

Scilab for Real Dummies, Introducing an Open-Source Alternative to Matlab

v1.0 / Scilab 5.3.2 (5.3.3)

Johnny Heikell



"It is a mistake often made in this country to measure things by the amount of money they cost." Albert Einstein

About this presentation

I compiled this presentation while familiarizing myself with Scilab for basic engineering applications. The exercise taught that a reason to the limited spread of Scilab is lack of good tutorials in English, which makes learning Scilab a frustrating experience for programming newbies. It's a pity because Scilab deserves better recognition. Hopefully this presentation can be of use to at least some Scilab aspirants.

The text no doubt has shortcomings and errors. I hope to come up with an improved version in a not too distant future (with Scilab 6). Please post comments & suggestions at: <http://scilabdummies.wordpress.com/>

Espoo in August 2011

Johnny Heikell

www.heikell.fi

LinkedIn

Copyleft: This material is released under the only
condition that you do not put restrictions or a price
tag on your redistributions—modified or not—and
add this requirement to child copies.

Otherwise © J. Heikell 2011

Tribute to old gods



The best Scilab tutorials are non-English. The following are the ones that I have consulted most for this work:

- Timo Mäkelä's Scilab/Xcos tutorials (3 parts) in Finnish <<http://sites.google.com/site/tjmakela/home>>. Heavy on mathematical formalism, standard dull LaTeX typesetting, but **the best one I know**
- Jean-Marie Zogg's *Arbeiten mit Scilab und Scicos* in German <http://www.fh-hwtchur.ch/uploads/media/Arbeiten_mit_Scilab_und_Scicos_v1_01.pdf>. It's **good** and informal, and contains details that Mäkelä has omitted. Needs updating
- Wolfgang Kubitzki's mixed tutorials in German that can be found at <<http://www.mst.fh-kl.de/~kubitzki/>>. **Quite good**, a lot of details, few practical examples (scripts in separate .zip files)

I am indebt to their work.

“To copy from one is plagiarism, to copy from many is research.” Unknown

Why I did it the way I did it



As a grad student at KU in 1990-91, I needed to quickly learn MathCAD or Matlab. A fellow student showed me MathCAD basics in 15 minutes with the use of a sine function. The lecture went something like this:

- "*First you declare the variables that you need*"
- "*Then you define the function that you want to plot*"
- "*After that you write the plot commands*"

With that teaching I got started and was able to use MathCAD for my MS thesis.

Lessons learned: Show examples and skip the academic trivia.

I am deeply grateful to Jim for his lesson. We'll repeat it as soon as Scilab is installed and opened.

Why PowerPoint?



Why do I release this tutorial as a PowerPoint* presentation when there is enough material for a 400-page book? There are several reasons:

1. These were originally personal notes, I recognized only later that they may be of use to others
2. It is easy to edit PPT material slide by slide
3. You get a quick overview of the discussion at hand by shifting between PPT's Normal and Slide Sorter views
4. PPT has an advantage over PDF in allowing the reader to modify the work the way (s)he likes
5. You can copy-paste the provided scripts into Scilab's Editor without having to rewrite them, only minor editing is needed
6. And finally, I have seen **too many depressing LaTeX documents**

*) .ppt documents do not require MS software. LibreOffice works as well (at least up to PPT 2003) but some editing may be needed. Oracle threw in the towel on OpenOffice in April 2011, but it lives on in the Apache Incubator.

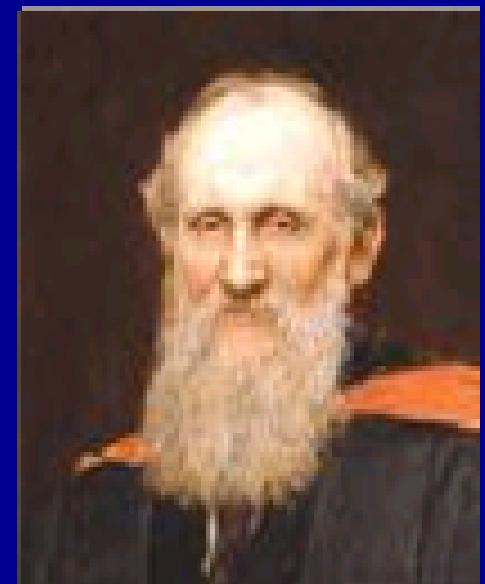
Why simulate?

- British physicist and engineer Lord Kelvin (William Thomson) is known to have said:

"When you can measure what you are speaking about and express it in numbers, you know something about it."

- His words can be paraphrased in computer-age terms:

"When you can simulate what you are speaking about and present it visually, you know something about it."



Lord Kelvin 1827-1904

Contents

- | | |
|---|---|
| 1. <u>Introduction</u> | 11. <u>Flow control</u> |
| 2. <u>A first peek at Scilab</u> | 12. <u>Examples, Set 4</u> |
| 3. <u>The Console & Editor</u> | 13. <u>Doing math on Scilab</u> |
| 4. <u>Examples, Set 1</u> | 14. <u>Examples, Set 5</u> |
| 5. <u>Matrices, functions & operators</u> | 15. <u>Working with GUIs</u> |
| 6. <u>Examples, Set 2</u> | 16. <u>File handling</u> |
| 7. <u>Graphics & plotting</u> | 17. <u>Animation</u> |
| 8. <u>Examples, Set 3</u> | 18. <u>Miscellaneous</u> |
| 9. <u>Converting Matlab files</u> | 19. <u>Examples, Set 6</u> |
| 10. <u>Subroutines</u> | 20. <u>Adieu</u> |

HOW TO HYPERLINK IN POWERPOINT: 1) **Slide Show mode:** By clicking on the underlined text. 2) **Normal View mode:** Put cursor on the underlined text → right-click → Click: Open Hyperlink. (There is a bug in PowerPoint, hyperlinking to certain slides is impossible, e.g. to Chapter 19.)

1. Introduction

What is and why use Scilab?



[Return to Contents](#)

What Scilab is (1/2)



- A software package for scientific and engineering computing, quite similar to Matlab
- Scilab is a tool for **numeric computing**, as are Excel, GNU Octave, Matlab, etc. The alternative is **symbolic computing**, to which belong Maple, MathCad, Mathematica, and others
- Developed by Consortium Scilab (DIGITEO), behind which are a number of French institutions and companies
- Included in the Scilab package is Xcos, a graphic modeling and simulation tool. However, it is not compatible with Simulink. Xcos 1.0 came with Scilab 5.2, before there was Scicos. The **confusion** is complete with a rival called Scicoslab
- Scilab is free and can be downloaded at www.scilab.org

What Scilab is (2/2)

- Scilab is matrix-oriented, just like Matlab
- It allows matrix manipulations, 2D/3D plotting, animation, etc.
- It is an open programming environment that allows users to create their own functions and libraries
- Its editor has a built-in, though elementary, debugger
- Main components of Scilab are:
 - An interpreter
 - Libraries of functions (procedures, macros)
 - Interfaces for Fortran, Tcl/Tk, C, C++, Java, Modelica, and LabVIEW—but not for Python and/or Ruby
- Which is “better,” Matlab or Scilab?
 - Matlab outperforms Scilab in many respects, but Scilab is catching up. The use of Matlab is motivated only in special circumstances due to its high cost

Why use Scilab—personal reasons

- Matlab 6.5 (R13) was not compatible with my new Windows Vista laptop. MatWorks, Inc., recommended to buy a new version
- I refused to pay another license fee for Matlab and went looking for open-source alternatives:
 - Sage felt bulky, immature, and focused on pure mathematics
 - Python is not optimized for scientific and engineering tasks
 - Python(x,y) messed up my PC when I installed it. Maybe I should have tried SciPy instead?
 - I grew tired of GNU Octave before I figured out how to download and install it (I want a tool to use, not to fight against)
 - Scilab was the fifth alternative that I looked at. It gave no immediate problems, so I stuck to it. Later I have come across bugs and crashes/lockups—and become frustrated with its poor documentation

Would I still select Scilab? Yes, I am impressed by Scilab and believe that the competitors cause you gray hair as well—one way or another.

Why people don't use Scilab

The following are some comments about Scilab and open-source software in general that I have come across:

- “Scilab? Never heard of it”
- “Octave is closer to Matlab”
- “As a company we have to use software that will be supported ten years from now”
- “It doesn’t have the toolboxes that we need”
- “There is a cost involved in shifting to a new software tool, even if the tool is gratis”
- “Training and documentation support is poor”
- “There are no interfaces for other software tools that we use”
- “It seems to be rather slow”

Conclusion: Scilab, like other open-source programs, lacks credibility in the eyes of users—particularly professional users. The situation is similar with various excellent Linux distros and the LibreOffice office package. Users trust products that have to be paid for

Scilab advantages



- Numeric computing is better suited for complex tasks than symbolic computing
- Not all mathematical problems have closed form solutions, numeric computing will therefore always be needed
- Scilab is similar to Matlab and keeps developing even closer. It is quite easy to step from one to the other
- Scilab requires less disk space than Matlab and GNU Octave
- It includes a **Matlab-to-Scilab translator** (.m files to .sci files)
- Data plotting is said to be simpler than with GNU Octave (but the trend is toward more complex handle structures)
- The Xcos toolbox installs automatically with Scilab, be it, that Xcos is not compatible with Simulink
- Scilab installs without immediate problems on Windows computers
- **Scilab is free**—if your wasted time and frustrations are worth nothing. The fight for a limited number of expensive licenses (Matlab, Mathematica, etc.) is not an issue in professional life

Scilab disadvantages



- Numeric computing introduces rounding errors, contrary to symbolic computing
- The learning effort required by numeric computing is higher than for symbolic computing
- Scilab **lacks a unified tutorial and/or user's manual**. You “try and cry” and waste time searching for information on its use*
- In some cases Scilab executes much slower than Matlab and GNU Octave (improvements are said to be under way)
- Scilab’s tools for creating GUIs are poor compared with Matlab
- The Help Browser is very formal and of little use to newbies
- Scilab has **bugs** and tends to **crash/lockup** (it happens to Bill Gates as well. Often)
- On-line support from Equalis costs \$495 or more per annum (the French prefer \$ to €)

*) Scilab is not alone. The open-source community has a poor track record in documentation because “paperwork” does not bring recognition.

Terminology: “function”

The C programming language brought confusion with its unrestricted use of the term “function” and this is repeated in Scilab. The term refers to (at least):

- Mathematical functions in general
- Scilab’s built-in functions
- User defined functions (UDF)

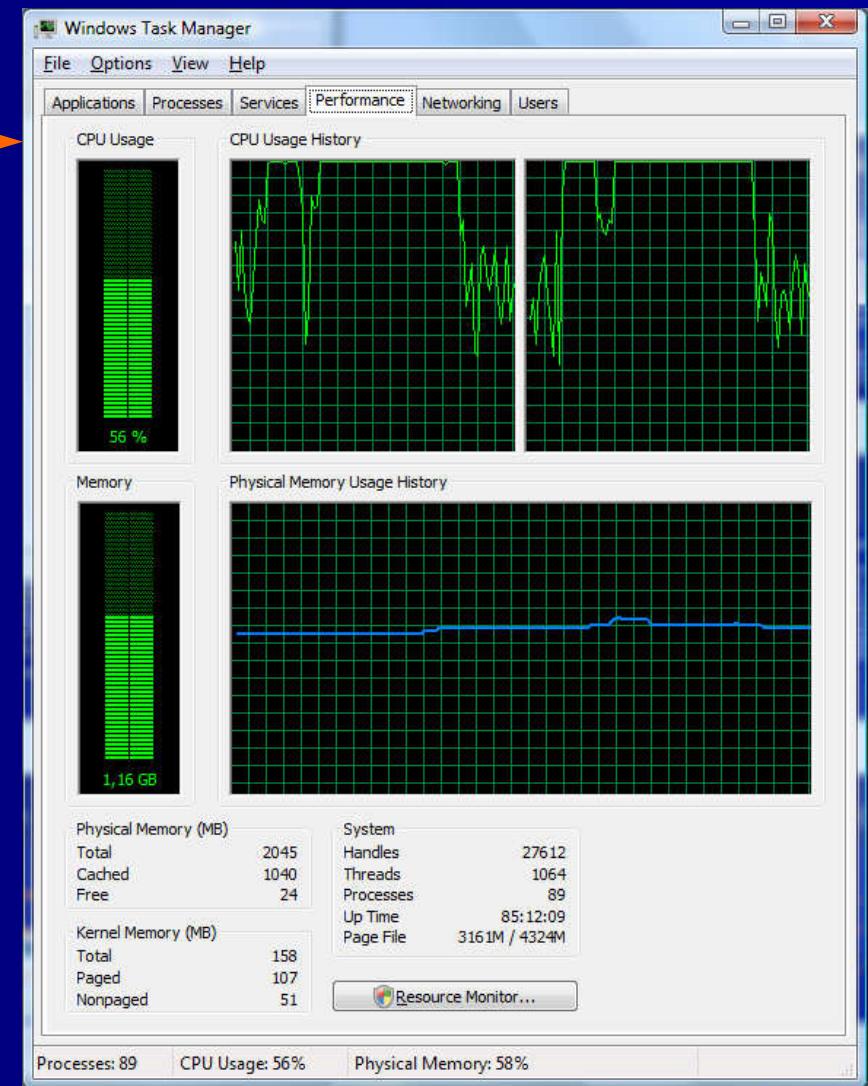


I would prefer the terms *function*, *macro* (or *procedure*), and *subroutine* respectively (protests from dogmatic programmers are overruled). Sometimes I talk about subroutine, but it is not always possible. For instance, function is the term that must be used to define a UDF in Scilab. And there is also the risk of adding to the bewilderment by applying own terminology. The confusion remains...

Intro to problems (1/3): crashes & lockups

Processor loads of this magnitude are normal during computer startup. However, this is the situation after Scilab had crashed and I had closed it. “WScilex.exe” had another of its lockups and required to be closed with the Task Manager (or by rebooting the computer).

The Scilab team’s standard answer to problems like this is to make sure that the computer’s drivers are up-to-date. It has not worked for my Windows Vista PC.



Intro to problems (2/3): new releases*

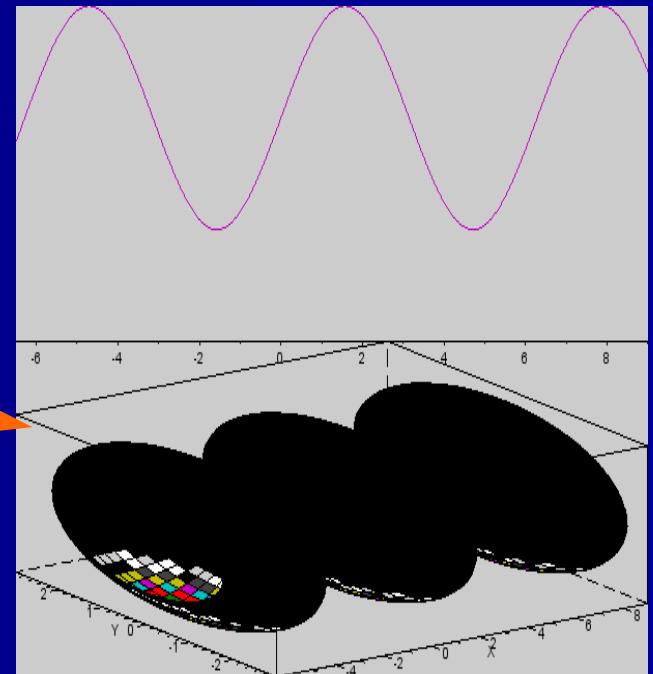


- With Scilab 5.2 came a problem that I did not experience with version 5.1.1: Copy-pasting from Scilab's Editor to PowerPoint frequently caused the latter to **crash**. The bug has been fixed
- With Scilab 5.3.0 I found that the paths File/Open file in... and File/Save file in... on the Editor were unresponsive
- Some scripts that I originally wrote using Scilab 5.1.1 did not work with Scilab 5.3.0, and GUIs on 5.3.2 are a real pain down there
- Typically larger updates come with bugs and are quickly followed by minor "bug fix" updates (a.k.a. patches). Scilab 5.3.1 emerged within three months of 5.3.0. This is universal in the software business
- It is wise to **keep an old Scilab version** until you know that the new release can be trusted (I was happy I had kept version 5.1.1 when GUIs on 5.3.1 & 5.3.2 gave me problems)

*) Various Scilab versions are mentioned. I have worked with Scilab 5.1.1 - 5.3.2. Scilab 5.3.3 came too late to be considered.

Intro to problems (3/3): ATOMS and nerds

- ATOMS is Scilab's system for downloading and installing user-developed toolboxes. It has given me real gray hair
- I installed two toolboxes and Scilab plots became a mess. Here you can see what the later discussed rotation surface looked like with toolboxes installed
- I found what caused it after reinstalling Windows and the toolboxes. It took me days to get all programs running
- The idea of user contributions is basically sound, but there is a **risk with nerds** that have more zeal than ability and tenacity to properly test their programs



Embedded information

- Scilab comes with some built-in information structures. The major ones are:
 - The **Help Browser** that can be accessed from various windows. Its utility improved with Scilab 5.3.1 when demonstrations were included, but the Help Browser is still a hard nut for newbies. It confuses by sometimes referring to obsolete functions
 - **Demonstrations** that can be accessed from the Console. Not really tutorials and some of them act funny, some may cause Scilab to crash, and others still ask for a C compiler
 - **Error messages** displayed on the Console. Quite basic messages, sometimes confusing, sometimes too brief
- What is really missing is an **embedded tutorial** (or even a user's manual of the Matlab style) that is updated with each Scilab release

Information on the Web (1/2)

- The main portal is **Wiki Scilab**, <<http://wiki.scilab.org/Tutorials>>, where most of the accessible tutorials are listed
- Scilab's **forge** <<http://forge.scilab.org/>> is a repository of "work in progress," many of which exist only in name. Its set of draft documents is valuable
- Wiki Scilab's **HowTo** page <<http://wiki.scilab.org/howto>> has some articles of interest
- Free sites:
 - **Scilab File Exchange** website <<http://fileexchange.scilab.org/>>. A new discussion forum managed by the Scilab team and "dedicated to easily exchange files, script, data, experiences, etc."
 - **Google discussion group** at <<http://groups.google.com/group/comp.soft-sys.math.scilab/topics>>
 - **MathKB** <<http://www.mathkb.com/>>. Contains, among other things, a Scilab discussion forum. Mostly advanced questions
 - **spoken-tutorial** <http://spoken-tutorial.org/Study_Plans_Scilab/>. Screencasts under construction by IIT Bombay. Scilab basics

Information on the Web (2/2)

- YouTube has some video clips on Scilab, but nothing really valuable
 - Equalis <<http://www.equalis.com>>. By registering you gain free access to the discussion forum
 - <<http://usingscilab.blogspot.com/>> used to be a very good blog but is now terminally ill. Worth checking the material that is still there
 - Scilab India <<http://scilab.in/>> is basically a mirror of Scilab Wiki, with added obsolete material and a less active discussion forum
- If you know German:
- German technical colleges produce helpful basic tutorials on Scilab (better than their French counterparts). Search the Internet e.g. using the terms "Scilab" + "Einführung" and limit the language option to German

Conclusion: A lot of resources have gone into producing the existing scattered documentation, but they have been uncoordinated and have produced little relative the input effort. Lousy management!

Books



There is **not** a single good textbook in English on Scilab like you find in abundance on Matlab. These are the books that I am familiar with:

- Beater, P.: *Regelungstechnik und Simulationstechnik mit Scilab und Modelica*, Books on Demand GmbH, 2010. Basic control systems for mechanical engineers. Scilab plays only a minor role in the book
- Das, V.V.: *Programming in Scilab 4.1*, New Age International, 2008. Reference manual with uninviting layout, obsolete functions, and no practical examples. Useless
- Chancelier, J.-P. et al.: *Introduction à Scilab, Deuxième édition*, Springer, 2007. An intermediate/advanced textbook with some engineering applications. Approaching obsolescence
- Campbell, S.L. et al: *Modeling and Simulation in Scilab/Scicos*, Springer, 2006. Based on Scilab 3.1, over half of the book is on Scicos. Of some use, but dull the way Springer and LaTeX make them
- Gomez, C. et al.: *Engineering and Scientific Computing with Scilab*, Birkhäuser, 1999. Often referred to but outdated and of no use

On updates & literature

Scilab evolves rapidly and one frequently encounters obsolete features. Functions are often declared obsolete, although Scilab still may support them, and other functions are removed altogether. There is obviously no collection of obsolete/removed functions and their current equivalents (if any).

The Scilab team is slow with information on major updates. For instance, the GUI interface is said to have been completely renewed with version 5.x, but so far the only (poor) GUI description that I have seen is for version 4.x. It's almost three years now...

Rapid development is a reason to why the limited literature on Scilab is mostly obsolete, sometimes outright misleading. I got a hands-on experience with all the changes that had to be made to 5.1.1 scripts before they agreed to run on version 5.3.x (and not all do)



Scilab learning obstacles

Learning Scilab can be frustrating to a person with limited previous programming experience. The biggest hurdles are:

- **Lack of hands-on tutorials** for English-speaking newbies. The situation is better, though not good, with some other languages
- **Excessive number of Scilab functions.** There are some two thousand of them. There are often numerous options to select between; some of which work, some don't, some of which you know, most you don't
- **Unhelpful Help Browser.** Even when you have a hunch of which function to use, you cannot get it right because of the Help Browser's cryptic explanation
- **Basic programming errors.** Creating infinite loops, dividing by zero, using * instead of .* , etc. We all make them, there is no way around them than by practicing. "Übung macht den Meister!"



On the bright side...



- Scilab works! Despite my complaints it mainly does a fine job
- It is a great thing that it is given away for free to all of us who cannot afford expensive commercial simulation tools
- It is a great thing that it is give away for free to all commercial and non-commercial institutions that care about cost-effectiveness
- It is a free gift (though with restrictions*) to science and engineering and deserves support of us who happily download whatever comes gratis on the Web
- It deserves support because Scilab, like other open-source IT solutions, faces an uphill struggle against vast commercial interests and skeptical individuals
- Long live the free and open-source/access community!

*) Scilab is released under the French CeCILL license. The question is, is it really a Free and Open-Source license that allows you to release a Scilab copy under a new name, the way OpenOffice was turned into LibreOffice?

