathbb{A}\mathbb{U} = \mathbb{F}$ with the vector $\mathbb{U}$ gathering all the unknowns $U_{i,j}$, i.e. for $(i,j) \in [\![1, n-1]\!] \times [\![1, n-1]\!]$. When the function f is smooth enough one can prove that:

$$\max_{i,j} |U_{i,j} - u_{i,j}| = O(h^2).$$

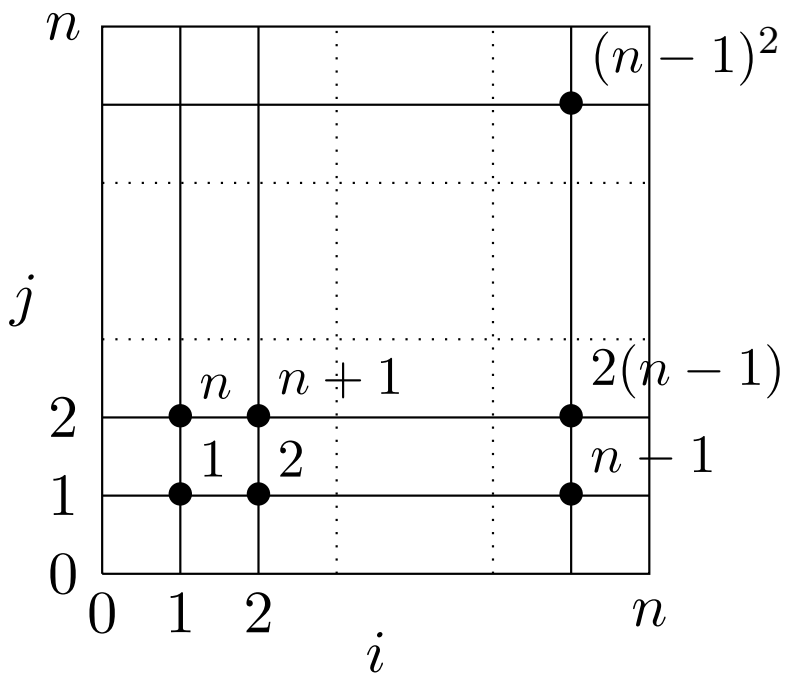
Moreover the matrix $\mathbb{A}$ is symetric positive definite.

Problem 3 IV

To obtain a precise matrix we have to choose **an ordering** for unknowns and equations, we will choose:

$$k = i + (n-1)(j-1), \quad (i,j) \in [\![1, n-1]\!] \times [\![1, n-1]\!]$$

illustrated by this figure:



Problem 3 V

With this order the matrix $\mathbb{A}$ has the following structure:

$$\mathbb{A} = \frac{1}{h^2}\left[\begin{array}{c|c|c|c|c|c} B & -I & 0 & \dots & \dots & 0 \\ \hline -I & B & -I & 0 & \dots & 0 \\ \hline 0 & -I & B & -I & \dots & 0 \\ \hline \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \hline 0 & \dots & 0 & -I & B & -I \\ \hline 0 & \dots & \dots & 0 & -I & B \end{array}\right], B = \begin{bmatrix} 4 & -1 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 4 & -1 \\ 0 & \dots & 0 & -1 & 4 \end{bmatrix}$$

with $(n-1) \times (n-1)$ blocks, each of dims $(n-1) \times (n-1)$. A closer look shows that the matrix $h^2\mathbb{A}$ is formed with 5 diagonals:

- the main diagonal filled with 4,
- the diagonals numbers $-n+1$ and $n-1$ filled with -1,

Problem 3 VI

- and the diagonals -1 and 1 being filled with the vector:

$$[\underbrace{-1,-1,\ldots,-1}_{n-2},0,\underbrace{-1,-1,\ldots,-1}_{n-2},\ldots,0,\underbrace{-1,-1,\ldots,-1}_{n-2}]$$

  i.e. the $n-1$ blocks $\underbrace{[-1,-1,\ldots,-1]}_{n-2}$ are interleaved by a 0.

This matrix will be easy to build with the `my_spdiags` function (once the previous vector built). As function f try:

$$f(x,y) = 32(x(1-x)+y(1-y))$$

In this case the exact solution is:

$$u(x,y) = 16x(1-x)y(1-y)$$

This problem is particular because the finite difference method is exact (up to floating point errors...).