2. A first peek at Scilab

What you face when trying to get started—including “Scilab in 15 minutes”



[Return to Contents](#)

Windows installation (1/3)

1. Download Scilab from
www.scilab.org
(Windows on the top,
other OSs below)

2. The right operating system
should be on top. Save the file,
typically it goes to your own
Downloads folder

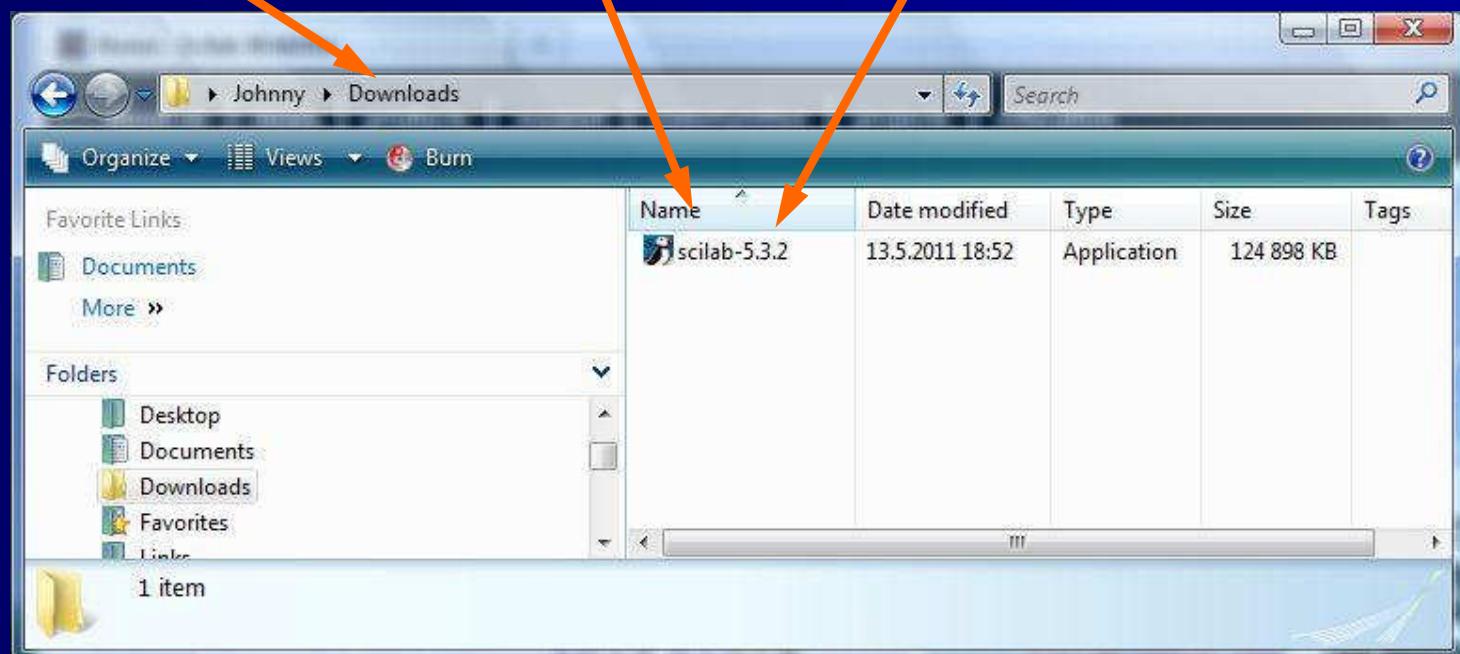


Windows installation (2/3)

3. Scan the downloaded file for viruses

Inside the Downloads file

4. Double-click on the file to install Scilab, follow the prompts



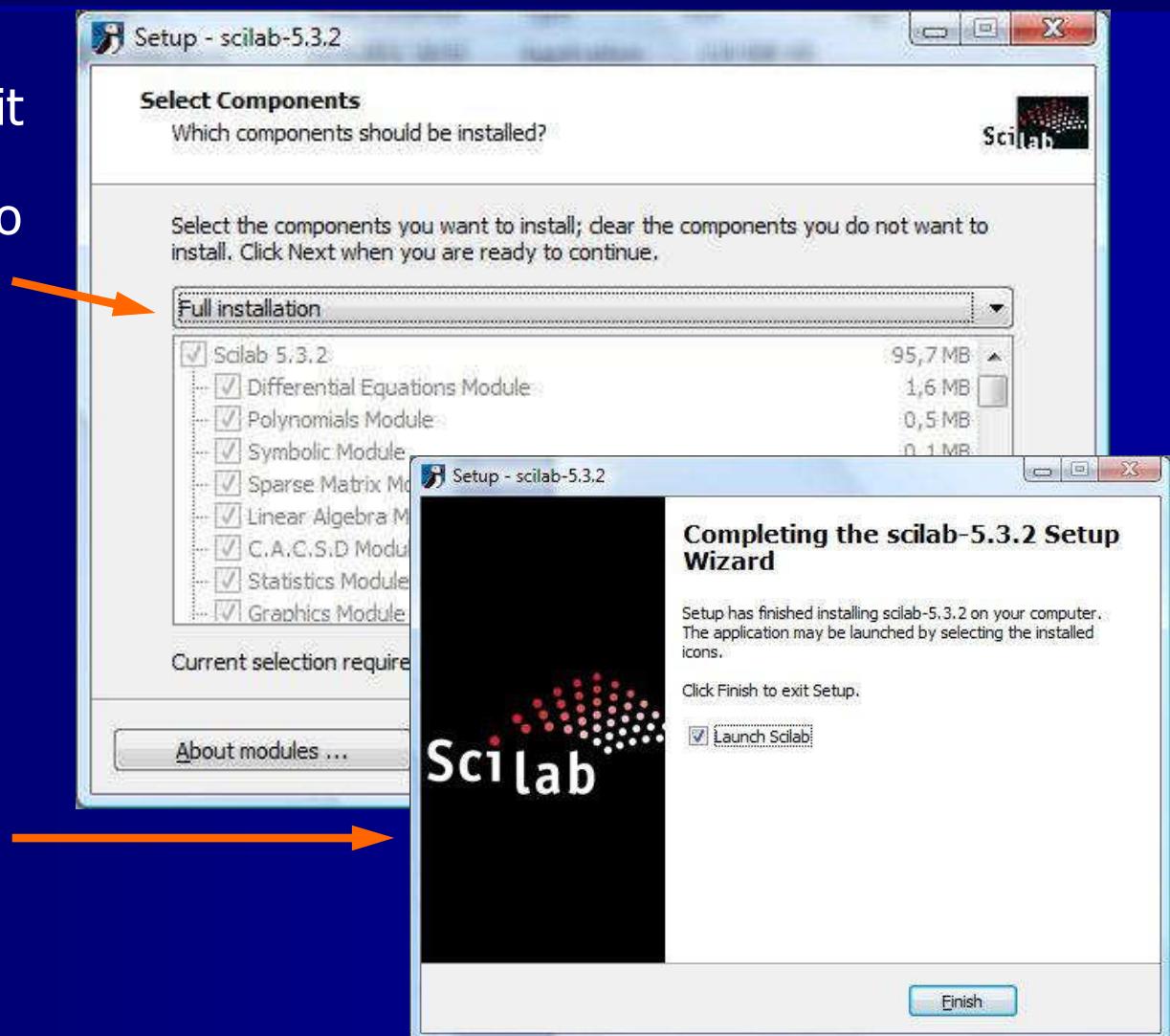
Windows installation (3/3)

5. Scilab suggests that it should install all toolboxes (modules). Go for it unless you are really short of memory

6. Accept Scilab license terms (you have no option so why do they ask?). Click Next as many times as needed

7. You're all set to use Scilab (no need to reboot the computer)

Note: Scilab does not uninstall an old version

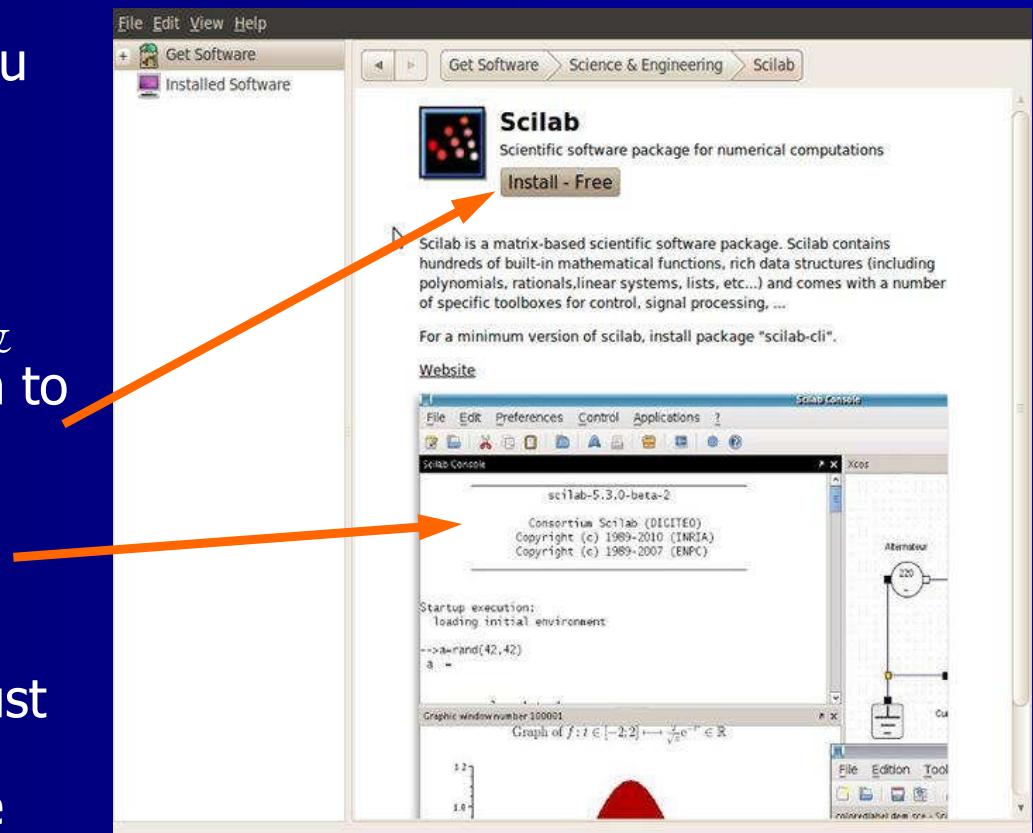


Linux installation



This discussion is valid for Ubuntu 10.04 LTS with the GNOME desktop*

- Click: Applications/Ubuntu Software Center/Science & Engineering and scroll down to Scilab; then just Click Install
- Only Scilab 5.3.0 beta-2 is available at the repository
- For the latest version you must go to Scilab's web site and download Linux binaries. The installation, however, is a **trickier question** and I do not cover it here (have not tried it)



*) Ubuntu 11.04 with Unity has been released, but I have not gone for it

The Console

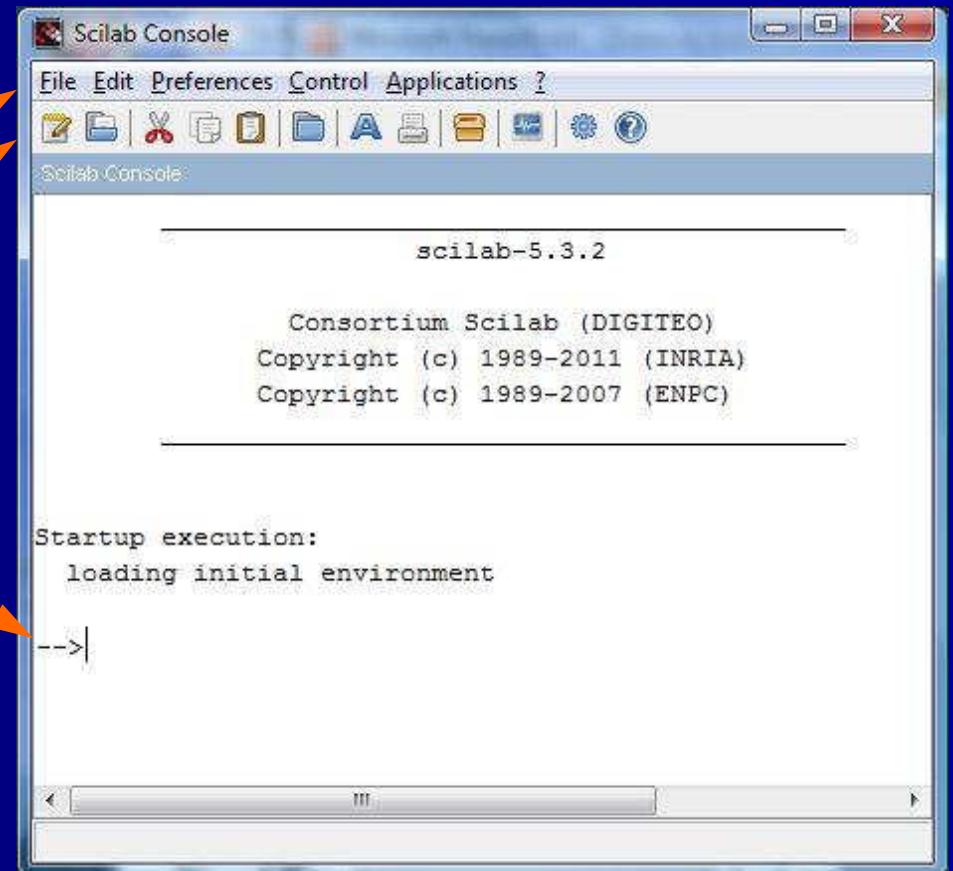
Click on Scilab's shortcut icon to open the Console
(Command Window in Matlab*):

Menu bar

Toolbar

Command prompt

If no shortcut icon has been created: Click: Start\All Programs\scilab\scilab (do not select Scilab Console)



*) The Console has other names as well: Workspace, Startup/Main Window, etc.

Folks:

**Here it comes, the lesson on
MathCad that Jim gave me back in
1991, transformed to Scilab. A
lecture worth gold in three slides**

Scilab in 15 minutes

(1/3): write a script

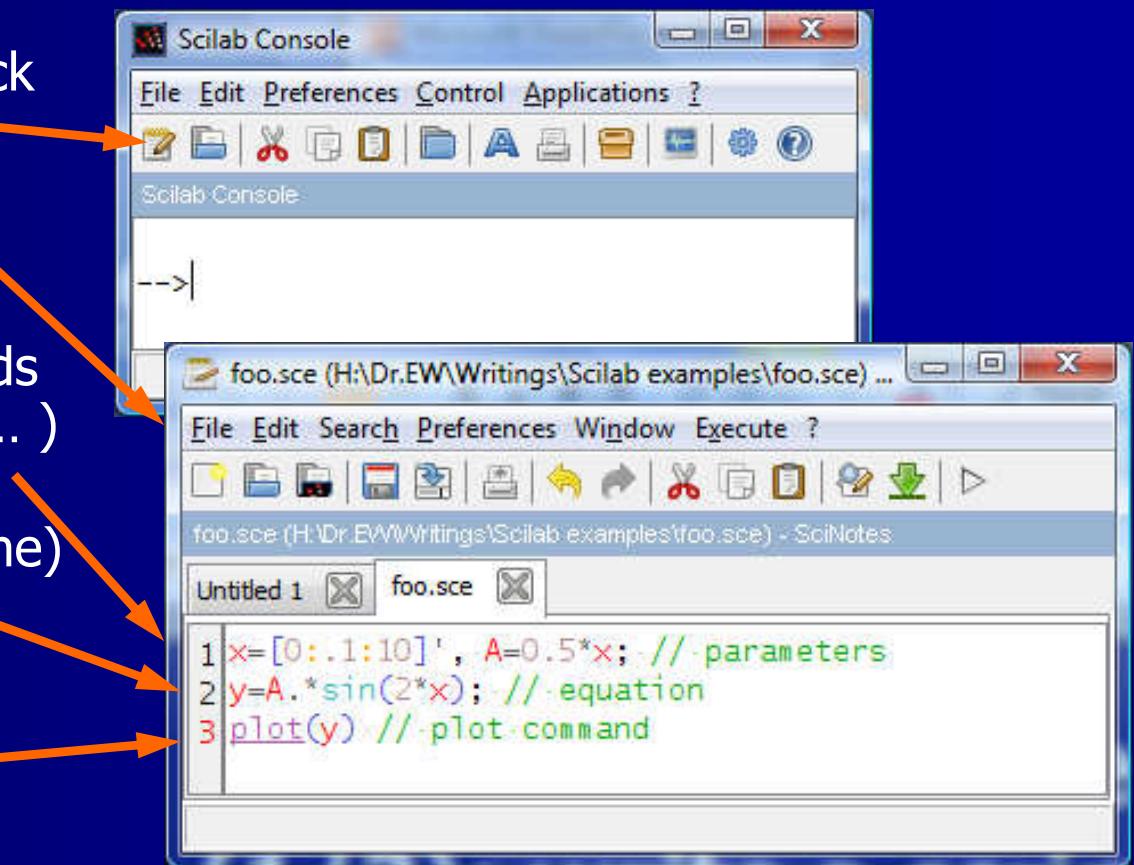
Recall how Jim taught me MathCAD in 15 minutes? Now we'll repeat that lesson in Scilab. We do it by using the Editor (SciNotes):

Step 1: On the Console, Click the leftmost icon on the toolbar. The Editor pops up

Step 2: Define whatever variables your function needs (row 1). Note comment (`//...`)

Step 3: Next, define the (sine) function in case (row 2)

Step 4: Finally, write the plot command (row 3)



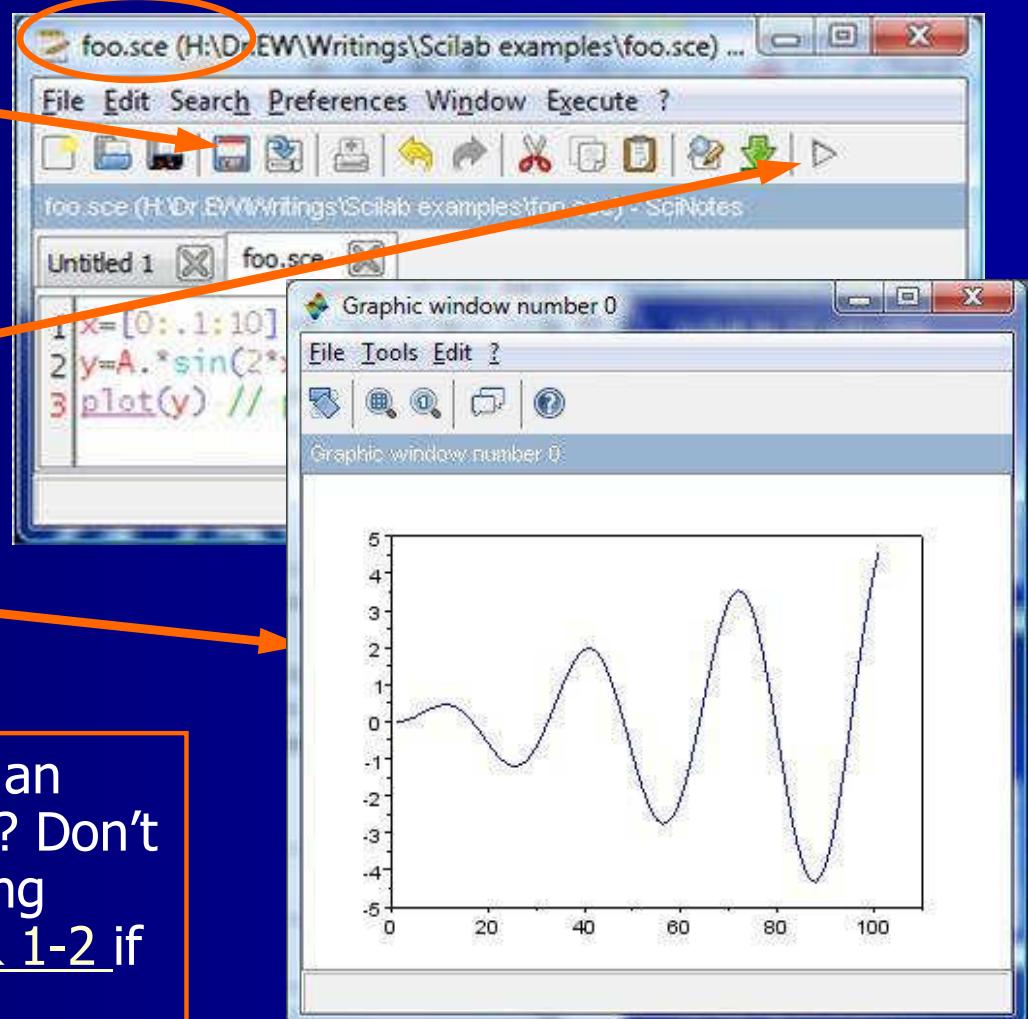
Scilab in 15 minutes

(2/3): save and plot

Step 5: Save the script by Clicking on the Save icon and name it e.g. foo.sce

Step 6: Finish by running (executing) the script by a Click the Execute icon (a second one came with 5.3.2)

Step 7: Up pops the Graphics Window with the a plot of the defined equation



Did you have problems or get an error message on the Console? Don't worry, we'll return to everything later. Jump to [Examples 1-1 & 1-2](#) if you are in a hurry.

Scilab in 15 minutes

(3/3): discussion

This exercise showed the essentials of Scilab in engineering applications:

- Scilab's user interface consists of three main windows:
 - The **Console**, which pops up when Scilab is opened and on which it outputs textual data (numeric answers, error messages, etc.)
 - The **Editor** (SciNotes), which is the main tool for writing, saving, and executing scripts (programs)
 - The **Graphics Window**, on which Scilab presents plots
- The recipe for using Scilab is the one that Jim taught me:
 - First you declare the **variables** that are needed
 - Then you define the **function** that you want to plot
 - And finally, plug in the **plot** instruction



That was Scilab
Let's go pizza

The Console's menu bar

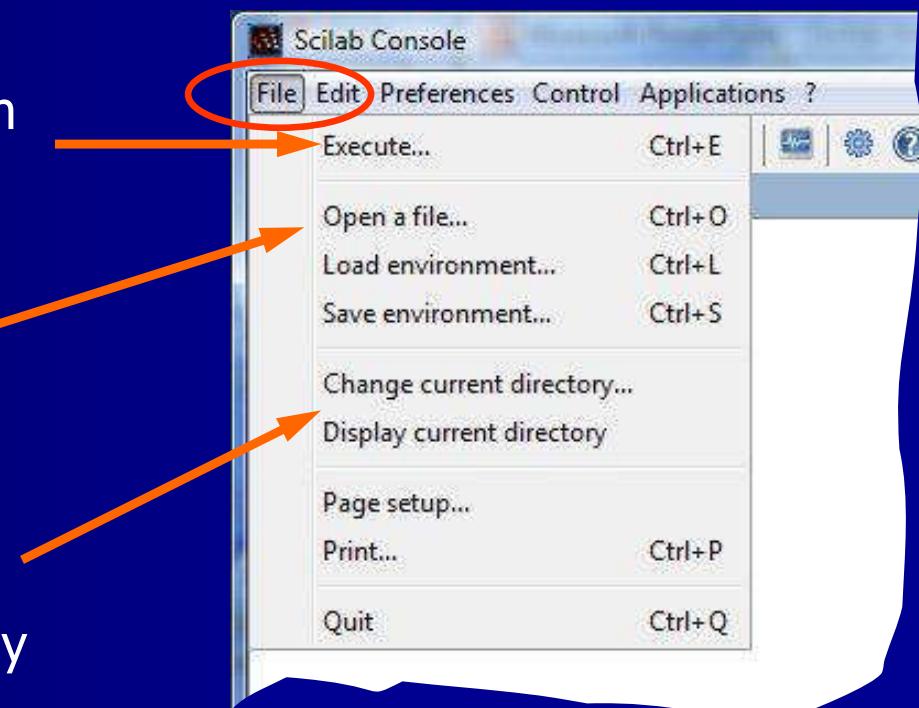
(1/6): File

Among the functions under the File drop-down menu that you will quickly encounter are:

Execute... : From here you can run Scilab scripts (or from the Editor, as seen later)

Open... : Similar to the Open... command in MS Office programs

Change current directory...,
Display current directory:
Pay attention to those two, they will be needed to tell Scilab where to look for a script that you want to open



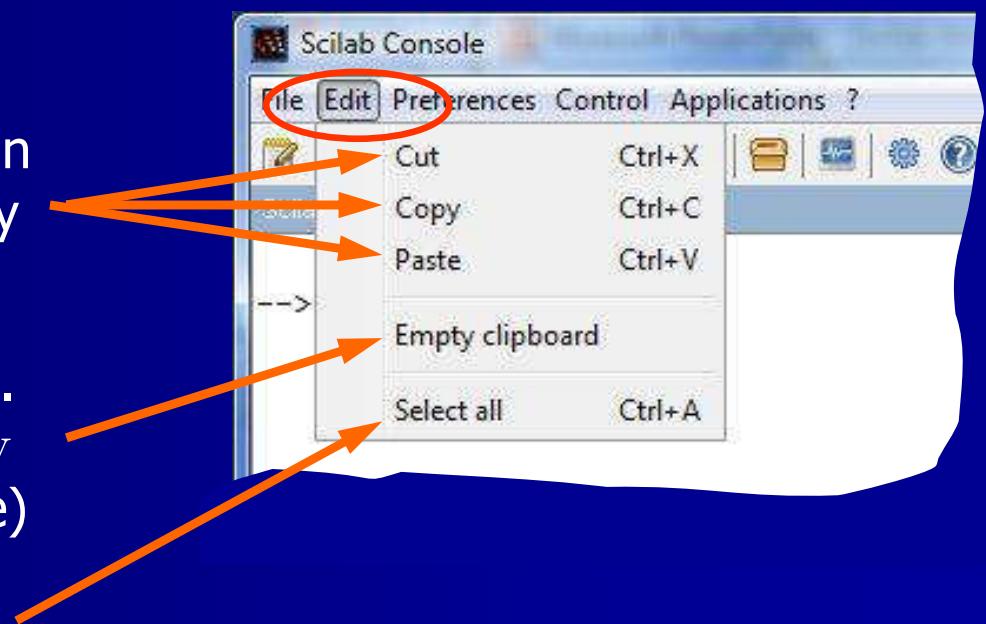
The Console's menu bar (2/6): Edit

The functions under the Edit drop-down menu are self-explanatory.

The Cut, Copy, and Paste commands have their own icons in the toolbar. You also find them by right-clicking on the PC mouse

Be careful with Empty clipboard. You may not be able to use Copy after clicking it! (Happened to me)

I have used Select all a lot to copy-paste the demos in this presentation

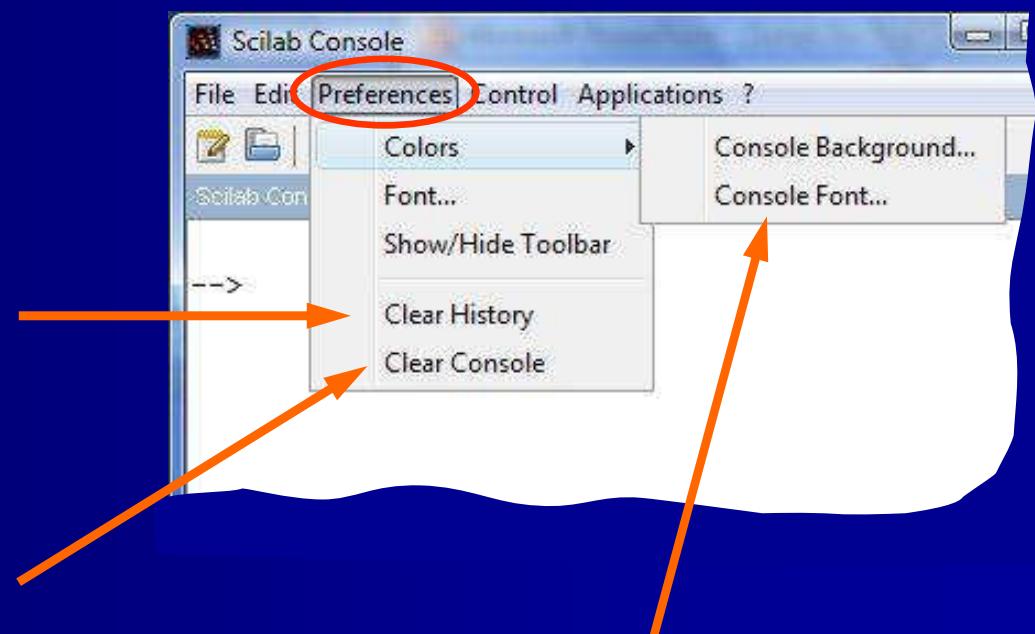


The Console's menu bar (3/6): Preferences

The functions under the Preferences drop-down menu are quite similar to what you can find on a PC

I can only guess that Clear History is similar to Clear Private Data in Firefox, but there is no Show History alternative and Help is not helpful

Clear Console empties the console. You achieve the same by pressing F2

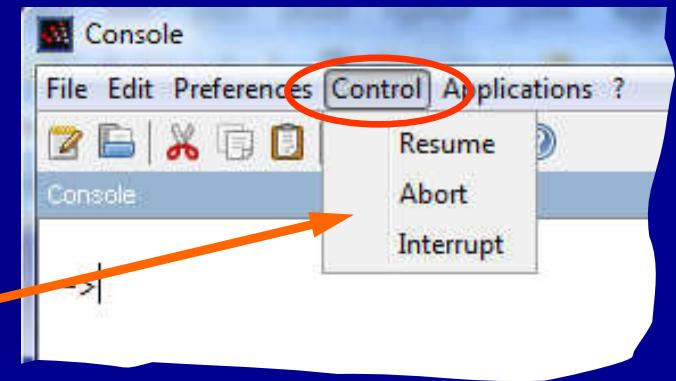


Change visual appearance of the Console

The Console's menu bar (4/6): Control

I did not need the Control drop-down menu a single time while doing this presentation, so obviously it is not very useful

My guess would be that the Resume , Abort, and Interrupt alternatives give the user a way to interfere with the execution of a program



The Help Browser is not very helpful and it does not even recognize the Interrupt command

The Console's menu bar (5/6): Applications

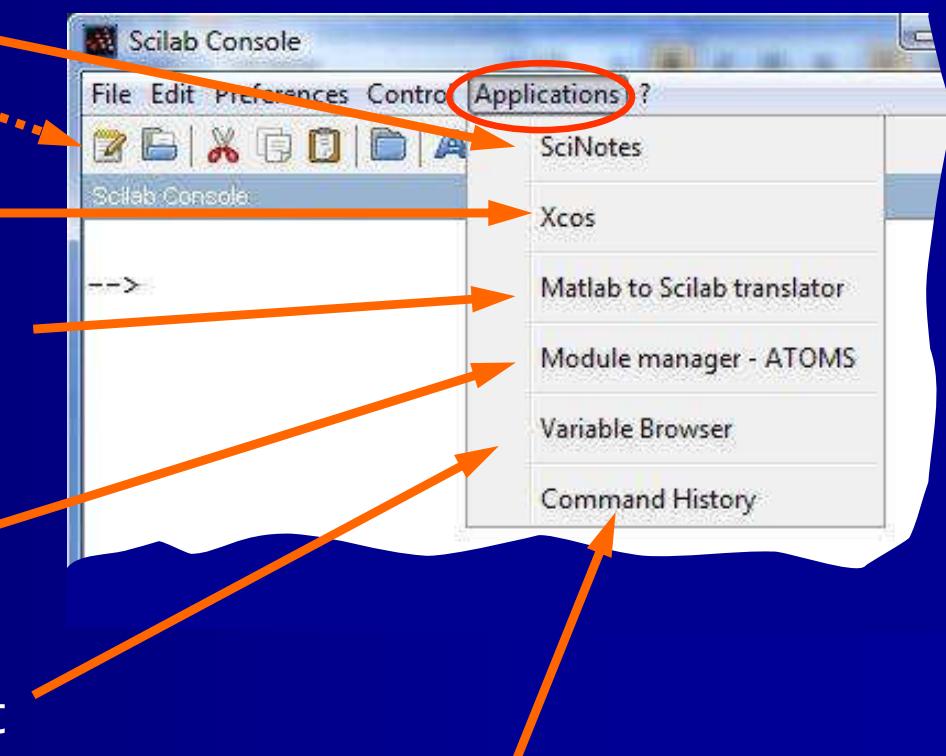
SciNotes: Opens Scilab's Text Editor (same as Launch SciNotes in the toolbar)

Xcos: Opens Xcos

Matlab to Scilab translator:
Used to translate a Matlab .m-file to a Scilab .sci file

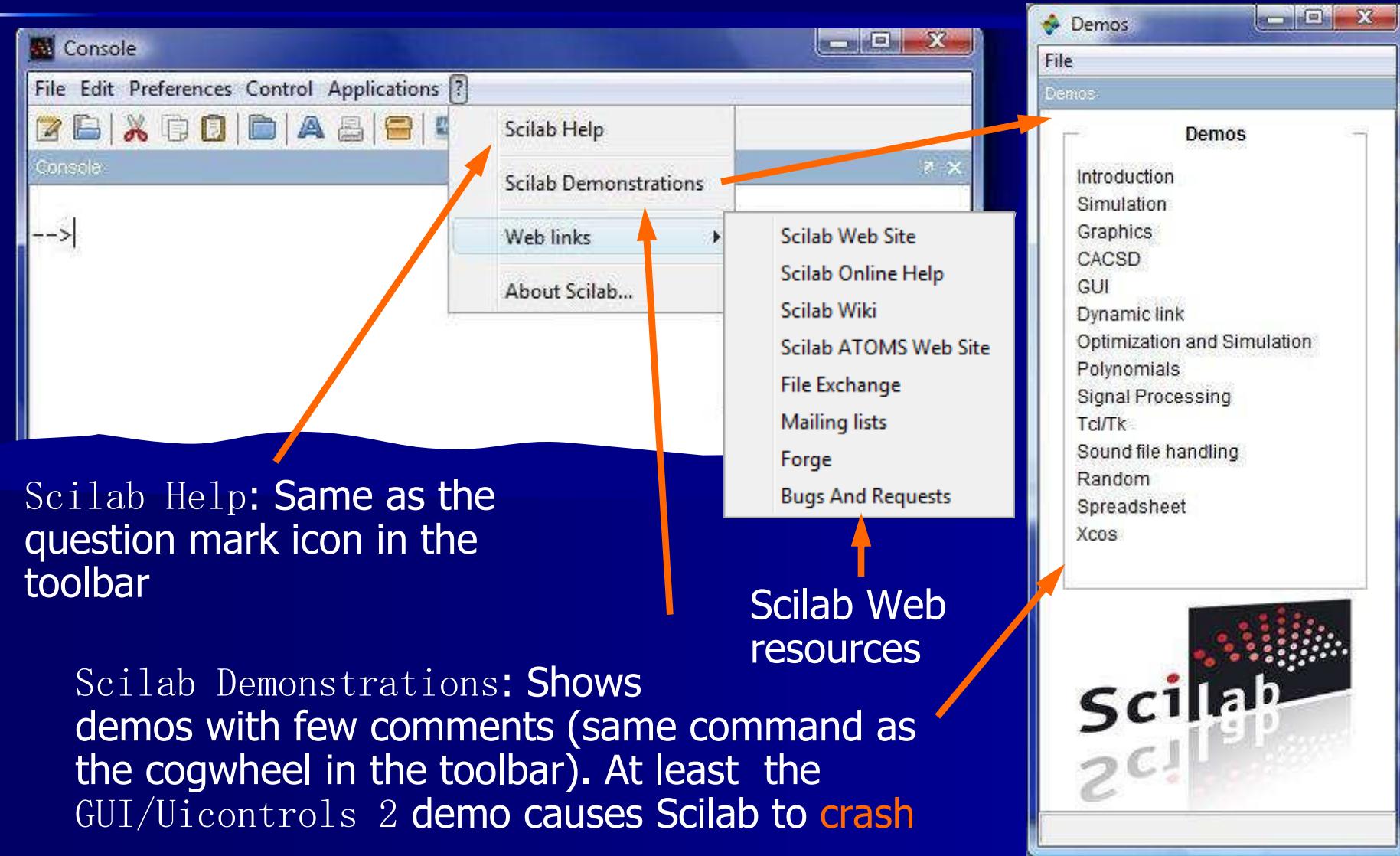
Atoms: Opens the [online](#) module manager

Variable Browser: Opens a list with variables (same as the browsevar; command)



Command History: Opens a list with commands used

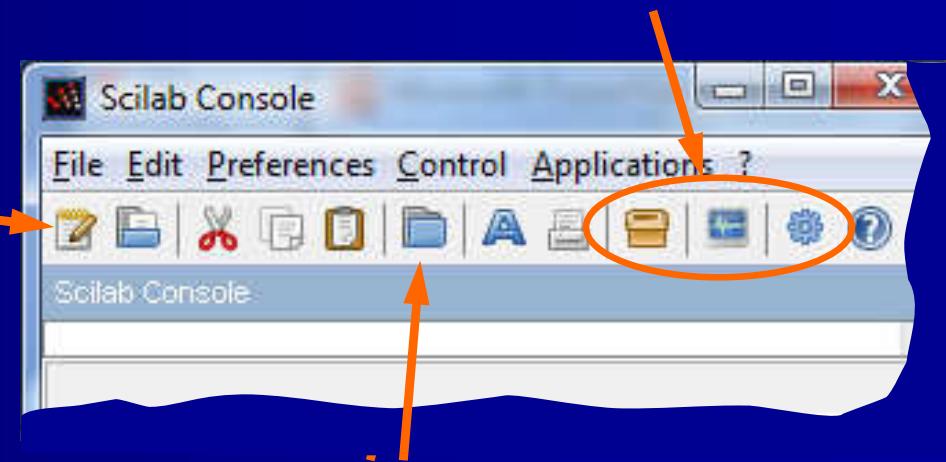
The Console's menu bar (6/6): Help alternatives



The Console's toolbar

Launch Editor: Opens Scilab's Editor (SciNotes, another part of its Integrated Development Environment (IDE). Basic tutorials seldom stress the fact that **normally we work with (write, edit, save, run) executable Scilab scripts on the Editor, not on the Console.** The Editor is presented a few slides below

The Atoms, Xcos, and Demonstrations icons came with Scilab 5.2



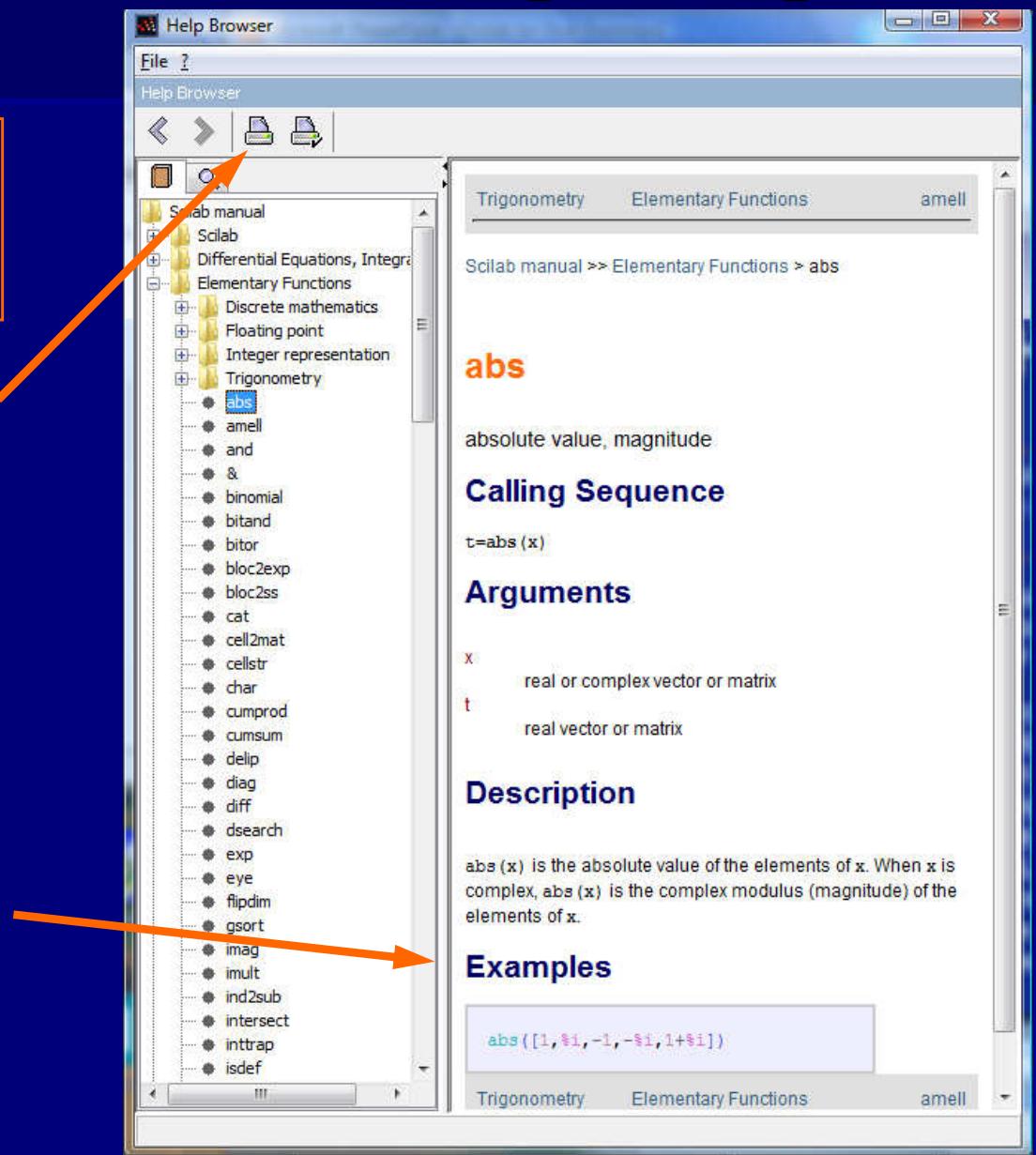
Change Current Directory: It can also be found under File in the menu bar. You need it to point out from which directory (folder) Scilab should search for a script that you want to execute (run)

The Help Browser (1/3)

In the Console, Click on the Help Browser icon to open it

Help discussions become more readable if you print them as PDF files

The Help Browser is a brief “encyclopedia” of Scilab’s main features and functions. Explanations of functions are augmented by **examples** (see next slide for a demo), but this does not compensate for a good tutorial

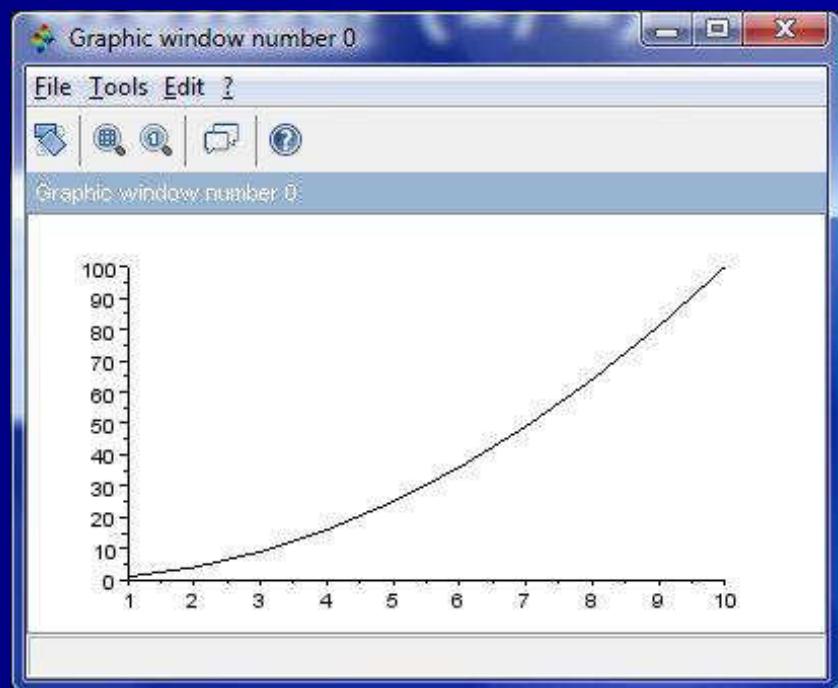


The Help Browser (2/3)

1. In the script box with examples, Click on the Execute icon to see how the scripts executes (not all work)
2. The Graphics Window with the plot pops up (in this cases it briefly flashes the first plot)

Examples

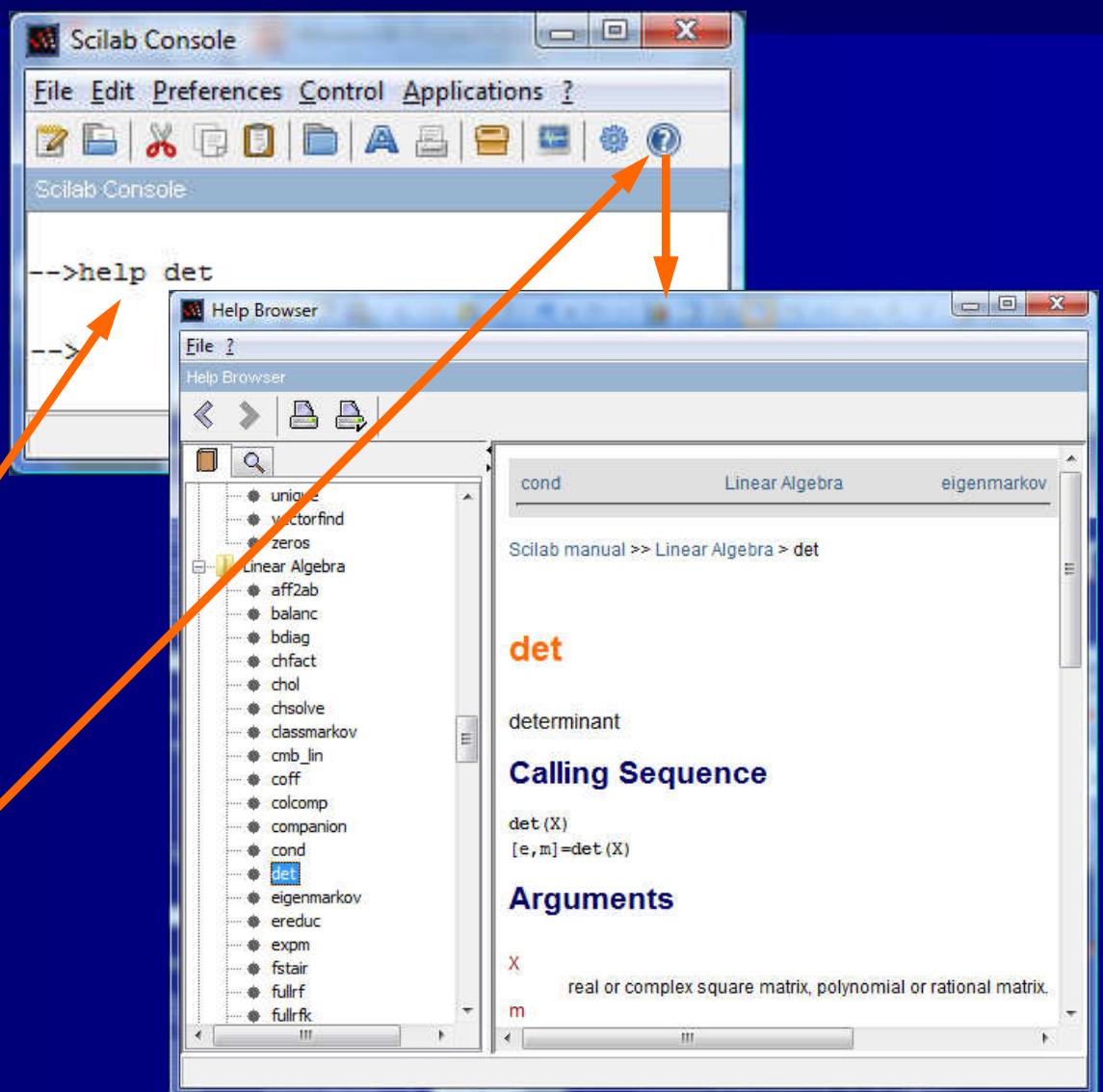
```
def([y]=f(x),"y=sin(x)+cos(x)") >   
x=[0:0.1:10]*%pi/10;  
fplot2d(x,f)  
clf();  
fplot2d(1:10,'parab')
```



3. Click on the Editor icon and the script is transferred to Scilab's text Editor where you can play with it (must be saved before it can be run)

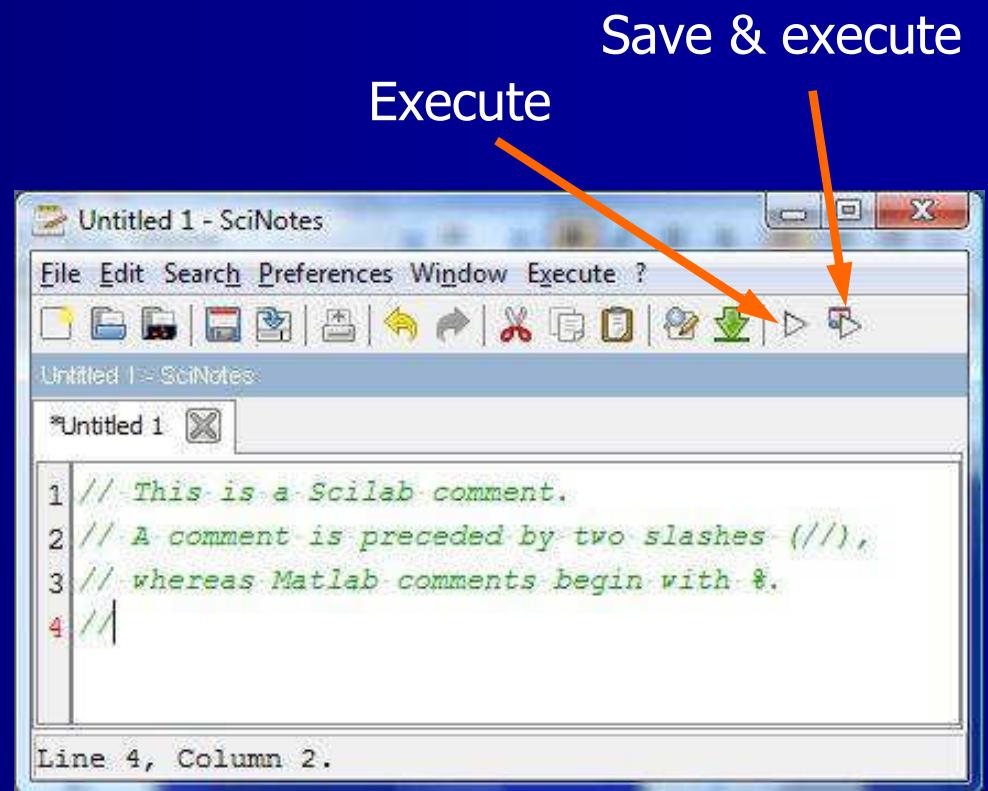
The Help Browser (3/3): help function_name

To find the proper use of any function—assuming that the name is known—the Help Browser can be opened from the Console by entering the command `help function_name` command, in the shown case `help det()` (the brackets can be omitted). The alternative is to open the Browser with the Help icon



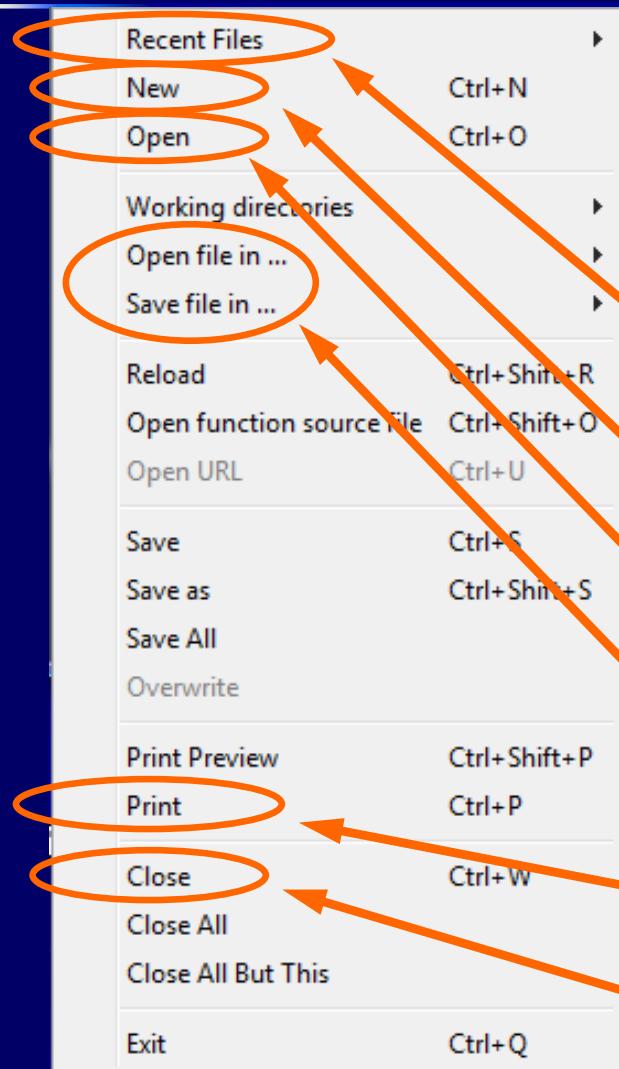
The Editor (SciNotes)

- The (Text) Editor is where executable Scilab scripts are written, maintained & run
- Open the Editor by clicking on the Launch SciNotes icon in the Console, or by clicking: Applications\SciNotes
- A Scilab script is a text file with a name of the type
 - * .sce (the alternative
 - * .sci is also used but
 - * .sce is the default)
- It is good practice to use scripts also for small tasks.
Then all “projects” are saved and commented, ready for reuse



But don't forget to create a properly organized archive for your programs!