Problem 3 VII

The scilab coding part could be done again using 2 files:

1. `problem3.sci` with the functions for the exact solution, the f function, the matrix $\mathbb{A}$ and a function to get the internal mesh points in the good order ; in this file your task is to complete the function which build the matrix $\mathbb{A}$.
2. a top level script (`problem3.sce`) to complete:

```
n = 20;
h = 1/n;
[X,Y] = internal_points(n);
F = h^2*test1_lap2d(X,Y);
A =  mat_laplacian2d(n);
Ue =  test1_solexact(X,Y);

// solve the linear system (try /, umfpack and taucs)
// to be completed here
...........
// compare approx and exact sols
err = norm(U-Ue,"inf")
```

Problem 3 VIII

To compare the 3 solvers you can measure the time they spent in computation, using `tic()` and `toc()` functions like this:

```
tic(); U = A\F; t_bs = toc();   // computing time for \
```

Try from $n = 20$ until $n = 300$ for instance. Possible others solvers: conjugate gradient (see `help pcg`), gmres (see `help gmres`) but **for pde's in 2 space dimensions**, **direct (modern) sparse solvers are really efficient**.

Problem3 : solution I

```
function A = mat_laplacian2d(n)
   v = -ones(1,n-2);
   w = v;
   for k = 1:n-2, w = [w,0,v]; end
   m = (n-1)^2;
   A = my_spdiags([m,m], 0,4, -1,w, 1,w, n-1,-1, -n+1,-1);
endfunction

// solve the linear system (try /, umfpack and taucs)
tic(); U1 = A\F; t_bs = toc()
tic(); U2 = umfpack(A,"\",F); t_um = toc()
tic(); Cptr=taucs_chfact(A); U3 = taucs_chsolve(Cptr,F,A);
taucs_chdel(Cptr); t_tcs = toc()

// compare approx and exact sols
err1 = norm(U1-Ue,"inf")
err2 = norm(U2-Ue,"inf")
err3 = norm(U3-Ue,"inf")
```

Other graphics (contour2d, plot3d, etc. . .) I

Contour lines

To draw contour lines of a function defined on a rectangle, one could use the `contour2d` function whom basic usage is:

```
contour2d(x,y,Z,selected_levels)
```

- `x` and `y`, 2 vectors ($x_1 < x_2 < \ldots x_{nx}$ and $(y_1 < y_2 < \ldots y_{ny})$ defining a 2d grid where the "z" values are available ;
- `Z` should be a matrix of size $nx \times ny$ with $Z_{i,j}$ the function value at (x_i, y_j).
- `selected_levels` could be either an integer scalar or a vector of given levels values.

Other graphics (contour2d, plot3d, etc. . .) II

Try:

```
// may be you have to put this 2 lines:
// usecanvas(%f);
// system_setproperty('jogl.gljpanel.nohw','');
x = linspace(0,2*%pi,60);
Z = cos(x')*cos(x);
clf();
contour2d(x,x,Z,10);  // plot 10 levels
```

On this example we can see the default colormap of scilab: contour2d use the colors number 1, 2, . . . , 10 to draw the 10 levels. It is possible to change the default colormap at the "Figure" level. Try this:

```
f = cgf();  // the handle of the current figure
f.color_map = jetcolormap(10);  // jet colormap with 10 colors
```

Other graphics (contour2d, plot3d, etc. . .) III

There are several such colormaps in scilab, see `help colormap`. Now the same example but with selecting the precise levels to draw:

```
zlevels = -0.8:0.2:0.8; // the selected levels
clf();
contour2d(x,x,Z,zlevels);
a = gca(); a.isoview = "on";  //  iso scale
```

Sometime the numbers printed on the figure to give level values are not too clearly readable. A possibility is to put them in a legend, see the last example of the `contour2d` help page.

Other graphics (contour2d, plot3d, etc. . .) IV

Plotting in 3d

One possibility for 3d plots is to use the `plot3d` function. Basic usage is `plot3d(x,y,Z)` with argument `x`, `y` and `Z` like in `contour2d`, try:

```
clf();
plot3d(x,x,Z)
// plot3d1 colors the faces using level z
clf();
plot3d1(x,x,Z)   // not too much colors
f = gcf(); f.color_map = jetcolormap(64);  // use more colors
a = gca(); a.isoview="on";   // use iso scale
f.color_map = hotcolormap(64); // try another colormap
// add a title
title("$z = cos(x)cos(y)$","FontSize",5)
```