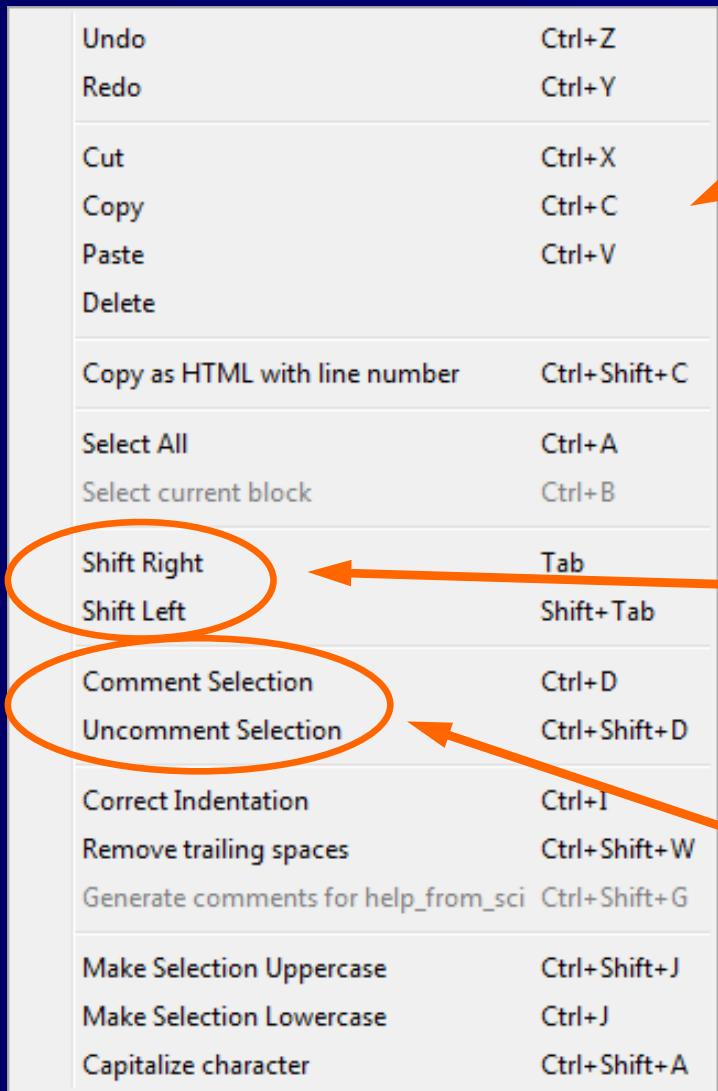
Editor menu bar (1/5): File



File commands that you are most likely to run into:

- Recent files **gives quick access to recently edited scripts**
- New **opens a second tab for a new script to be edited**
- Open **opens a saved script into the Editor**
- Open file in... **and Save file in... do not work in Scilab 5.3**
- Print **is an ordinary print command**
- Close **closes the file in case**

Editor menu bar (2/5): Edit



Commands under Edit are mostly self-explanatory. Note however the following four:

Shift Right/Left:
Indent/unindent a row by one step (this pair should be on the toolbar)

Comment/Uncomment Selection:
Add/delete a comment

Editor menu bar (3/5): Preferences

The Preferences drop-down menu allows you adjust Editor settings to your liking

I had difficulties reading scripts on the Editor (poor contrast with default settings) and used Set Colors... and Set Font to change from default values

Line numbering
Current file encoding
Default SciNotes file encoding
Line Endings
Page Setup
Horizontal wrapping
Auto-completion on (, ... Ctrl+H
Auto-completion on if,function,... Ctrl+Shift+H
Enable Auto Reformat
Highlight current line
Set Colors ...
Set Font
Reset default font
Configure Tabulation
Restore opened files on start-up

Editor menu bar (4/5): Preferences, comment

Users can send **bug reports** to Scilab's development team (link at <www.scilab.org>). I filed the following report (Bug 8802):

“Default color settings on the Editor produce poor contrast ... Changing font colors is tedious due to the excessive number of options under Preferences\Set colors... (an overkill, you could say). I would suggest default settings with just four colors (red, green, blue and black). ”

To which I got this answer:

“You can modify more easily the colors configuration in modifying the file: C:\Documents and Settings\Johnny\Application Data\Scilab\scilab-5.3\scinotesConfiguration.xml (or a path which is similar) ”

I found scinotesConfiguration.xml under C:\Program Files\scilab-5.3\modules\scinotes\etc\. XML color codes must be changed in this file. I wish you good luck

Editor menu bar (5/5): Execute



The Execute drop-down window contains three options:

... file with no echo: A simple execute command (same as clicking the Execute icon on the toolbar)

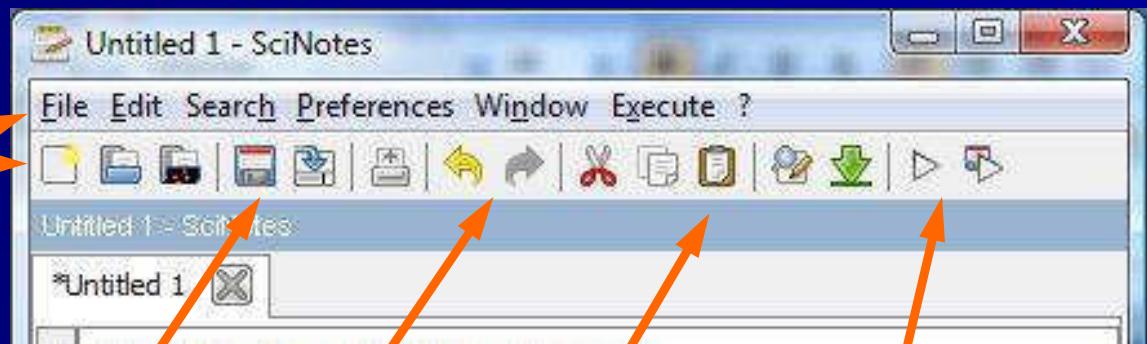
... file with echo: Executes the script and echoes it (shows it) on the Console

... until the caret, with echo: Beats me, what it means

The Execute commands used to be simpler. I have no idea why they changed them this way. My recommendation is to use the Execute icon on the toolbar (see next slide)

Editor toolbar

New... Opens a second tab for a new script to be edited (the same command can be found under File)



The Save icon looks like the Dutch tricolor, but you'll get used to it. The next one is Save as...

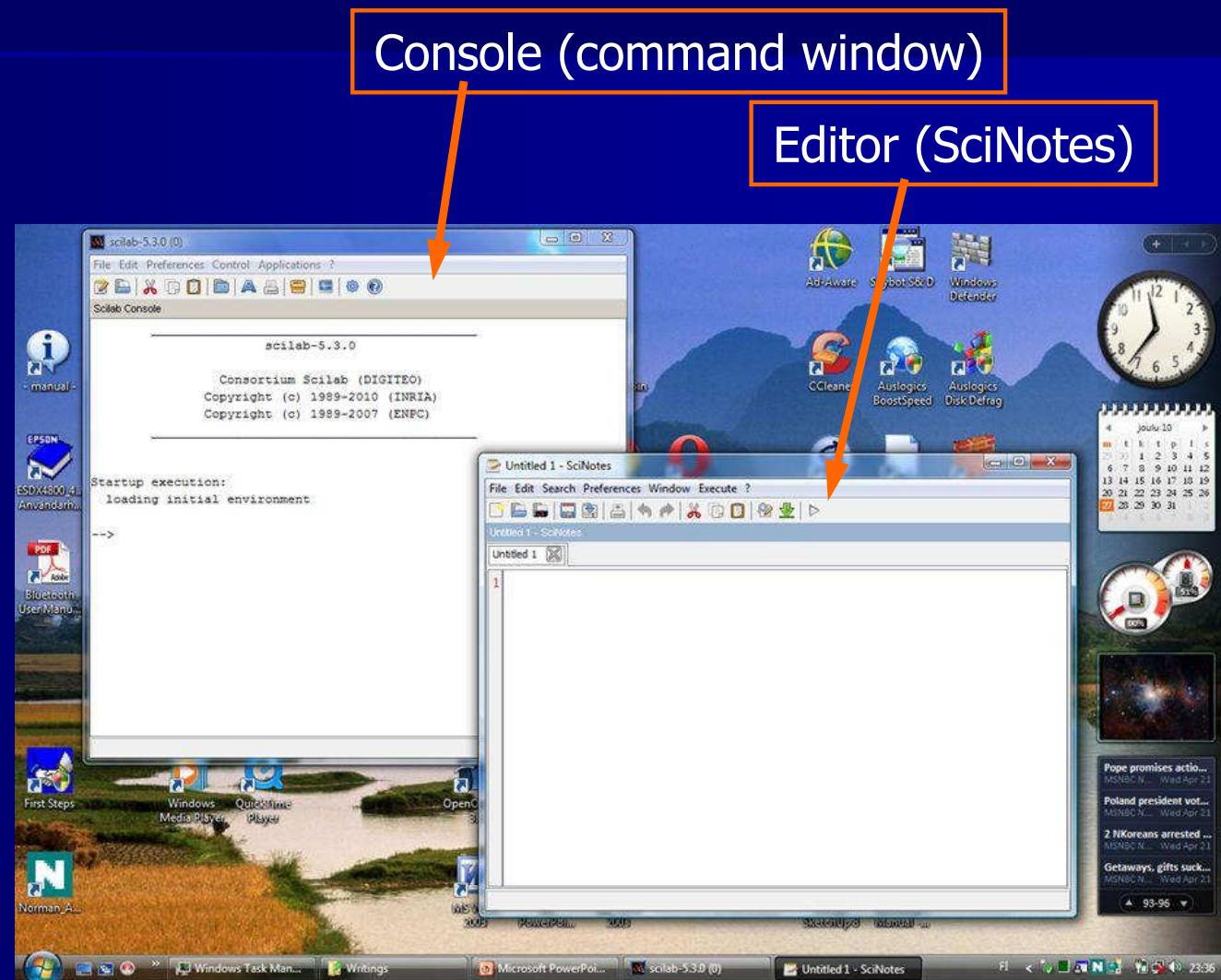
The Undo/Redo arrows are quite normal

The Paste icon is a bit unusual (French?)

The Execute (or Save & execute) icon is what you normally use to run a script

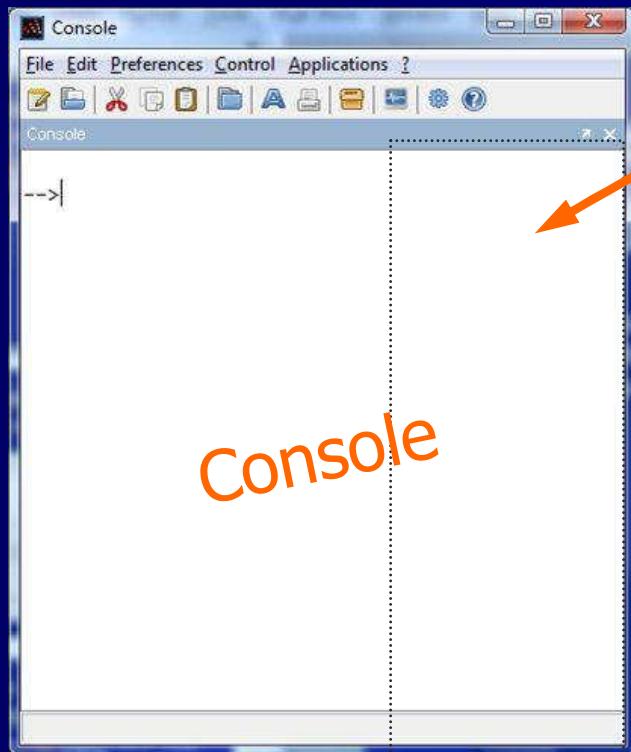
Ready to go

Your desktop should now look something like the one here. As we have seen, both the Editor and the Console are needed since when the scripts—created on the Editor—are executed numeric outputs is returned to the Console



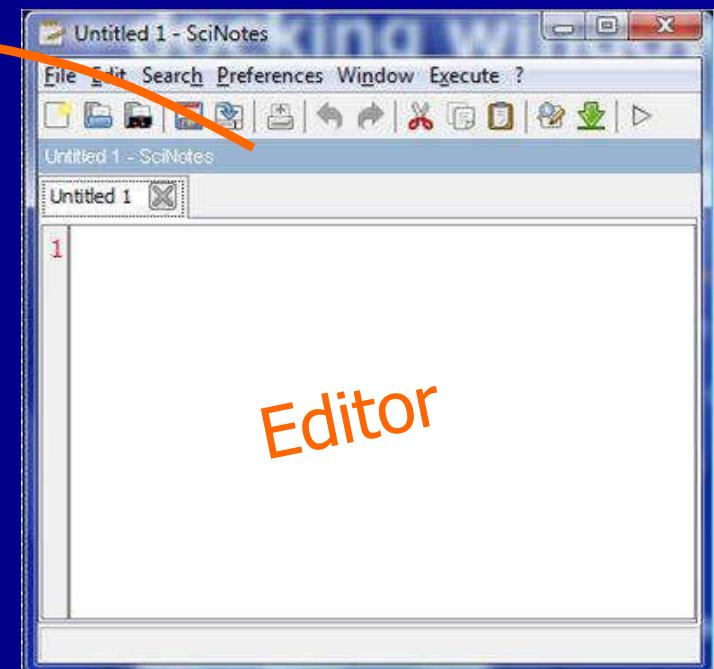
One more thing (1/2): docking windows

- It is possible to **dock** Scilab windows; i.e., to form a unified workspace similar to the one in Matlab. Here is how to do it:



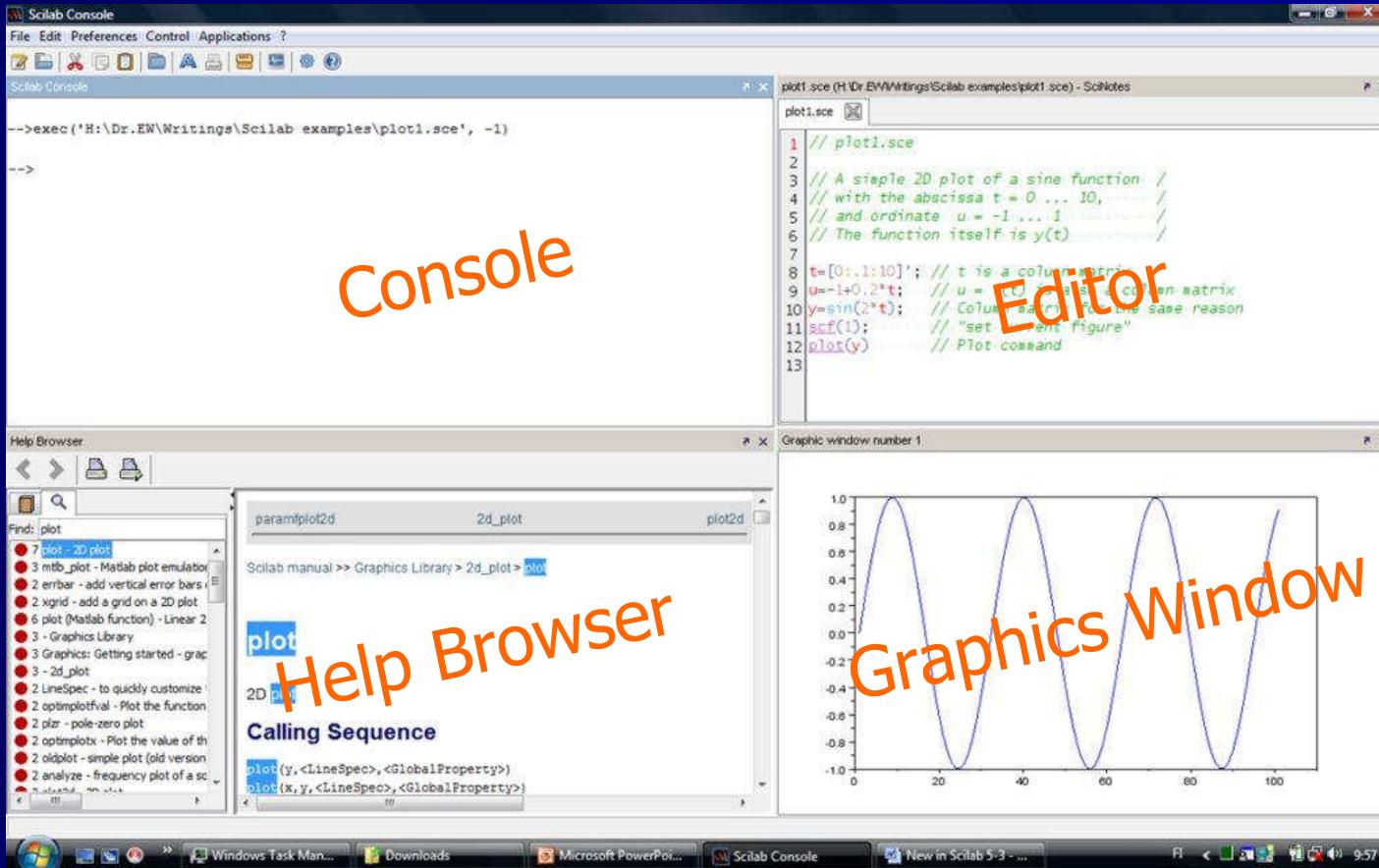
Console

Press the left mouse button on the darkened bar of an active window, drag over another window and release. The next page shows one case



Editor

One more thing (2/2): docking windows



Each window part has an arrow in the upper right-hand corner, by which you can release it from docking

On scripts and functions

- Scilab has two command types:
 - **Scripts**. A set of commands used to automate computing. Script commands are normally returned to the Console, but plots are returned to the Graphics Window
 - **Functions** (macros). Short programs that interface with the environment through input and output variables. A list of common **built-in functions** is given on the next slide. Functions defined by the user can either be local (integrated in a script) or global (stored as a separate file and accessible to any script)
 - I may use the term “code” to make general references to either scripts or functions
- As was already said—and will repeated—one should rather create scripts and functions on the (Text) Editor (SciNotes)

Built-in functions

Below is a list of common math functions in Scilab. A full list of built-in functions can be found under Help\Elementary Functions, which also explains requirements on **arguments** (there are both mandatory and optional arguments).

sin(), cos(), tan(), cotg()	Trigonometric functions, e.g. sin(.2*%pi)
asin(), acos(), atan()	Arc functions
sinh(), cosh(), tanh(), coth()	Hyperbolic functions
asinh(), acosh(), atanh()	Inverse hyperbolic functions
sqrt(), exp()	Square root, e.g. sqrt(2) / exponent
sum()	Sum
min(), max()	Minimum / maximum value
abs(), sign()	Absolute value, e.g. abs(sinc(x)) / sign
real(f), imag(f)	Real & imaginary parts of a complex f

Predefined variables & constants

Main predefined and write-protected variables/constants are:

<code>%i</code>	$i = \sqrt{-1}$	Imaginary unit
<code>%pi</code>	$\pi = 3.1415927\dots$	Pi
<code>%e</code>	$e = 2.7182818\dots$	Napier's constant e
<code>%eps</code>	$\epsilon = 2.22 \cdot 10^{-16}$	Precision (machine dependent)
<code>%inf</code>		Infinite (not mathematically infinite)
<code>%nan</code>		Not a Number
<code>%s</code>	s	Polynomial variable
<code>%z</code>	z	Polynomial variable
<code>%t , %T</code>	true	Boolean variable
<code>%f , %F</code>	false	Boolean variable

Scilab operators (1/2)

The list contains the majority of operators used in Scilab. Many will be explained in detail later.

;	End of expression, row separator
,	Instruction, argument or column separator
'	Conjugate (matrix) transpose, string delimiter*
.'	Non-conjugate transpose
[] , [] '	Vector or matrix definition concatenation, transposed matrix
()	The pair of left/ right parenthesis is used for various purposes
+ , -	Addition, subtraction
* , .*	Multiplication, element-by-element multiplication

*) Both simple ('') and double ("") quotes are allowed to define character strings

Scilab operators (2/2)

/ , ./	Right division, element-by-element right division
\ , .\	Left division, element-by element left division
^ or ** , .^	Power (exponent), element-by-element power
.*	Kronecker product
./. , .\.	Kronecker right and left division
	Logical OR
&	Logical AND
~	Logical NOT
==, >=, <=, >, <, <>, ~=	Equal to, equal or greater than, equal or less than, greater than, less than, not equal to (two alternatives)

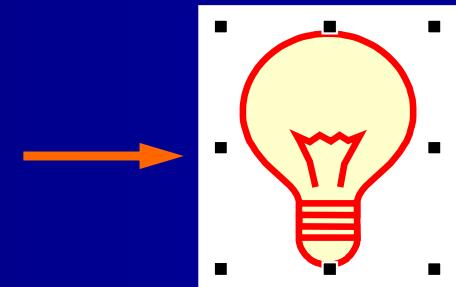
Computing terminology: a brief introduction

- **Arguments:** Values provided as inputs to a command (input arguments) or returned by the command (output arguments)
- **Command:** A user-written statement that provides instructions to the computer ("statement" is an often used alternative)
- **Default:** Action taken or value chosen if none has been provided
- **Display:** To output a listing of text information on the computer screen
- **Echo:** To display commands or other input typed by the user
- **Execute:** To run a program or carry out the instructions specified in a command
- **Print:** To output information on a computer printer (often confused with "display")
- **Returns:** Results provided by the computer in response to a command

On “handles”

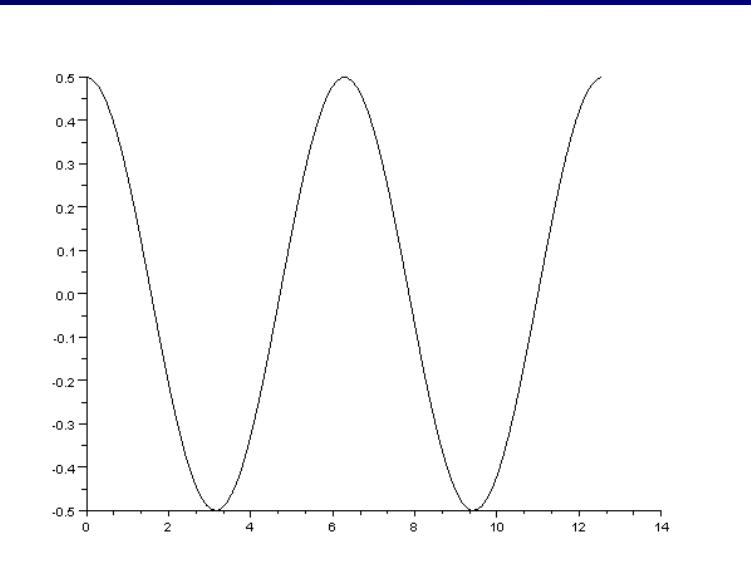
You will often see Scilab’s Help Browser refer to a “handle,” but Help does not provide a helpful explanation of the term. Here is a brief account:

- In graphics software the word handle refers to the points placed around a figure or plot that allow you to manipulate the object (see figure)
- A Matlab tutorial gives the following explanation that is also valid for Scilab: “Whenever Matlab creates a graphics object, it assigns an identifier (called **handle**) to it. You can use this handle to access the object’s properties.”
- You need handles to edit graphical plots beyond the means offered by basic plot functions (`plot2d()`, `plot3d()`, etc.)
- We’ll return handles when discussing graphics & plotting (Ch. 7)



Check handles with gcf()

- The function `plot2d()` produces the plot below
- The command `gcf()` gives the list to the right
- The list is the handle for the defined function (Scilab literature also refers to individual rows in the list by the term "handle")



```
-->x = linspace(0,4*%pi,100); plot2d(x,0.5*cos(x))
```

```
-->f = gcf()
```

```
f =
```

Handle of type "Figure" with properties:

```
=====
```

```
children: "Axes"
```

```
figure_position = [567,485]
```

```
figure_size = [628,592]
```

```
axes_size = [610,460]
```

```
auto_resize = "on"
```

```
viewport = [0,0]
```

```
figure_name = "Graphic window number %d"
```

```
figure_id = 0
```

```
info_message = ""
```

```
color_map= matrix 32x3
```

```
pixmap = "off"
```

```
pixel_drawing_mode = "copy"
```

```
anti_aliasing = "off"
```

```
immediate_drawing = "on"
```

```
background = -2
```

```
visible = "on"
```

```
rotation_style = "unary"
```

```
event_handler = ""
```

```
event_handler_enable = "off"
```

```
user_data = []
```

```
tag = ""
```

foo

- The term “foo” is used in many tutorials. It may be confusing if you are not intimately familiar with programming
- Simply stated, foo can be interpreted as “something comes here.” The professional expression is **placeholder name**, also referred to as **metasyntactic variable**
- Example:

```
for k = 1:2:n  
    foo;  
end
```

- Alternative placeholder names that you may come across are foobar, bar, and baz. I prefer to use dots (....)

3. Playing with the Console & Editor

Those awkward first steps; a bit
about what Scilab does



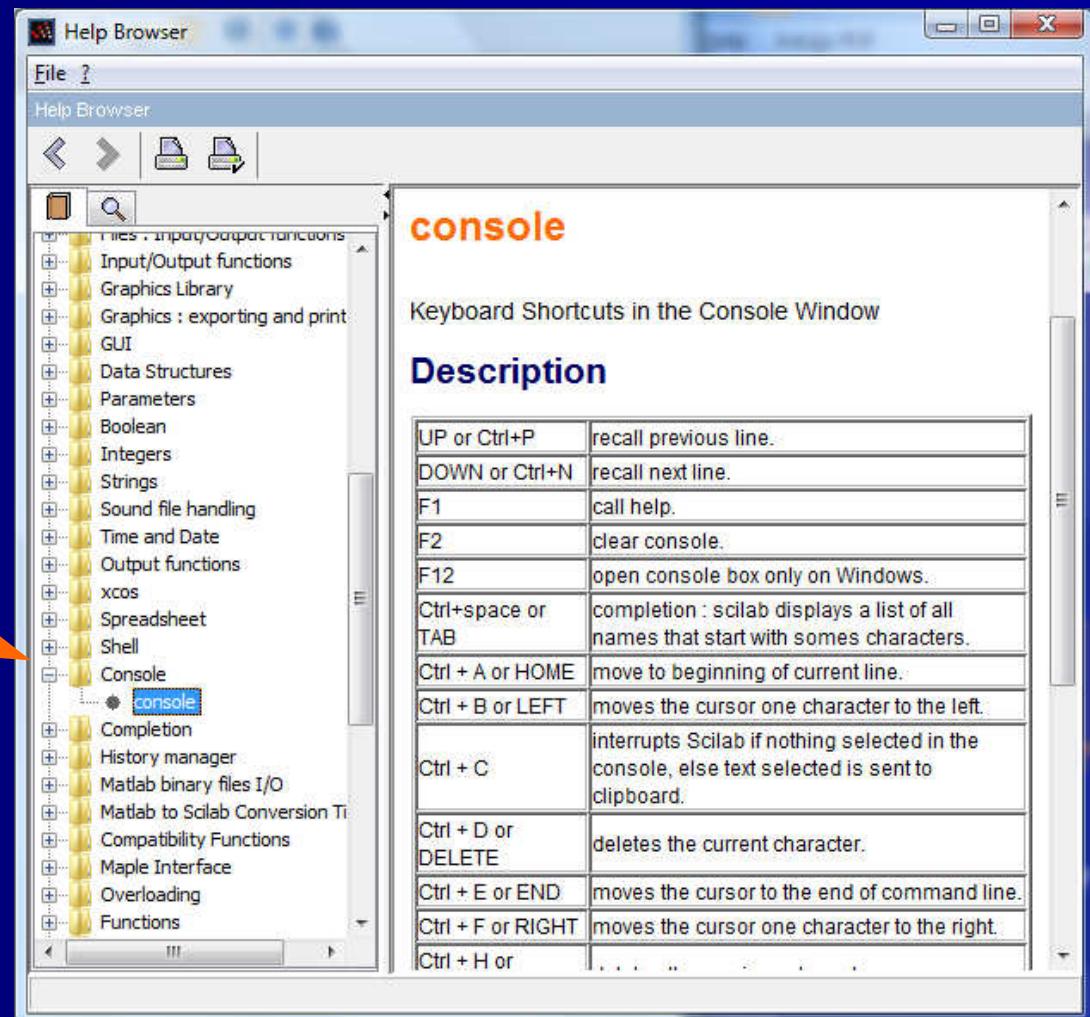
[Return to Contents](#)

Console keyboard shortcuts

Keyboard shortcuts allow speedier execution of commands, but require frequent use to stay memorized

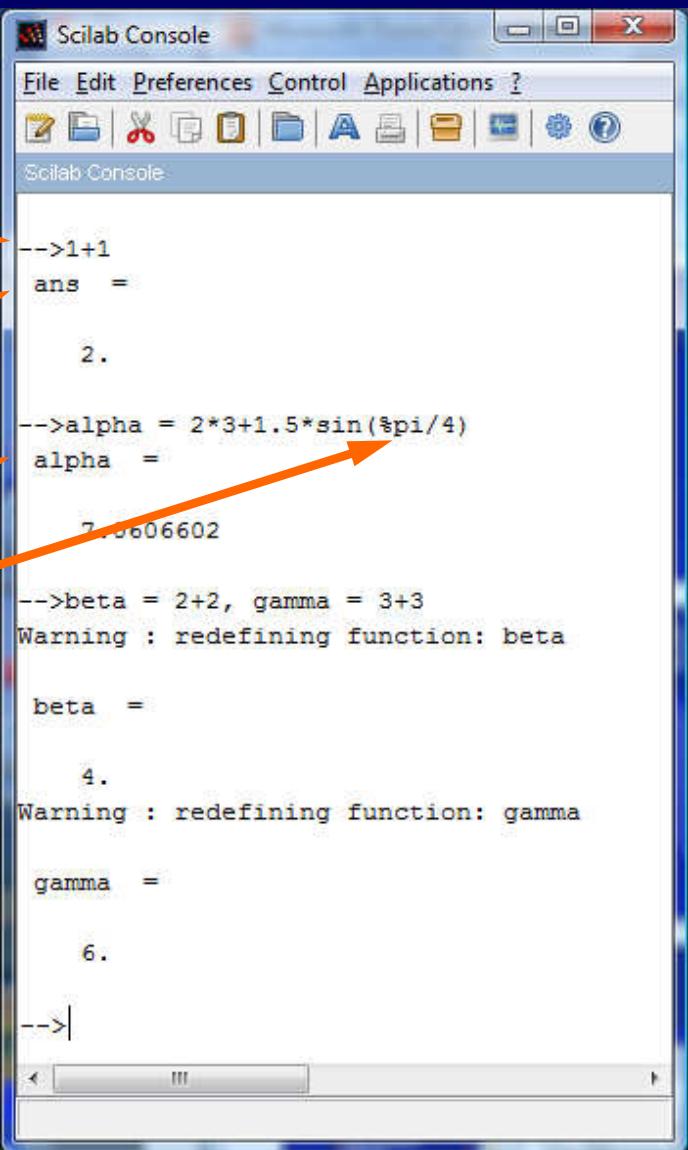
In the Help Browser,
Click: Console/console
for a list of keyboard
shortcuts

The simplest ones to memorize are:
F1 = Open Help Browser
F2 = Clear Console



Simple calculations

- The Console can be used as a **calculator** by writing arithmetic expressions after the command prompt and pressing Enter
- If no variable name is given, Scilab uses the inbuilt variable `ans`
- When a variable name is given (here `alpha`) it will be used instead. π is an inbuilt variable (constant) represented by `%pi`
- Expressions can be written on the same line by separating them with a comma (the warning can be ignored)
- Scilab displays an executed command unless it ends with a semicolon (`;`)



The screenshot shows the Scilab Console window with the following session history:

```
-->1+1
ans  =
2.

-->alpha = 2*3+1.5*sin(%pi/4)
alpha =
7.0606602

-->beta = 2+2, gamma = 3+3
Warning : redefining function: beta
beta =
4.
Warning : redefining function: gamma
gamma =
6.

-->|
```

Four orange arrows point from the list items to specific parts of the console output. One arrow points to the first calculation `-->1+1`. Another points to the result `ans = 2.`. A third points to the assignment of `alpha` with its value. The fourth points to the assignment of `beta` and `gamma`, highlighting the warning message about redefining functions.

List of variables (1/2)

The command `who` (+ Enter) produces a list of some Scilab variables. At least on my Windows Vista laptop the columns are right aligned (French logic or a bug?). Note that variables from the previous example are displayed

```
-->who

Your variables are:

      gamma          beta          alpha
fmtchildren    linspace      %h_delete
x              %c_a_c        delete_frame
subdemolist   demo_gui_update create_frame
resize_gui    WSCI           home
help           atomsguilib   atomslib
modules_managerlib simulated_annealinglib genetic_algorithmslib
parameterslib scicos_autolib   scicos_utilslib
fft            demo_toolslib  development_toolslib
spreadsheetlib tclscilib     m2scilib
texmacslib     statisticslib windows_toolslib
compatibility_functilib special_functionslib sparselib
stringlib      %s            polynomialslib
%z             optimbaselib  neldermeadlib
optimsimplexlib linear_algebraolib jvmlib
interpolationlib iolib         dynamic_linklib

```

List of variables (2/2)

- The command `browsevar` opens the Variable Browser window (it used to be called Browser Variables, thus the command `browsevar`)
- The list that pops up gives information about the type and size of each variable
- Recall that the Variable Browser also can be called via the menu bar:
Applications/Variable Browser

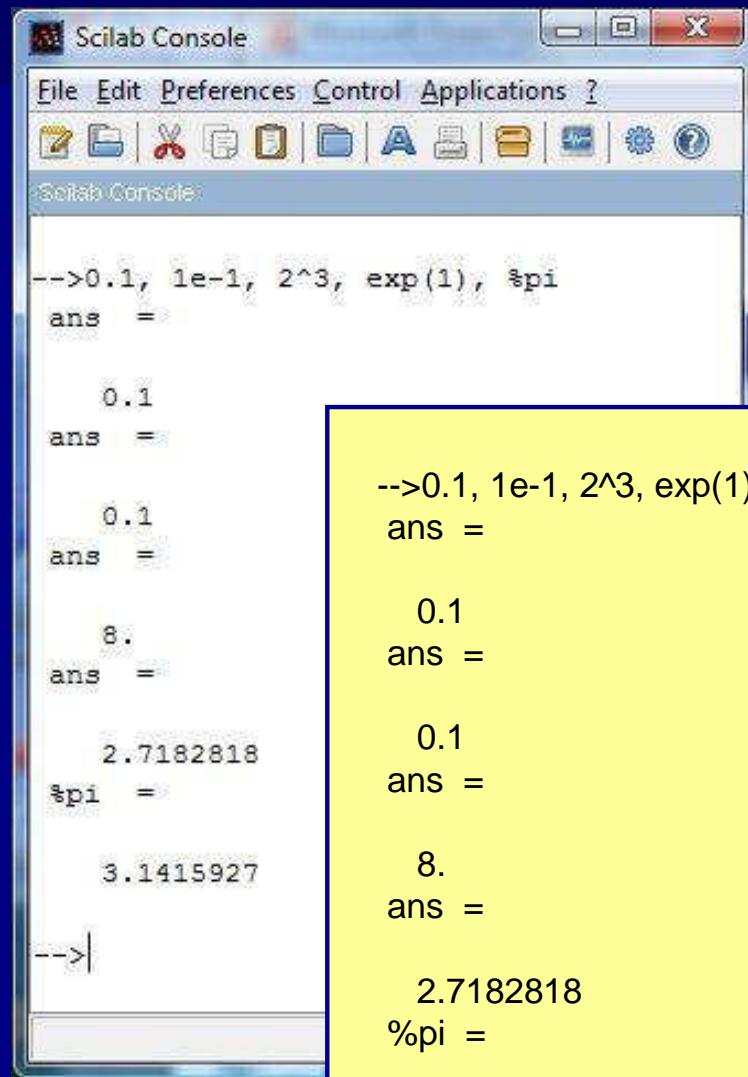


The screenshot shows the "Variable Browser" window, which is a standard Windows-style dialog box. The title bar says "Variable Browser". Below the title bar is a menu bar with "File" and "Filter" options. The main area is a table with columns: "Icon", "Name", "Bytes", "Type", and "Visibility". The table lists various variables with their corresponding details. An orange arrow points from the explanatory text in the slide to the "Variable Browser" window.

Icon	Name	Bytes	Type	Visibility
	gamma	3		1 local
	beta	3		1 local
	alpha	3		1 local
	x	102		1 local
abc	subdemolist	157		10 local
abc	WSCI	18		10 local
abc	home	11		10 local
abc	PWD	18		10 local
%T	%tk	2		4 local
%T	%pvm	4		4 local
%T	MSDOS	2		4 local
%T	%F	3		4 local
%T	%T	3		4 local
	%nan	3		1 local
	%inf	3		1 local
abc	SCI	14		10 local
abc	SCIHOME	29		10 local
abc	TMPDIR	27		10 local
%T	%gui	3		4 local
%T	%fftw	3		4 local
%T	%t	3		4 local
%T	%f	3		4 local
	%eps	3		1 local
	%io	4		1 local
	%il	4		1 local

Entering numbers

- Scilab allows numbers to be entered in different ways, as shown in this example
- Some expressions have alternate forms. For instance, there are three power expressions (^), (**) and (.^), but Scilab picks them in that calling order
- Note that e and π are given with **seven decimals**, which puts limits to the achievable accuracy (a function for **double precision** does exist)
- Consult Help if you need to change the display format
- **Note:** From now on I'll show only the contents of the Console (on light yellow background)



The image shows a screenshot of the Scilab Console window. The window has a title bar "Scilab Console" and a menu bar with File, Edit, Preferences, Control, Applications, and Help. Below the menu is a toolbar with various icons. The main area is labeled "Scilab Console". It displays the following text:
-->0.1, 1e-1, 2^3, exp(1), %pi
ans =

0.1
ans =

0.1
ans =

8.
ans =

2.7182818
%pi =

3.1415927
-->|



```
-->0.1, 1e-1, 2^3, exp(1), %pi
ans =
0.1
ans =
0.1
ans =
8.
ans =
2.7182818
%pi =
3.1415927
```

Computing precision (1/2)

```
-->a = 1 - 5*0.2
a =
0.

-->b = 1 - .2 - .2 - .2 - .2 - .2
b =
5.551D-17
```

Look at the two examples to the left. In both cases we are computing $1 - 5 \cdot 0.2$, but in two different ways

In the first case the answer is **correct** (0)

In the second case the answer is $5.55 \cdot 10^{-17}$, which quite obviously is **incorrect**

The reason is that numeric computing has **finite precision** (rounding errors)

We must take this limitation into account in Scilab simulations

Computing precision (2/2)

Here are two more cases where finite precision shows up. The answers should be 0 (zero) and T (True) respectively (Note that 1.225D-15, 1.225e-16, 1.225*10⁻¹⁶ and 1.225*10⁻¹⁶ express the same thing)

Assume that the mentioned variable *a* is part of a script with an if...then...else...end structure (conditional branching will be covered in Chapter 11). The result is that alternative 1 is never executed because *a* is never exactly zero

We must test *a* with some finite bounds, e.g.:

if $\text{abs}(a) < 1\text{e-}6$ then

.....

$|a| < 10^{-6}$

```
-->a = sin(%pi)
a =
1.225D-16

-->0.1 == 1.0 - 0.9
ans =
F
```

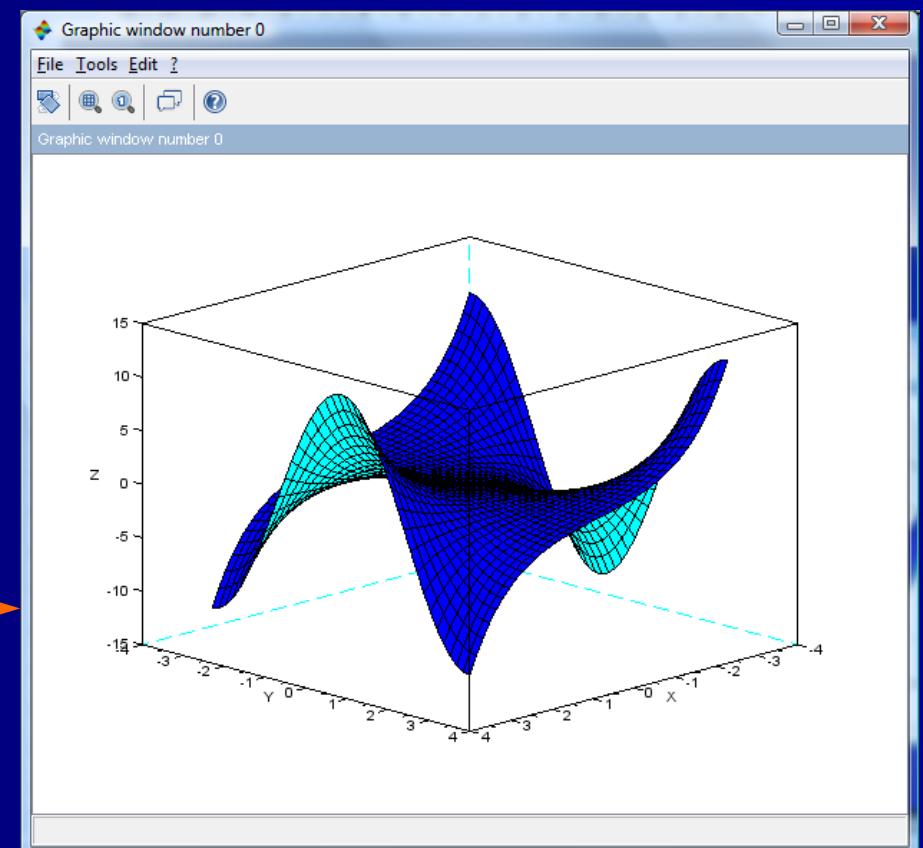
```
foo
if a == 0 then
    alternative 1
else
    alternative 2
end
```

Displaying graphics

- The Console can also be used to give commands for plotting graphics:

```
-->x = linspace(-%pi,%pi,40);  
  
-->y = linspace(-%pi,%pi,40);  
  
-->plot3d(x,y,sinh(x')*cos(y))
```

- The graphical picture is plotted in the **Graphics Window** that pops up automatically ([more in Ex. 1](#))
- The meaning of the entered code will become clear as we proceed



Command line editing

- Suppose we make a mistake when entering the command line and Scilab returns an error message
- Instead of retyping the whole line, we can press the up arrow (\uparrow) on the keyboard to return the line and correct the mistake
- In the shown example the function for the square root, `sqrt()`, was first erroneously typed `sqt()`
- Note that this is just one of several alternatives for command line editing

```
-->a = 2; b = sqt(a)
```

```
!--error 4  
Undefined variable: sqt
```

```
-->a = 2; b = sqrt(a)
```

```
-->a = 2; b = sqrt(a)  
b =
```

```
1.4142136
```

Press up arrow
Correct

Editing demo

- Evaluate the function
 $\log(s^2 - 2s \cdot \cos(\pi/5) + 1)$
for $s = 0.5, 0.95,$ and 1
- Do not rewrite the function, use instead the **up arrow** to edit previous commands!

```
-->s=.5; log(s^2-2*s*cos(%pi/5)+1)
ans =
- 0.8187489
-->s=.95; log(s^2-2*s*cos(%pi/5)+1)
ans =
- 1.006851
-->s=1; log(s^2-2*s*cos(%pi/5)+1)
ans =
- 0.9624237
```



Complex numbers

- Scilab handles complex numbers as easily as real numbers
- The variable `%i` stands for $\sqrt{-1}$
- The first example shows how Scilab evaluates some functions with the complex argument $x = 2 + 3i$
- An imaginary `sin()` argument produces a result!
- The second example shows how Scilab does arithmetic operations with two complex equations, x and y

```
-->x = 2 + 3*%i;
```

```
-->abs(x)  
ans =
```

3.6055513

```
-->real(x)  
ans =
```

2.

```
-->imag(x)  
ans =
```

3.

```
-->sin(x)  
ans =
```

9.1544991 - 4.168907i

```
-->atan( imag(x), real(x) )  
ans =
```

0.9827937

```
-->x = 2 + 3*%i; y = 1 - 1*%i;  
-->z1 = x - y
```

z1 =

1. + 4.i

```
-->z2 = x * y  
z2 =
```

5. + i

```
-->z3 = x / y  
z3 =
```

- 0.5 + 2.5i

Vectorized functions

- Scilab functions are vectorized, meaning that functions can be called with vectorial arguments
- In the shown example, first a column vector called t is created
- Next the vector is used as argument in the sin() function, in the expression for y

- If the values of t are of no interest, the printout can be avoided by putting a semicolon after the expression for t:

$t = [0:5]'; y = \sin(0.2*t)$

- Vectors will be discussed in connection with matrices in Chapter 5

```
-->t = [0:5]'  
t =  
  
0.  
1.  
2.  
3.  
4.  
5.  
  
-->y = sin(0.2*t)  
y =  
  
0.  
0.1986693  
0.3894183  
0.5646425  
0.7173561  
0.8414710
```

Long command lines

- Long command expressions can be divided among two or more lines
- One tool for that purpose is **two or three periods** (...) to indicate that the statement continues
- Long matrix expressions can be written on separate lines by omitting the semicolon that normally ends a row (bottom)

```
-->p=1+2+3+4+5+6+7+8+9+10+11+12+...
-->13+14+15+16+17+18+18+19+21+22+23+24+25
p =
```

323.

```
-->q = 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + ...
-->1/7 + 1/8 + 1/9 + 1/10 + 1/11 + 1/12
q =
```

2.1032107

```
-->A = [1 2 3 4 5
-->6 7 8 9 10
-->11 12 13 14 15]
A =
```

1.	2.	3.	4.	5.
6.	7.	8.	9.	10.
11.	12.	13.	14.	15.

Polynomials

- You run into polynomials e.g. if you use frequency domain (state-space) analysis in control engineering
- Here $s = \%s$ is the **seed** that defines the polynomial of "s". An alternative, often used form of the seed definition is $s = (0, 's')$
- The polynomials can be defined through their root vectors
- Scilab translates the roots to their respective polynomials
- When we divide the num polynomial by the den polynomial, Scilab presents the full polynomial expression

```
-->s=%s;
-->num = poly([0,-1,-2],'s')
num =

$$2 \quad 3$$


$$2s^2 + 3s + s$$


-->den=poly([-0.5,-1.5,-2.5,-3.5],'s')
den =

$$2 \quad 3 \quad 4$$


$$6.5625 + 22s^3 + 21.5s^2 + 8s + s$$


-->fr=num/den
fr =

$$\frac{2 \quad 3}{2 \quad 3 \quad 4}$$


$$\frac{2s^2 + 3s + s}{6.5625 + 22s^3 + 21.5s^2 + 8s + s}$$

```

Roots of polynomials

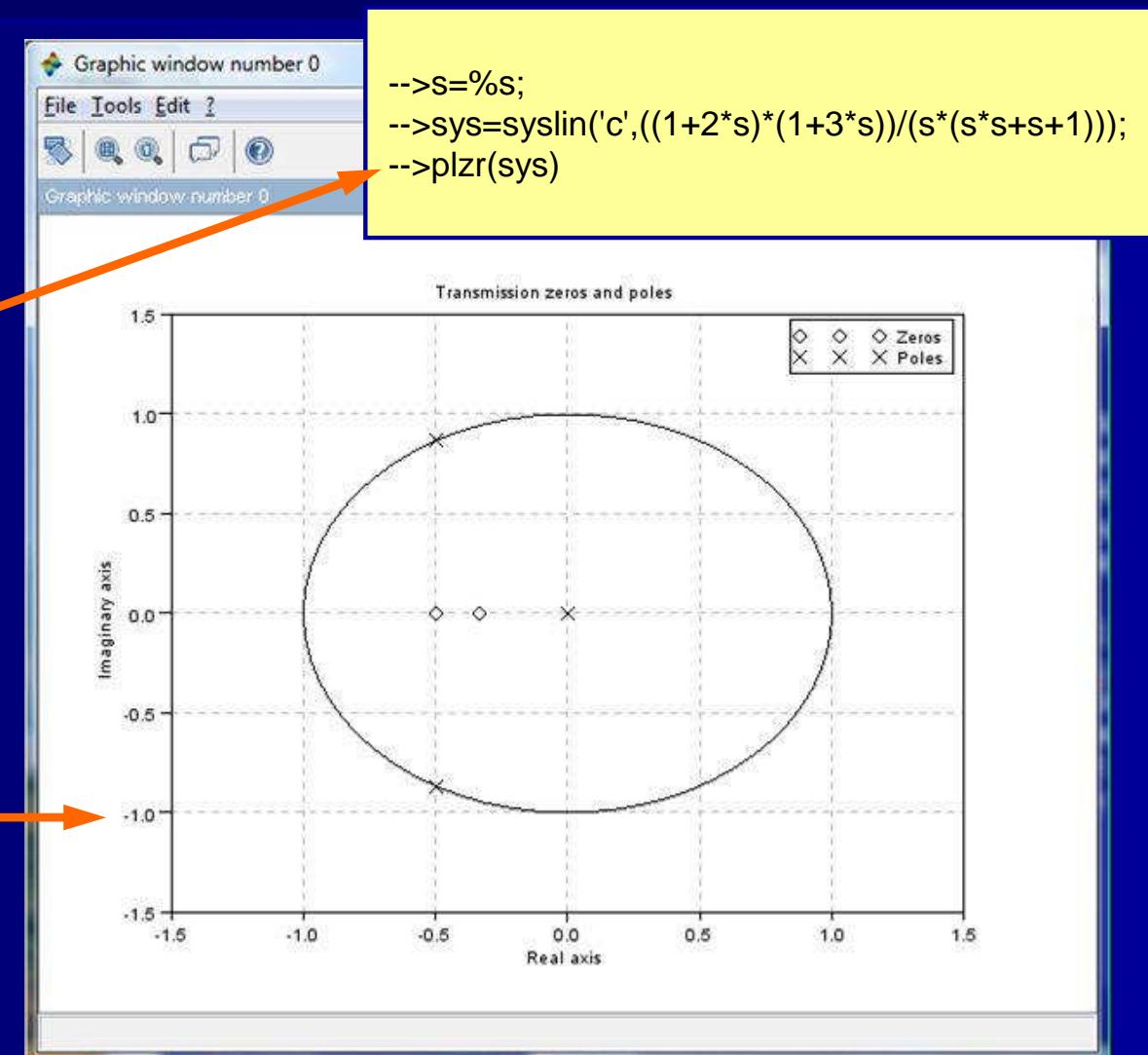
- Determining roots (zeros) of polynomials can be a tedious undertaking
- However, Scilab has a handy tool for the task in the form of the `roots()` function
- To the right the polynomials on the previous slide have been determined
- Note that the seed `s=%s` has to be defined here as well

```
-->s=%s;  
  
-->x=roots(2*s+3*s^2+s^3)  
x =  
  
0  
- 1.  
- 2.
```

```
-->s=%s;  
  
-->z=roots(6.5625+22*s+21.5*s^2+8*s^3+s^4)  
z =  
  
- 0.5  
- 1.5  
- 2.5  
- 3.5
```

Poles and zeros: plzr()

- The plzr() function plots the poles and zeros of a polynomial
- The syslin() function used here will be discussed later
- When pressing Enter after the plzr(sys) command, the Graphics Window opens and displays the plot (The Graphics Window will be discussed in Example 1-1)



Gadgets (1/2): calendar

Among Scilab's built-in gadgets is a calendar. The command

```
calendar()
```

returns the calendar for the present month, the command

```
calendar(y,m)
```

returns the calendar for the year and month in case (shown for June 2013)

```
-->calendar(2013,6)
```

```
ans =
```

```
ans(1)
```

```
Jun 2013
```

```
ans(2)
```