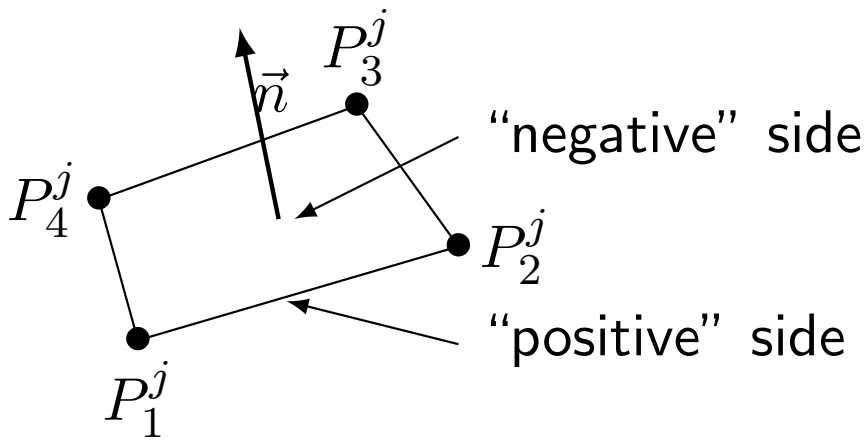
Other graphics (contour2d, plot3d, etc. . .) V

In fact `plot3d` can plot any surface which have been discretized by a collection of facets:

```
plot3d(xf,yf,zf)
```

with `xf`, `yf`, and `zf`, three arrays of same size *nb_vertices* $\times$ *nb_facets*. The facet j is described by the j columns of these 3 arrays: it is constituted by the *nb_vertices* points with coordinates:

$$P_i^j = (xf_{i,j}, yf_{i,j}, zf_{i,j}), \quad i = 1, \ldots, nb\_vertices$$



Other graphics (contour2d, plot3d, etc. . .) VI

Note that the "positive" side is not the usual one and this has a consequence when attributing a color to a facet: only the "positive" side is filled with the given color (all negative sides have a same color). Here is a complete example for plotting the unit tetrahedron:

$$P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \; P_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \; P_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \; P_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

```
//     f1 f2 f3 f4
xf = [ 0  1  0  0;
       1  0  0  0;
       0  0  0  1];
yf = [ 0  0  0  0;
       0  0  1  0;
       1  1  0  0];
zf = [ 0  0  0  0;
```

Other graphics (contour2d, plot3d, etc. . .) VII

```
        0  1  0  1;
        0  0  1  0];
clf()
plot3d(xf,yf,zf);
a = gca(); a.isoview="on";
show_window();
```

To attribute a color for each facet you have to add the color information like this:

```
colors = [2, 3, 4, 5];
clf()
plot3d(xf,yf,list(zf,colors));
a = gca(); a.isoview="on";
show_window();
```

Other graphics (contour2d, plot3d, etc. . .) VIII

It is possible to have a shaded rendering by attributing a color to each facet vertex. Here is an example:

```
clf();
f = gcf();
f.color_map= jetcolormap(128);
colors = [ 1, 43,  1,  1;
           43,128, 85,128;
           85, 85,128, 43];
plot3d(xf,yf,list(zf,colors));
a = gca(); a.isoview="on";
title("a shaded tetrahedron","FontSize",4)
show_window();
```

How to display usual parametric surfaces I

To display parametric surfaces:

$$x = x(u, v); y = y(u, v); z = z(u, v); (u, v) \in [a, b] \times [c, d]$$

like for instance the Moebius strip:

$$\begin{aligned} x &= (R + \rho \sin(\theta/2)) \cos(\theta) \\ y &= (R + \rho \sin(\theta/2)) \sin(\theta) \\ z &= \rho \cos(\theta/2) \end{aligned}$$

with `plot3d` without computing the facets by ourself (it is cumbersome) here is a possible method:

How to display usual parametric surfaces II

1. write the corresponding parametrization using component-wise operators:

```
function [x,y,z] = moebius(theta, rho)
   R = 1;
   x = (R + rho.*sin(theta/2)).*cos(theta);
   y = (R + rho.*sin(theta/2)).*sin(theta);
   z = rho.*cos(theta/2);
endfunction
```

2. discretize the parameter rectangular domain:

```
m = 200;
n = 80;
u = linspace(0,2*%pi,m);
v = linspace(-0.4,0.4,80);
```

3. build the grid points (u_i, v_j) using `ndgrid` and then call `nf3d` which computes the facets:

```
[U,V] = ndgrid(u,v);
[X,Y,Z] =  moebius(U,V);
```

How to display usual parametric surfaces III

4. then call `plot3d`:

```
clf();
plot3d(xf,yf,zf);
a = gca(); a.isoview = "on";
title("Moebius strip", "FontSize", 5)
show_window();
```