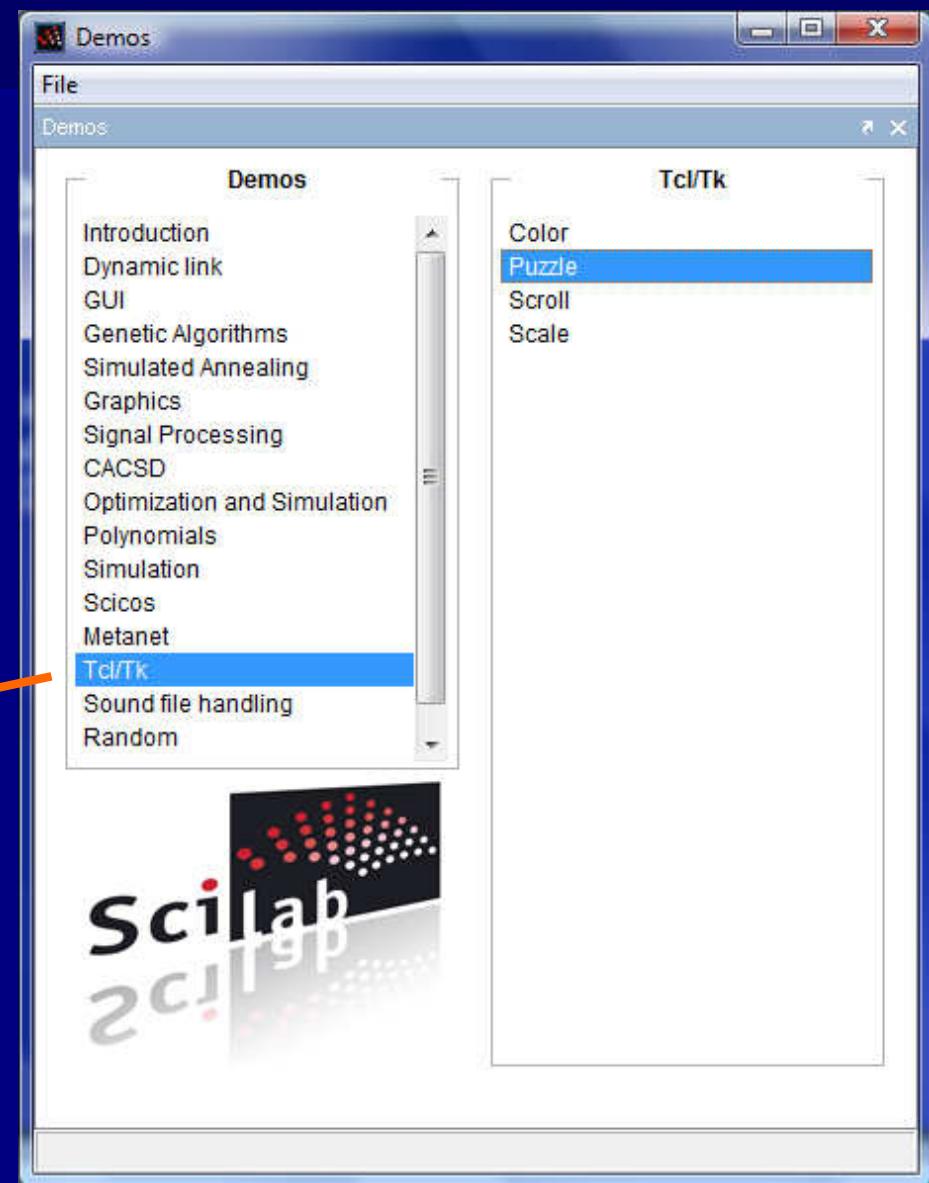
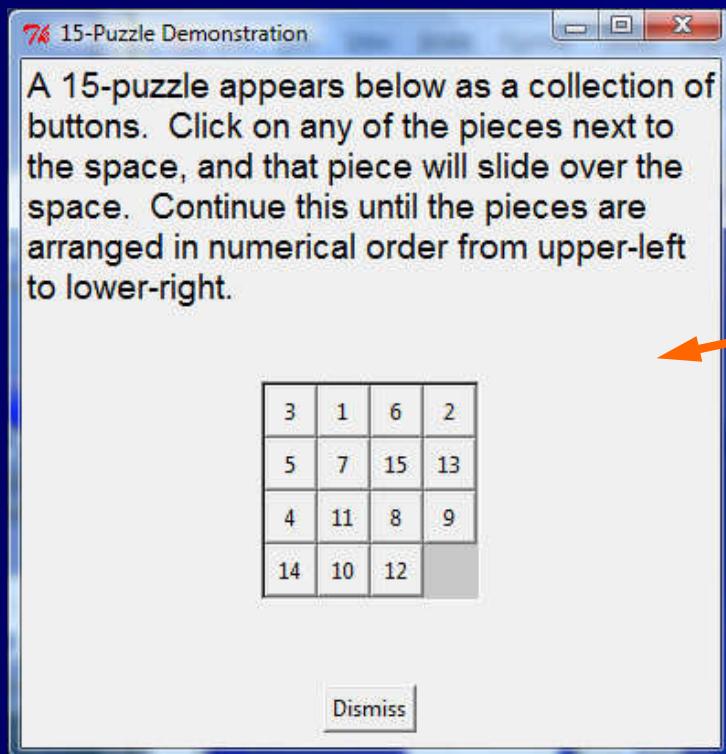
M	Tu	W	Th	F	Sat	Sun
---	----	---	----	---	-----	-----

```
ans(3)
```

0.	0.	0.	0.	0.	1.	2.
3.	4.	5.	6.	7.	8.	9.
10.	11.	12.	13.	14.	15.	16.
17.	18.	19.	20.	21.	22.	23.
24.	25.	26.	27.	28.	29.	30.
0.	0.	0.	0.	0.	0.	0.

Gadgets (2/2): puzzle

Another gadget is a puzzle that can be found under
Demonstrations\Tcl/Tk\Puzzle



Scilab the spy: historymanager

Software that we install on our computers tend to spy on us by collecting information on what we do. Have you ever cared to figure out how much data e.g. Windows' index.dat has stored about your computer & surfing behavior?

Scilab spies with (at least) its history manager. You can access this data by entering displayhistory() on the Console. The file can be cleaned via Preferences\Clear History



```
Scilab Console
File Edit Preferences Control Applications Help
Scilab Console
182 : fibonaccci(2)
183 : fibonaccci(3)
184 : fibonaccci(4)
185 : fibonaccci(1024)
186 : // -- Begin Session : Fri Mar 11 18:51:20 2011 -- //
187 : fibonaccci(32)
188 : exec fibonacci(32)
189 : fibonacci(16)
190 : fibonacci(32)
191 : fibonacci(16)
192 : fibonacci(32)
193 : randomwalk(500)
194 : randwalk(500)
195 : randwalk(1000)
196 : randwalk(500)
197 : randwalk(1000)
198 : help break
199 : k=0;
200 : while q == 1,
201 : k = k+1;
202 : if k > 6 then
203 : break
204 : end;
205 : end
206 : k = 0;
207 : while l == 1,
208 : k = k + 1;
209 : if k > 6 then
210 : break
211 : end;
212 : end
213 : k = 0;
214 : while l == 1,
```

Sorry,
I could not copy-paste an extract because
PowerPoint crashed repeatedly (it happens
to Bill Gates as well... Often.)

4. Examples, Set 1

Demonstration of basic Scilab
programs



[Return to Contents](#)

Example 1-1: script for a simple plot

- Let's elaborate on the example from "Scilab in 15 minutes"
- We work with the Editor using the same script as before, but with added comments
- Save the function when it has been typed in. I call it **plot1.sce** and have saved it on my USB thumb drive, you can save it wherever you like
- To run the script, Click on the Editor's Execute icon
- What happens is shown on the next slide

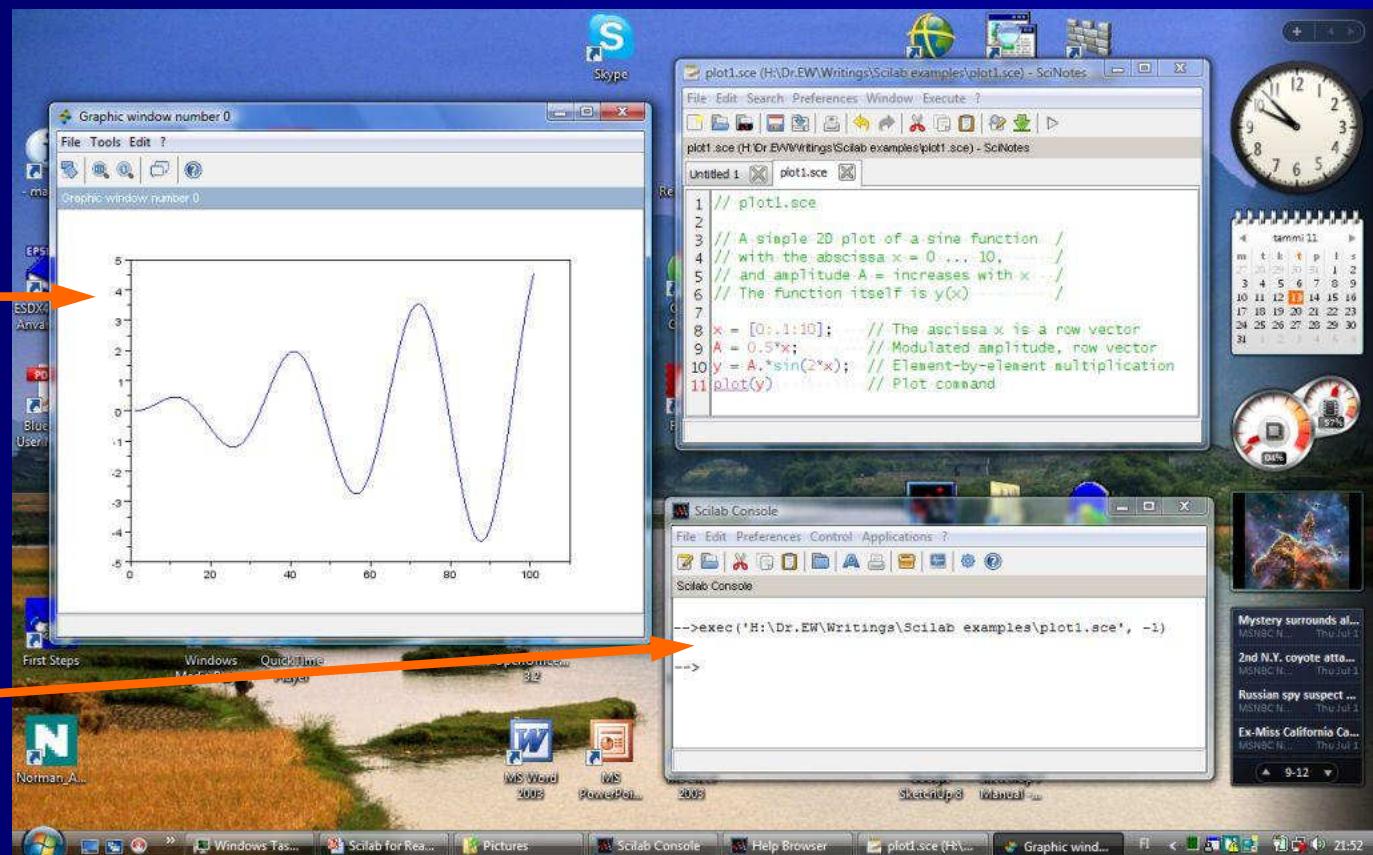
```
// plot1.sce
// A simple 2D plot of a sine function    /
// with the abscissa x = 0 ... 10,        /
// and amplitude A = increases with x    /
// The function itself is y(x)          /
x = [0:.1:10]; // The abscissa x is a row vector
A = 0.5*x; // Modulated amplitude, row vector
y = A.*sin(2*x); // Element-by-element multiplication
plot(y) // Plot command
```

Note: Comments begin with a double slash (//). Scilab disregards everything behind the // when it executes the code

Ex 1-1: the Graphics Window

As seen before, Scilab uses a third window, the **Graphics Window**, to present the plot

Information on the executed script is echoed to the Console. Error messages are also displayed on the Console



These three are the windows that we mainly work with, but there are more. You have already seen a few like the Help Browser and Variable Browser

Ex 1-1: using the Console

- The script could also be executed from the Console
- After the command prompt, type

```
exec plot1.sce
```

- And the result is an **error message**
- The reason? **Scilab looks for plot1.sce in the wrong place**
- To see where Scilab was looking, Click: File\Display current directory
- The answer is shown in the lower window: It looks in Scilab's program file, which is not where I put it

```
-->exec plot1.sce  
!--error 241  
File "plot1.sce" does not exist.
```



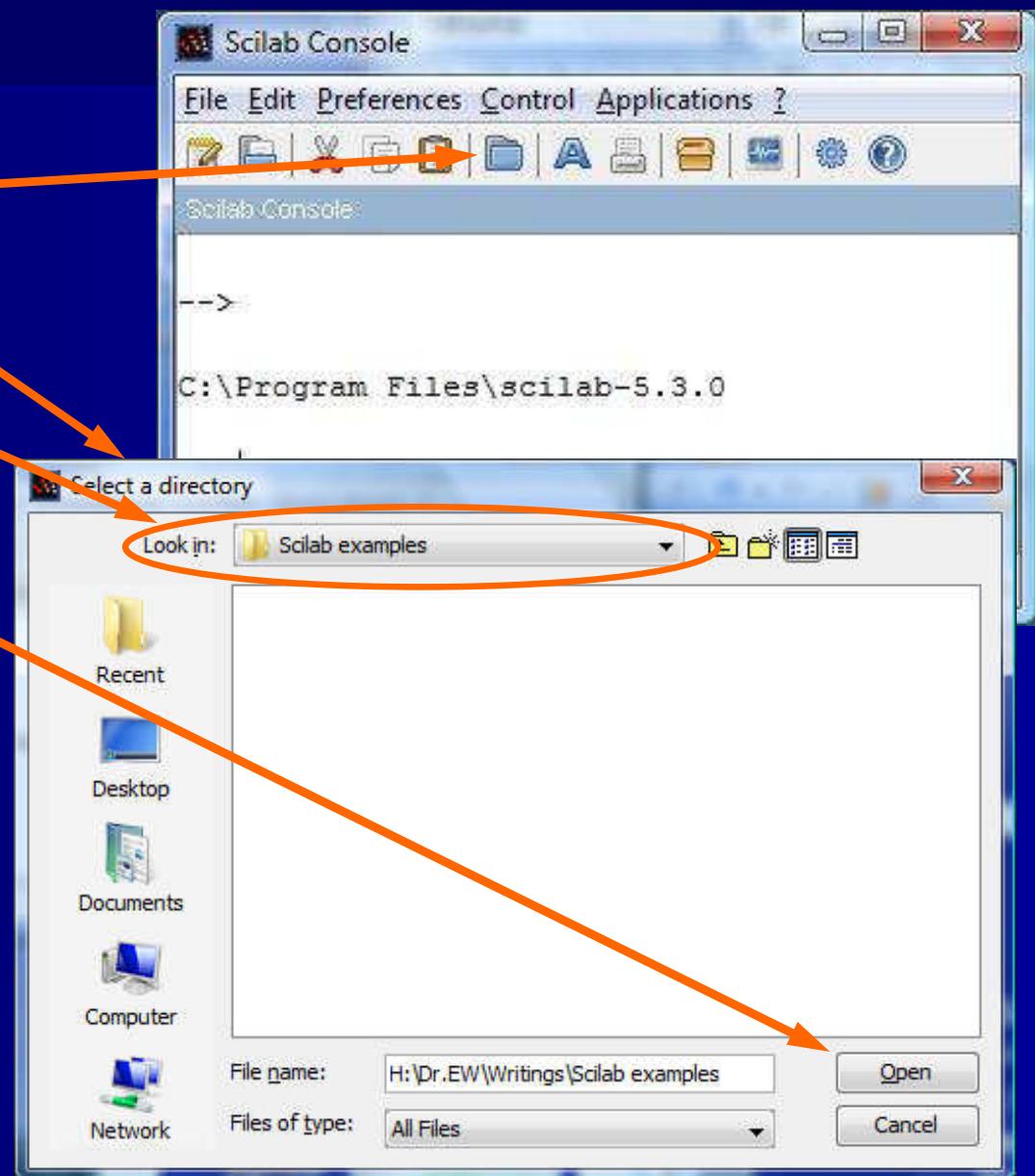
```
-->  
-->  
C:\Program Files\scilab-5.3.0  
-->
```

Ex 1-1: change directory

- Click on the icon Change current directory...
- A new window pops up
- Define the right file with the drop-down menu
- Click: Open
- You can then return to the Console and type in the command

```
exec plot1.sce
```
- And it works, as seen on the next slide

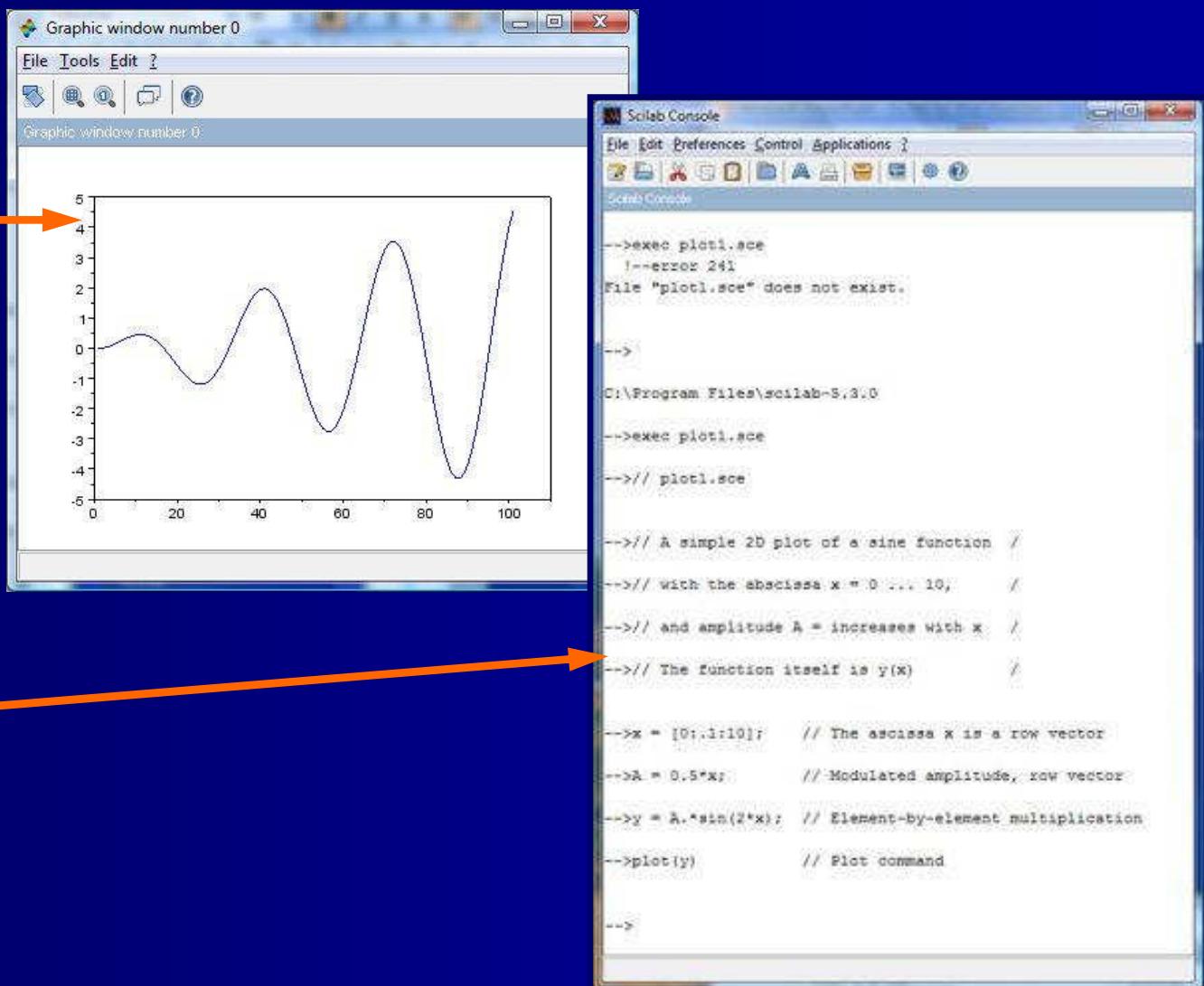
Note: The command `chdir()` allows the directory to be changed "on the run"



Ex 1-1: plot and echo

Up pops the Graphics Window with a plot of the defined function...

while the script is echoed to the Command Window (Console)



Ex 1-1: comments (1/4), command details

Editor contents will from now on be shown on light green background

- The vector definition $x=[0:0.1:10]$ can be interpreted as “from 0 to 10 in steps of 0.1”
- Multiplication by the Dot Operator ($.*$) is necessary to tell Scilab it should multiply the vectors element-by-element. Change to ordinary multiplication ($*$) and you’ll get this error message on the Console

```
// plot1.sce
// A simple 2D plot of a sine function /
// with the abscissa x = 0 ... 10, /
// and amplitude A = increases with x /
// The function itself is y(x) /
x = [0:.1:10]; // The abscissa x is a row vector
A = 0.5*x; // Modulated amplitude, row vector
y = A.*sin(2*x); // Element-by-element multiplication
plot(y) // Plot command
```

```
-->exec('H:\Dr.EW\Writings\Scilab examples\plot1.sce', -1)
y = A*sin(2*x); // Element-by-element multiplication
!--error 10
```

Inconsistent multiplication.

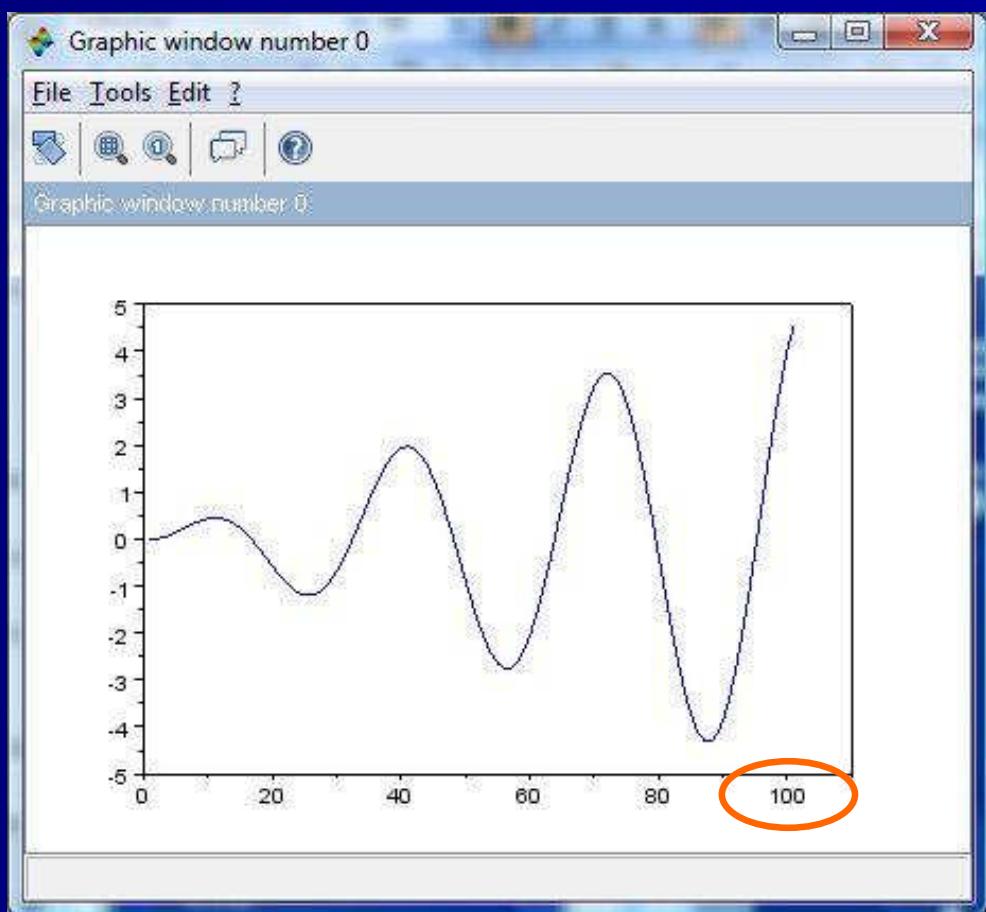
at line 10 of exec file called by :
exec('H:\Dr.EW\Writings\Scilab examples\plot1.sce', -1)

```
-->
```

Ex 1-1: comments (2/3), the plot

The plot is very basic as it has
no title, axis labels, or grid.
We'll return to them in the
next example

The abscissa scale may seem
strange, the maximum value
for x was 10 but the scale
goes to 100. The figure 100 is
actually the number of
calculations, since they were
made in steps of 0.1 up to 10.
Try to change t to
`x=[0:0.2:10];` and you'll
see that the scale ends at 50
(the modified script must be
saved before it can be run)

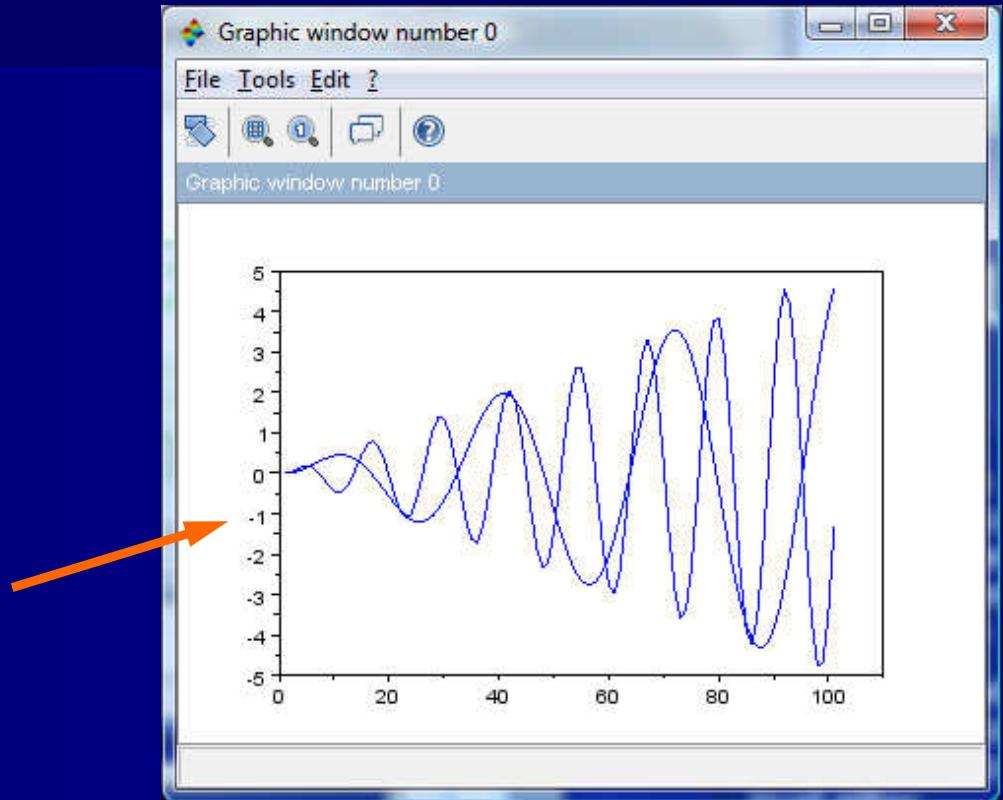


Ex 1-1: comments (3/4), clf

Assume that we make a change to the script, e.g. increase the frequency to $\sin(5*s)$, save it, and execute it immediately after a previous run

As a result Scilab plots the new graph on top of the previous one. To avoid this we must either

- Close the Graphics Window manually after each run, or
- Add the function `clf` (clear figure) to the script to make Scilab clean the window



clf;
x = [0:.1:10];
A = 0.5*x;
y = A.*sin(5*x);
plot(y)

An orange arrow points from the text "To avoid this we must either" to a green box containing the Scilab script. The first line, `clf;`, is circled in orange to highlight it.

Ex 1-1: comments (4/4), cleaning trash

Some programmers prefer to safeguard against different forms of **old junk that may interfere with the execution** of the script. To do this, three commands are added at the beginning of the script:

- **clear**, removes items from the workspace and frees memory*
- **clc**, cleans the Console; the echo signal is mainly erased
- **clf**, wipes an open Graphics Window

```
// plot1.sce  
  
// A simple 2D plot of a sine function /  
// with the abscissa x = 0 ... 10, /  
// and amplitude A = increases with x /  
// The function itself is y(x) /  
  
clear, clc, clf;  
x = [0..1.10]; // The abscissa x is a row vector  
A = 0.5*x; // Modulated amplitude, row vector  
y = A.*sin(2*x); // Element-by-element multiplication  
plot(y) // Plot command
```

Thus our final script looks like this. Pay attention to the semicolon (;) at the end of each expression apart from the last

*) Careful with clear, it may cause havoc in some cases (there will be a demo on this later)

Example 1-2: the task, a decaying linear chirp

- Write the script for a linearly frequency modulated sinusoidal signal $s(t)$, i.e. a **linear chirp** of the type

$$s(t) = A(t) \cdot \sin \{ [2\pi(f_0 + k(t)t] + \phi \}$$

where k is the rate of frequency change, or chirp rate

- Use 2.5 periods of the basic frequency
- The amplitude should decay exponentially, $A(t) = 2e^{-t/3}$
- The initial phase shift ϕ shall be $\pi/4$
- Plot and print the result with a plotting method that differs from the previous one
- The plot shall have grid, title, and axis labels

Plug in the commands `plot()`, `histplot()`, `surf()`, and `plot3d()` on the Console to view examples of Scilab plots. See also [Chapter 7](#).

Ex 1-2: first iteration

- The linspace() function creates a linearly space plotting vector with the arguments from, to, number of points. The default value is 100 points, but more are needed here
- Here is the Dot Operator (.*) again
- The plot2d() produces the 2D plot. The arguments t and s stands for the x and y-axes, the number 5 produces a red graph

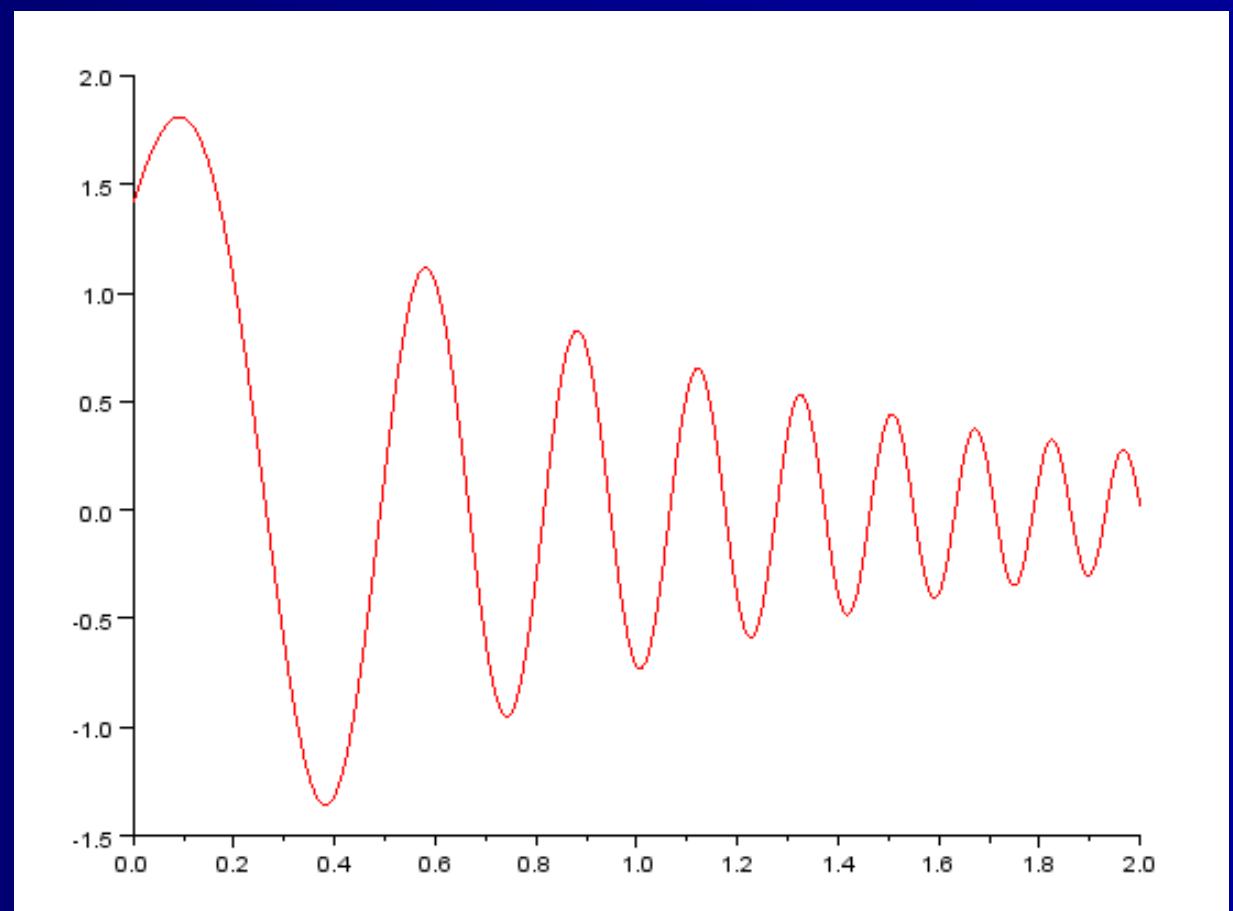
```
// f-modulation1.sce //  
// Plots a sinusoidal function of the type //  
// s = A(t)(sin(wt+x(t)+phi)), where w = angular //  
// velocity, x(t) = frequency modulation, phi = //  
// phase shift, and A(t) = amplitude //  
  
clear, clc, clf;  
f = 1; // Frequency  
w = 2*%pi*f;  
phi = %pi/4; // Initial phase shift  
fin = (4*%pi)/w; // End of plot  
t = linspace(0,fin,1000);  
A = 2*exp(-t);  
s = A.*sin(w*t + 10*t^2 + phi);  
plot2d(t,s,5)
```

Note: fin is used as the end of plot variable name because end is reserved (Scilab keyword)

Ex 1-2: plot

The plot looks as expected—including the initial phase shift—but it lacks a grid, title, and axis labels

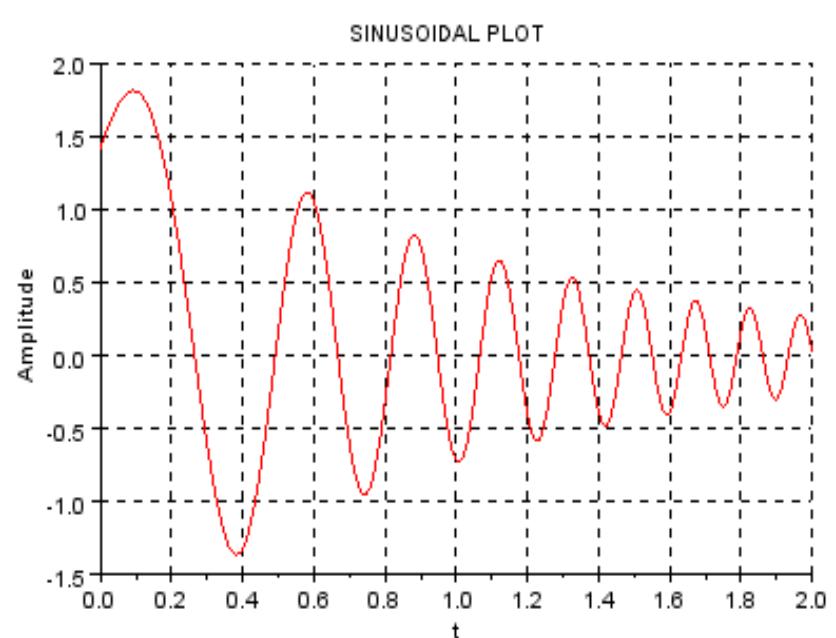
`plot2d()` is a more versatile function than `plot()`, which is similar to the `plot` function in Matlab



Ex 1-2: improved plot

Here I have added code to plot the grid, xgrid(), title, xtitle(), and x and y labels, xlabel(), ylabel(). Crude, but it works

```
// f-modulation2.sce  
  
// Plots a sinusoidal function of the type          /  
// s = A(t)(sin(wt+x(t)+phi)), where w = angular /  
// velocity, x(t) = frequency modulation, phi =      /  
// phase shift, and A(t) = amplitude                /  
  
clear, clc, clf;  
f = 1;                                // Freq  
w = 2*%pi*f;                          // Init  
phi = %pi/4;                           // End  
fin = (4*%pi)/w;  
t = linspace(0,fin,1000);  
A = 2*exp(-t);  
s = A.*sin(w*t + 10*t^2 + phi);  
plot2d(t,s,5)  
  
xgrid()  
xtitle('SINUSOIDAL PLOT')  
xlabel('t')  
ylabel('Amplitude')
```

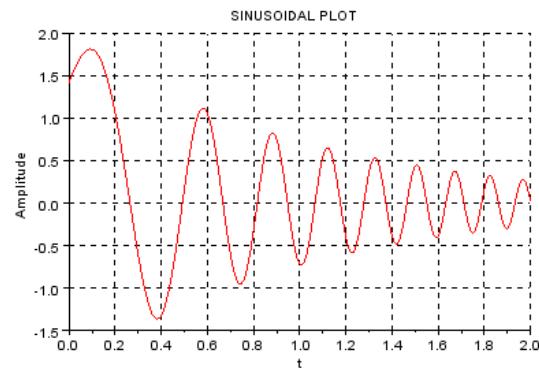


Ex 1-2: printing

- Scilab's windows (Console, Editor, Graphics Window) all have both normal and advanced print functions
- One way of getting a coherent printable document is to copy the contents of the windows and paste them into a word processing page (Scilab supports **LaTeX**)
- The image shown here was done on MS Word (OOo Writer did not recognize Scilab's file type). It was then printed as .PDF, saved as a .PNG file, and finally cropped with MS Picture Manager
- That's a tedious method. Consult Help for advanced print features

DECAYING LINEAR CHIRP IN SCILAB

```
// f-modulation2.sce //  
  
// Plots a sinusoidal function of the type //  
// s = A(t)(sin(wt+x(t)+phi)), where w = angular //  
// velocity, x(t) = frequency modulation, phi = //  
// phase shift, and A(t) = amplitude //  
  
clear, clc, clf;  
f = 1; // Frequency  
w = 2*%pi*f;  
phi = %pi/4; // Initial phase shift  
fin = (4 *%pi)/w; // End of plot  
t = linspace(0,fin,1000);  
A = 2*exp(-t);  
s = A.*sin(w*t + 10*t^2 + phi);  
plot2d(t,s,5)  
xgrid0  
xtitle('SINUSOIDAL PLOT')  
xlabel('t')  
ylabel('Amplitude')
```



Ex 1-2: checking

- To show that the frequency is linearly modulated, we can add frequency as a function of t to the plot
- For that we add the function f_mom to the script
- The plot command must also be modified. We
 - shift back to the `plot()` command and include both parameters, together with color information ('r', 'b')
 - fuse x-label 't' as an argument of `xtitle()`
 - swap y-label for `legend()`; the argument 2 refers to the upper left hand corner

```
// f-modulation3.sce
// Plots a sinusoidal function of the type
// s = A(t)(sin(wt+x(t)+phi)), where w = angular
// velocity, x(t) = frequency modulation, phi =
// phase shift, and A(t) = amplitude. Second
// plot for momentary frequency values

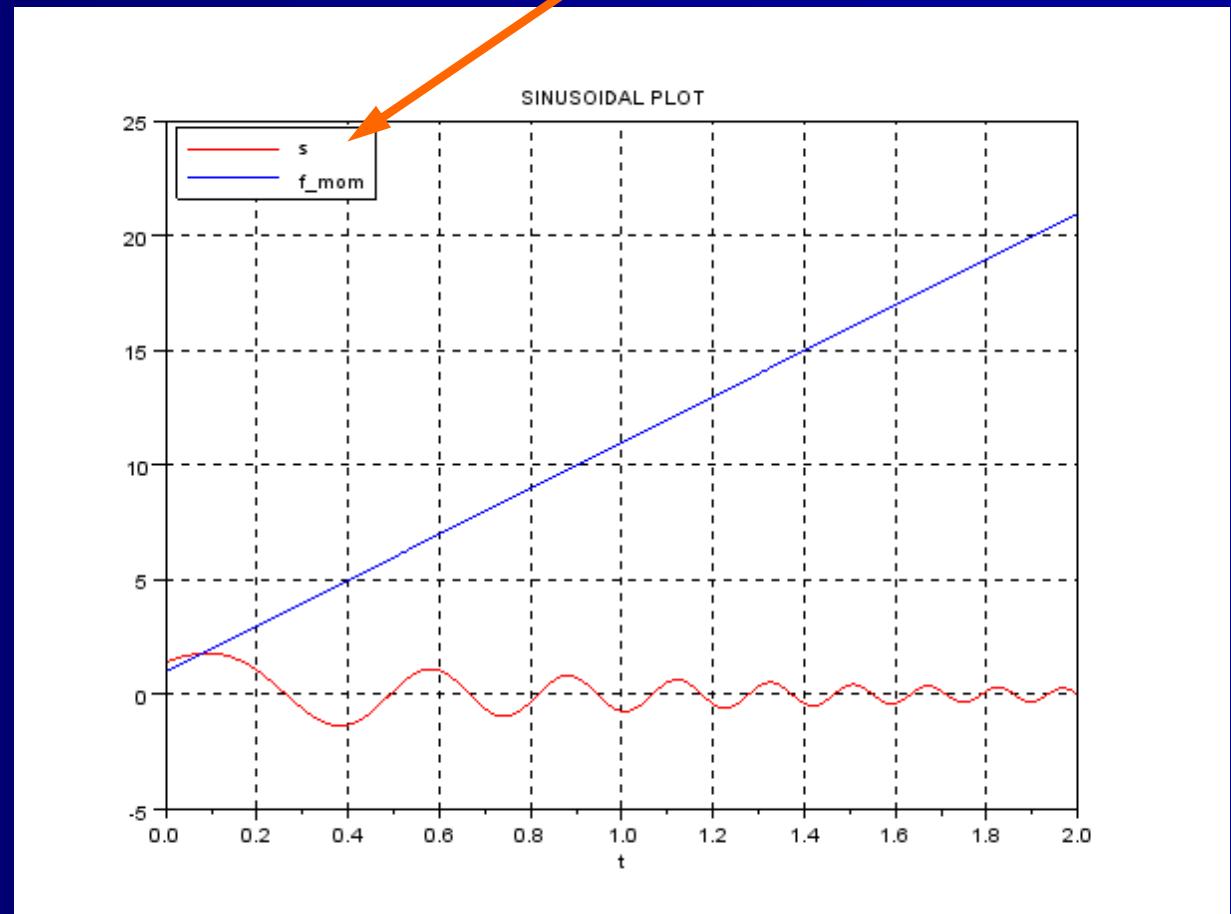
clear, clc, clf;
f = 1;                                // Frequency
w = 2*%pi*f;
phi = %pi/4;                           // Initial phase shift
fin = (4*%pi)/w;                      // End of plot
t = linspace(0,fin,1000);
A = 2*exp(-t);
s = A.*sin(w*t + 10*t^2 + phi);
f_mom = f + 10*t;                     // Momentary frequency
plot(t,s,'r',t,f_mom,'b')
xgrid()
xtitle('SINUSOIDAL PLOT','t')
legend('s','f_mom',2)
```

Ex 1-2: final plot

OK, not an optimal plot but the information is there.

With the big differences in vertical scales, we should either use logarithmic y axis or separate the two into **subplots**—but that comes later

Pay attention to the legend



Ex 1-2: discussion

- As was said earlier, Scilab evolves with time and approaches Matlab with each release
- As an example in case, Scilab's Help Browser recognizes the xlabel() and ylabel() that I used in the improved plot as **Matlab functions** and also refers to them as Scilab functions
- However, there are plenty of **obsolete Scilab functions** and you find them all over if you rely on old tutorials. Even Scilab's Help Browser may refer to them
- Be careful, particularly if a function name begins with x- ([cf. note in Chapter 7](#))
- You may have noticed that I begin the script with a **comment stating the name of the script** (e.g. // f-modulation3.sce /). I do this to help identify the script when I am looking at a printout

Example 1-3: Lotto, the task

The first part of this example is borrowed from Mäkelä's tutorial

Task 1: Create a user defined **function** (UDF) that draws a row of Lotto numbers. Assume that the Lotto row contains 7 numbers, 1-39

Task 2: Write a **script** that calls the previous function (or a modification of it, if necessary) and produces a plot of it to visually indicate if the function produces random numbers. Generate 10,000 draws for the task



Ex 1-3: task 1, script

- `dt=getdate()` returns `dd-mm-yyyy`
- `rand('seed',n)` sets the random generator seed to n
- `dt(9)` returns a number between 00 and 59, `dt(10)` returns milliseconds 000...999
- The while...end construct will be covered under the discussion below

Function ID, not a comment

```
function lotto
//-----
// The function draws 7 Lotto numbers [1,39] by first
// creating a seed using current date and time
// (second, millisecond) information
//
dt=getdate();           // Pick current date
rand('seed',1000*dt(9)+dt(10)); // Initialize random generator
numbers=floor(1+39*rand(1,7)); // Draw Lotto row
while(length(unique(numbers))<7) // If number repeats in row,
    numbers=floor(1+39*rand(1,7)); // then draw a new row
end
numbers=gsort(numbers); // Sort numbers in decreasing order
disp(numbers(7:-1:1)); // Display in increasing order
endfunction
```

Why the hassle with the seed? Without it Scilab generates the same sequence for each session. The `1000*t(9)+ dt(10)` argument improves randomness.

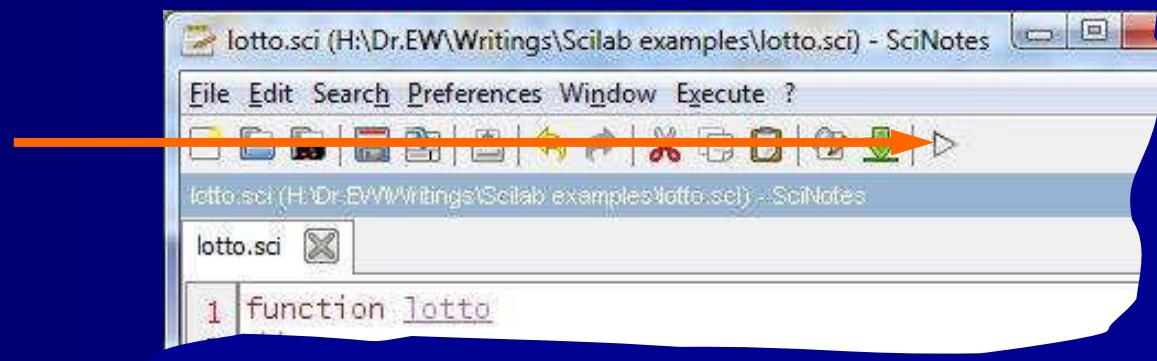
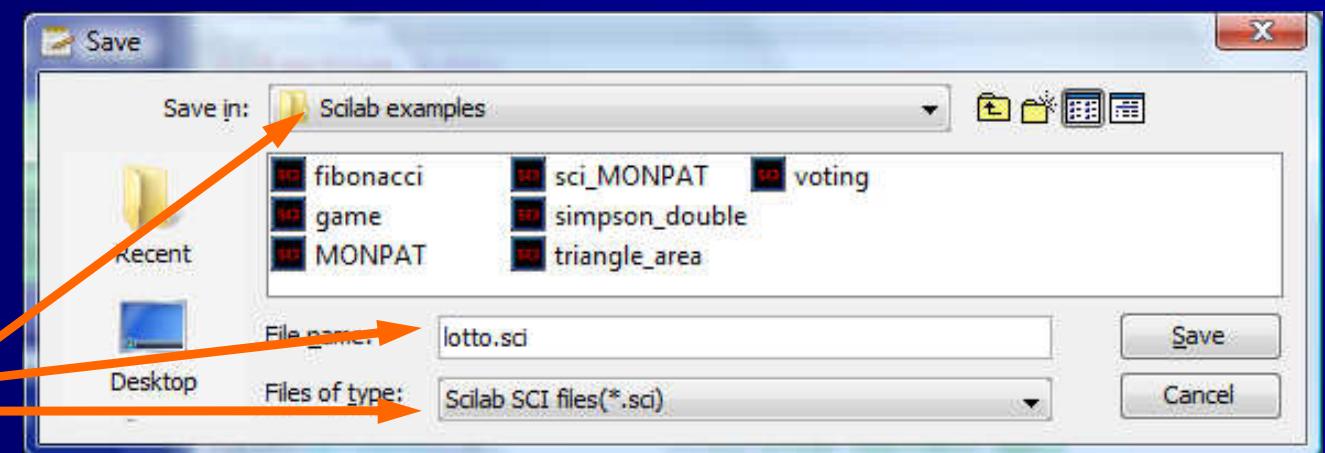
Ex 1-3: task 1, saving

This script (function) differs a bit from the earlier ones, so let's go through the save operation:

Save the script as lotto.sci in your preferred file

Next, Click on the Execute icon of the Editor to load the saved file into Scilab

Continues on next slide...



Ex 1-3: task 1, running

If the Console shows a **warning**, check with Help what it means. It can be ignored or the `funcprot(0)` command can be added to the script to avoid the warning. You can also jump to [Chapter 18](#) for a brief explanation

Execute (run) the loaded function by entering the function name on the Console

```
-->exec('H:\Dr.EW\Writings\Scilab examples\lotto.sci', -1)
Warning : redefining function: lotto
    . Use funcprot(0) to avoid this message
```

```
-->
-->exec('H:\Dr.EW\Writings\Scilab examples\lotto.sci', -1)
Warning : redefining function: lotto
    . Use funcprot(0) to avoid this message
-->help funcprot
-->lotto
3.  5.  13.  15.  33.  37.  39.
```

And the winning numbers are...

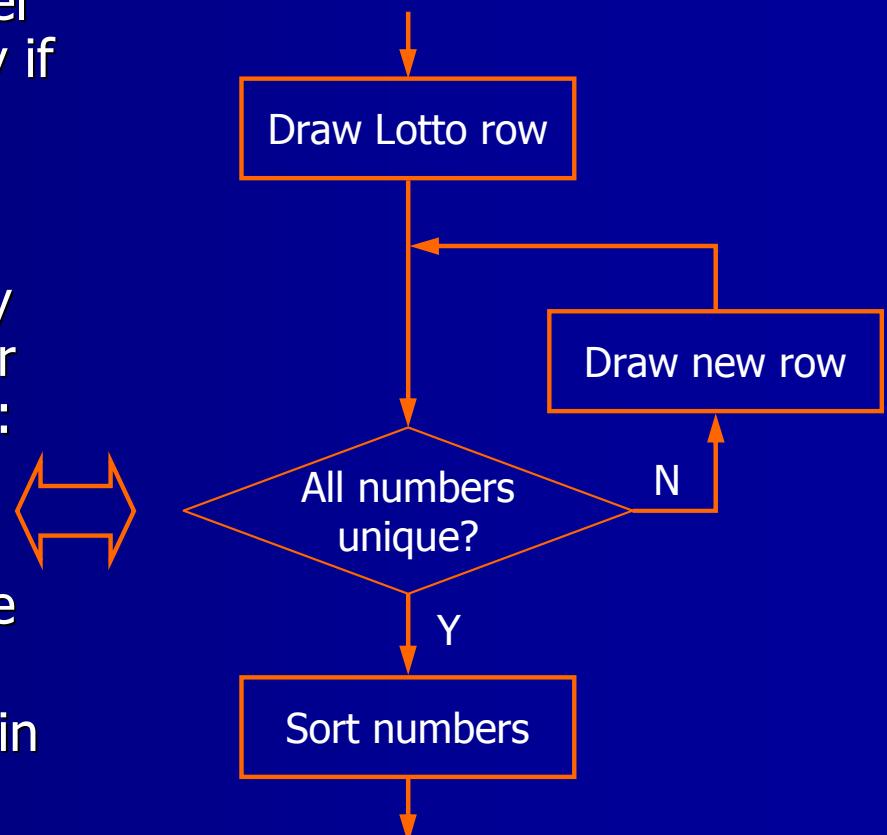
Ex 1-3: task 1, discussion

- This is already an intermediate level programming exercise. Don't worry if it gives you problems. Most of its details will be repeated later
- The flowchart of the while...end construct is shown to the right. Pay attention to the elegant solution for testing uniqueness of the numbers:

`length(unique(numbers)) < 7`

However, in theory it could become an almost infinite loop...

- We'll return to while ... end loops in Chapter 11



Ex 1-3: task 2, script (1/2)

The previous UDF must be modified if it is called by a separate code: 1) Delete sorting and display and 2) redefine the function ID to allow calling

In the latter case it has one or more **input arguments** (in) that are given to it by the calling command and **output arguments** [out] by which it returns the result of its calculations to the calling command (see next slide)

```
// lotto2.sce
//-----
// The script asks for the number of Lotto draws that we
// wish to do, using a separate dialog box. It then calls
// the local UDF lottodraw() that generates a row of N
// random Lotto numbers in the range [1,39]. It sorts the
// numbers into a vector by adding one (1) to the relevant
// vector element for each corresponding hit. The result
// is plotted after the entered number of draws.
//-----

clear,clc,clf;

// (SUBROUTINE) function lottodraw():
//-----
// The function draws N Lotto numbers [1,39], with
// N being defined through the input argument in.
// It delivers the drawn row to the calling script
// command through the output argument out. The
// randomness of the drawn numbers is improved by
// first creating a seed using current date and
// time (second, millisecond) information.
```

Ex 1-3: task 2, script (2/2)

Redefined function
(subroutine)

The number of Lotto
draws that we are
looking for is entered
via a separate dialog
box x_dialog()

The drawn Lotto
numbers are collected
in the columns vector
inside the for ... end
loop

The result is plotted as
step functions

```
function out=lottodraw(in)
dt=getdate(); // Pick current date
rand('seed',1000*dt(9)+dt(10)); // Initialize random generator
out = floor(1+39*rand(1,in)); // Draw Lotto row (out variable)
while(length(unique(out))<in) // If number repeats in row,
    out = floor(1+39*rand(1,in)); // then a new row is drawn
end
endfunction

// (MAIN) Call subroutine, update histogram, plot:
//-----
M = evstr(x_dialog('Enter # of...
lotto draws ',''));
N = 7;                                // Lotto numbers to draw
columns = zeros(1,39);                  // Initiate collecting vector
for k = 1:M
    numbers = lottodraw(N);             // Call to subroutine
    columns(numbers)=columns(numbers)+1; // Add 1 for drawn number
end

x = linspace(1,39,39);                  // Define x axis
plot2d2(x,columns,style=2)              // Plot as step functions
xtitle('RESULT OF LOTTO DRAWS')        // Add title & labels
xlabel('Lotto numbers [1,39]')
ylabel('Hits')
```

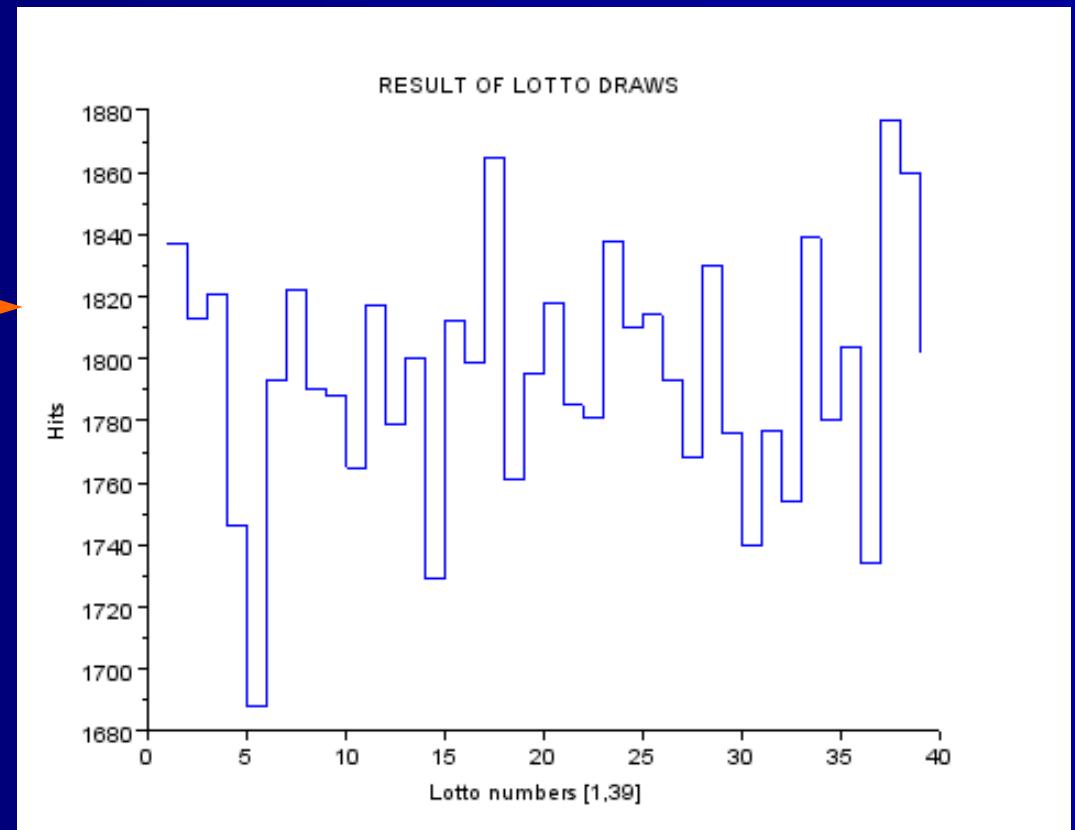
Ex 1-3: task 2, execution & plot



The dialog box pops up when executing the script. Enter the wanted number of Lotto draws and Click OK

The result is plotted on the Graphics Window. It is not too bad, considering that the average of 10,000 draws is $7 \times 10,000 / 39 = 1,795$

It takes my 1.6 GHz dual core processor about 10 seconds to compute 10,000 draws



Ex 1-3: comments (1/3)

- This was not exactly an engineering problem, but it showed many features of Scilab
- The UDF in Task 1 is unusual in being closed, having no input or output arguments—you just use it as it is. The local UDF demonstrated in Task 2 is the normal case
- In addition to rand(), Task 1 brings in several useful functions: getdate(), floor(), unique(), and gsort()
- The script in Task 2 is commented at length. Adding headings and comments takes time and they require space, but **comments are absolutely necessary** to understand the program at a later date
- Task 2 introduces the dialog box, a GUI (graphical user interface) feature to which we shall return in Chapter 15

Ex 1-3: comments (2/3)

- In addition to the `plot()` and `plot2d()` commands that we used, Scilab has numerous other ways of creating plots, together with options for adding clarifying **text strings** to the plots. Plotting will be covered in more detail in Chapter 7
- **Flow control**—in this case the term refers to the use of conditional branch structures—will be discussed in Chapter 11
- Examples 1-1 ... 1-3 were also intended to stress the fact that we are forced to “think matrix-wise” when working with Scilab. For instance, Scilab immediately generates an error message if we attempt to do ordinary multiplication (*) when a parameter is in matrix form and requires Dot multiplication (.*.) (Recall Example 1-1?)

Ex 1-3: comments (3/3), rounding functions

The rounding function `floor()` is one of four rounding functions in Scilab: `round()`, `fix()` (or `int()`), `floor()`, and `ceil()`

Pay attention to the difference between the first and the two last ones

<code>round()</code>	rounds to nearest integer
<code>fix()</code> or <code>int()</code>	returns integer part
<code>floor()</code>	rounds down
<code>ceil()</code>	rounds up

```
-->round(-2.7), round(2.7)  
ans =
```

```
- 3.  
ans =
```

```
3.
```

```
-->fix(-2.7), fix(2.7)  
ans =
```

```
- 2.  
ans =
```

```
2.
```

```
-->floor(-2.7), floor(2.7)  
ans =
```

```
- 3.  
ans =
```

```
2.
```

```
-->ceil(-2.7), ceil(2.7)  
ans =
```

```
- 2.  
ans =
```

```
3.
```

5. Matrices, functions & operators

An overview of basic matrix operations, functions, and operators



[Return to Contents](#)

Introduction

- As Scilab is built around matrices we are forced to use them
- Scilab stores numbers (and characters) in matrices
- A matrix can be seen as a table, consisting of m rows and n columns ($m \times n$ matrices, also denoted $i \times j$ matrices)
- **Scalar variables** do not exist *per se*, they are treated as 1×1 matrices
- The general form of a Scilab matrix (here 3×3 matrix) is

$A = [11 \ 12 \ 13; \ 21 \ 22 \ 23; \ 31 \ 32 \ 33]$

Row elements can also be separated by **commas**:

$A = [11, \ 12, \ 13; \ 21, \ 22, \ 23; \ 31, \ 32, \ 33]$

In both cases **semicolons** separate rows

- The next page shows both alternatives for the 3×3 matrix

“[The vector] has never been of the slightest use to any creature.”

Attributed to Lord Kelvin

The 3x3 matrix

Both alternatives for expressing matrices are interpreted in the same way by Scilab. Pick whichever you like

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

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

Note: Scilab may cause a copied screen text (as seen here) to be underlined when pasted to another document. If so, put the cursor at the end of the text and press Backspace (\leftarrow)

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

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

Row and column vectors

Task 1: Create a row vector with first element 0, last element 1 and increment (step size) 0.2. Note the order and colons that divide elements



```
-->row=[0:0.2:1]
row =
0. 0.2 0.4 0.6 0.8 1.
```

Task 2: Create a similar column vector. Note the asterisk that signifies the matrix transpose



```
-->column=[0:0.2:1]'
column =
0.
0.2
0.4
0.6
0.8
1.
```

In case the Console window is set too small and all elements do not fit in, Scilab interrupts plotting and asks if it should continue



Some special matrices

3x3 identity matrix



```
-->C=eye(3,3)  
C =  
  
    1.  0.  0.  
    0.  1.  0.  
    0.  0.  1.
```

3x2 matrix of ones



```
-->D=ones(3,2)  
D =  
  
    1.  1.  
    1.  1.  
    1.  1.
```

2x3 zero matrix



```
-->E=zeros(2,3)  
E =  
  
    0.  0.  0.  
    0.  0.  0.
```

The function `rand(m,n)` creates a **uniformly distributed** mxn matrix. Adding the argument 'normal' creates a **normal distributed** matrix

```
-->rand(4,4)  
ans =
```

0.2312237	0.3076091	0.3616361	0.3321719
0.2164633	0.9329616	0.2922267	0.5935095
0.8833888	0.2146008	0.5664249	0.5015342
0.6525135	0.312642	0.4826472	0.4368588

```
-->rand(4,4,'normal')  
ans =
```

- 1.3772844	- 0.6019869	- 0.3888655	- 0.7004486
0.7915156	- 0.0239455	- 0.6594738	0.3353388
- 0.1728369	- 1.5619521	0.6543045	- 0.8262233
0.7629083	- 0.5637165	- 0.6773066	0.4694334

Matrices are defined with square brackets, [], while parentheses, (), are used to cluster function arguments

Basic matrix calculations

```
-->A = [1 2 3; 4 5 6]; B = A; C = A + B  
C =
```

```
2. 4. 6.  
8. 10. 12.
```

Addition

```
-->A=[1 2 3; 4 5 6]; B=[A]; C=A/B  
C =
```

```
1. 1.518D-16  
3.795D-15 1.
```

Division (note rounding errors)

```
-->A = [1 2 3; 4 5 6]; B = A'; C = A * B  
C =
```

```
14. 32.  
32. 77.
```

Multiplication (note transpose!)

```
-->A = [2 3; 4 5]; H = inv(A)  
H =
```

```
- 2.5 1.5  
2. - 1.
```

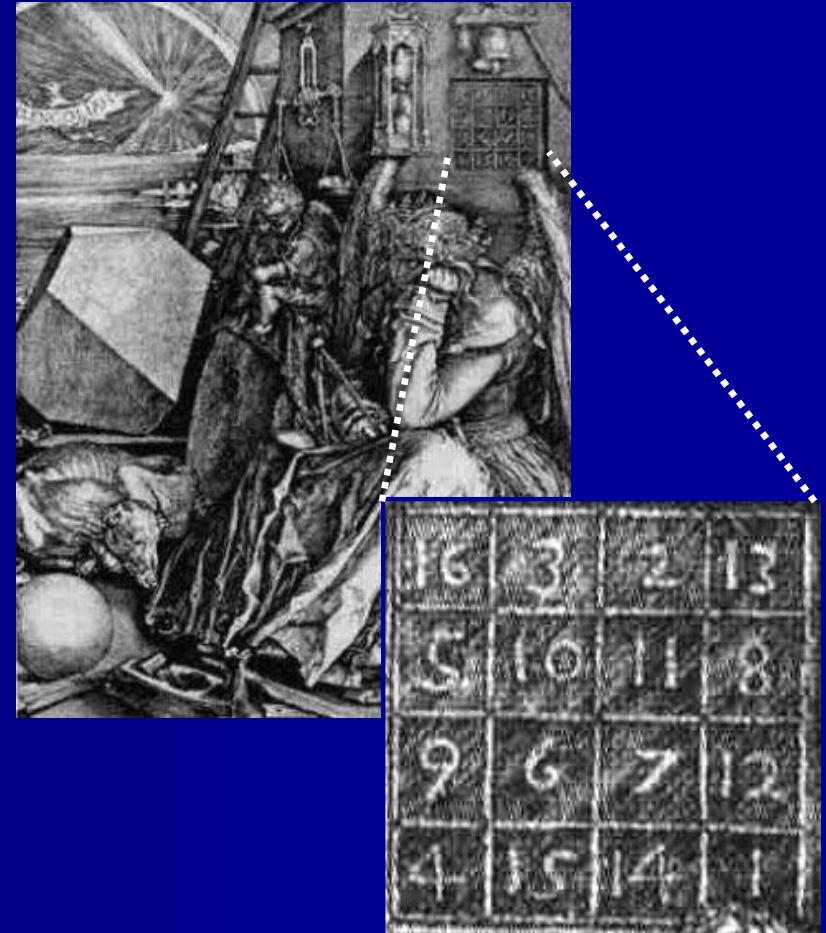
Inverse matrix

Note 1: Rules for matrix operations must of course be observed!

Note 2: Scilab returns D, not e, for the exponent (1.518D-16); the exact value is 0 but here we have a case of limited computing accuracy

Dürer's magic square

- German Renaissance artist and amateur mathematician Albrecht Dürer's "magic" square is a popular example in linear algebra
- In the window of Dürer's engraving the sum of any row, column, or diagonal yield the same result (34)
- We shall use the magic square to investigate some aspects of matrix operations
- The magic square will be denoted "M" to set it apart from other matrices
- Note that many matrix operations are defined only for square matrices



sum(), transpose, and diag()

- The magic square is entered in the Console's command line

- The statement `sum(M)` produces the sum of all elements. This **differs from Matlab**, where the same statement returns the sum of the four columns, i.e.,

$$\text{sum}(M) = 34. \quad 34. \quad 34. \quad 34.$$

- The transpose statement `M'` flips the matrix about its main diagonal
- The statement `diag(M)`, finally, returns the main diagonal as a column vector

```
-->M = [16 3 2 13; 5 10 11 8  
-->9 6 7 12; 4 15 14 1]
```

M =

```
16. 3. 2. 13.  
5. 10. 11. 8.  
9. 6. 7. 12.  
4. 15. 14. 1.
```

```
-->sum(M)
```

ans =

```
136.
```

```
-->M'  
ans =  
16. 5. 9. 4.  
3. 10. 6. 15.  
2. 11. 7. 14.  
13. 8. 12. 1.
```

```
-->diag(M)
```

ans =

```
16.  
10.  
7.  
1.
```

Sum of rows and columns: sum()

- Scilab returns the sums of rows and columns of a matrix A with the commands `sum(A,'c')` and `sum(A,'r')` respectively
- At first sight the use of '`c'`' and '`r'` arguments feels odd. The logic is that '`r'` returns the sums of matrix columns giving a **row vector**, while '`c'` returns the sums of matrix rows, a **column vector**
- Alternative statements are:
`sum(A,'r')` = `sum(A,1)` and
`sum(A,'c')` = `sum(A,2)`

```
-->A = [1 2 3; 4 5 6; 7 8 9]
```

```
A =
```

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

```
-->B = sum(A,'r')
```

```
B =
```

```
12. 15. 18.
```

```
-->C = sum(A,'c')
```

```
C =
```

```
6.  
15.  
24.
```

prod()

- The product of rows and columns can be formed in a similar way as sums
- **prod(A, 'r')** returns the product of each column as a row vector
- **prod(A, 'c')** returns the product of each row as a column vector
- **prod(A)** returns the product of all matrix elements

```
-->A=[1 2 3; 4 5 6; 7 8 9]  
A =
```

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

```
-->prod(A, 'r')  
ans =
```

28. 80. 162.

```
-->prod(A, 'c')  
ans =
```

6.
120.
504.

```
-->prod(A)  
ans =
```

362880.

min(), max()

- The same logic continues with the min() and max() functions
- min(A) picks out the smallest element in the matrix and max(A) the biggest
- min(A, 'r') returns a row vector consisting of the smallest elements in each column
- max(A, 'c') returns a column vector containing the biggest elements in each row

```
-->A=[3 0 1; 2 2 7; 5 9 4]
```

```
A =
```

```
3. 0. 1.  
2. 2. 7.  
5. 9. 4.
```

```
-->min(A)
```

```
ans =
```

```
0.
```

```
-->max(A)
```

```
ans =
```

```
9.
```

```
-->min(A, 'r')
```

```
ans =
```

```
2. 0. 1.
```

```
-->max(A, 'c')
```

```
ans =
```

```
3.  
7.  
9.
```

Min/max position & value

- A variation of the `min()` and `max()` functions allow us to determine the position and value of the smallest alt. largest matrix element
- $[\text{min_value } \text{min_pos}] = \text{min}(\text{A})$ picks out the position and value (in this order!) of the smallest element in the matrix, $[\text{max_val } \text{max_pos}] = \text{max}(\text{A})$ the largest
- Note 1: The designation of vector elements (here `min_val` etc.) is irrelevant
- Note 2: If the matrix contains multiple min/max values only the position of the first is returned

```
-->A = [5 3 1; 2 4 6];  
  
-->[min_val min_pos] = min(A)  
min_pos =  
1. 3.  
min_val =  
1.  
  
-->[max_val max_pos] = max(A)  
max_pos =  
2. 3.  
max_val =  
6.
```

mean()

- And the previously mentioned logic a final time with the `mean()` function
- `mean(A)` returns the mean value of all matrix elements
- `mean(A, 'r')` returns a row vector consisting of the mean of each column
- `mean(A, 'c')` returns a column vector containing the mean of each row



```
-->A=[1 2 3; 4 5 6]  
A =
```

```
1. 2. 3.  
4. 5. 6.
```

```
-->mean(A)  
ans =
```

```
3.5
```

```
-->mean(A, 'r')  
ans =
```

```
2.5 3.5 4.5
```

```
-->mean(A, 'c')  
ans =
```

```
2.  
5.
```

size()

- The function `size()` can be used to find out the size of a matrix
- The answer is given as the number of rows and columns (in that order)
- When row and column variables are named, the answer is given in alphabetic order (here columns first)
- Matrices with string elements (strings were used in the dialog box in Ex 1-3 and will be discussed in detail later) are treated the same way

```
-->v1 = [1 2 3 4];
```

```
-->v2 = v1';
```

```
-->size(v1)  
ans =
```

1. 4.

```
-->size(v2)  
ans =
```

4. 1.

```
-->A = [1 2 3 4; 5 6 7 8];
```

```
-->size(A)  
ans =
```

2. 4.

```
-->[n,m] = size([1 2 3; 4 5 6])  
m =
```

3.

```
n =
```

2.

```
-->size(['You' 'Me'; 'Alpha' 'Beta'; 'Two' 'Three'])  
ans =
```

3. 2.

length()

- The function `length()` is related to `size()`. For a matrix with numeric elements `length()` returns the number of elements
- For a matrix with string elements `length()` returns the number of characters in each element
- Note that matrices with mixed numeric and string elements are **not allowed**

```
-->length([1.23; 456,7890; 9])  
ans =  
  
3.
```

```
-->length(['Hello world' 'SCILAB'; 'Alpha' 'Beta'])  
ans =  
  
11. 6.  
5. 4.
```

find(condition)

- The function `find()` identifies and returns the row locations of those matrix elements that satisfy the Boolean condition stated in the argument
- An empty matrix `[]` is returned in case no element satisfies the given condition
- The statement `X=3` is not a valid Boolean condition. Although a numeric answer is returned, it is not legitimate
- Later we shall see that `find()` can also be used with strings



```
-->X = [9 1 8; 2 7 3; 6 3 5];  
  
-->find(X<5)  
ans =  
  
2. 4. 6. 8.  
  
-->find(X==3)  
ans =  
  
6. 8.  
  
-->find(X=3)  
ans =  
1.  
  
-->find(X~=3)  
ans =  
  
1. 2. 3. 4. 5. 7. 9.
```

gsort()

- Scilab does not recognize Matlab's sort() function (it used to before version 5.3). Instead we must use gsort(), which is different but serves the same purpose
- As shown to the right, gsort() picks out matrix elements in decreasing order and returns them column by column
- We achieve Matlab-like sorting by adding the arguments 'r' (row) and 'i' (increase) to gsort()
- Check with Help for details on arguments

```
-->matr = [-1 4 -2 2; 1 0 -3 3; -4 5 0 -5]  
matr =
```

```
- 1. 4. - 2. 2.  
 1. 0. - 3. 3.  
 - 4. 5. 0. - 5.
```

```
-->s_matr = gsort(matr)  
s_matr =
```

```
5. 2. 0. - 3.  
4. 1. - 1. - 4.  
3. 0. - 2. - 5.
```

```
-->matr = [-1 4 -2 2; 1 0 -3 3; -4 5 0 -5];
```

```
-->Mtbl_sort = gsort(matr, 'r', 'i')  
Mtbl_sort =
```

```
- 4. 0. - 3. - 5.  
- 1. 4. - 2. 2.  
 1. 5. 0. 3.
```

testmatrix()

- Magic squares of different sizes can be produced with the `testmatrix('magi',n)` function. It is the same as the `magic(n)` function in Matlab
- Additional matrices that can be produced by the `testmatrix()` function is `testmatrix('frk',n)` which returns the Franck matrix, and `testmatrix('hilb',n)` that is the inverse of the nxn Hilbert matrix. Check with Help for details

```
-->testmatrix('magi',4)
```

```
ans =
```

16.	2.	3.	13.
5.	11.	10.	8.
9.	7.	6.	12.
4.	14.	15.	1.

```
-->testmatrix('magi',5)
```

```
ans =
```

17.	24.	1.	8.	15.
23.	5.	7.	14.	16.
4.	6.	13.	20.	22.
10.	12.	19.	21.	3.
11.	18.	25.	2.	9.

$\det(M)$ & rounding errors

- Practical problems often require the determinant of a (square) matrix to be calculated
- The command `det()` returns the determinant
- The determinant of Dürer's magic square is zero (the matrix is **singular**), but as shown, the rounding error prevents Scilab from returning the exact answer (recall that we encountered this problem before)
- To get rid of the rounding error we can use the `clean()` function. It returns zero for values below 1e-10

```
-->M = testmatrix('magi',4)
M =
16.  2.  3.  13.
 5.  11. 10.  8.
 9.  7.  6.  12.
 4.  14. 15.  1.

-->det(M)
ans =
- 1.450D-12

-->clean(det(M))
ans =
0.
```

Deleting rows and columns

- Rows and columns can be deleted by using a pair of square brackets
- We start with the 4x4 magic square, denoted "m" because we shall distort it
- We first delete the third column. The Colon Operator argument is used to retain all rows, the argument 3 points to the third column. The result is a 3x4 matrix
- In the second instance we delete the second row, to end with a 3x3 matrix

```
-->m = [16 3 2 13; 5 10 11 8  
--> 9 6 7 12; 4 15 14 1]  
m =  
16.   3.   2.   13.  
5.   10.  11.   8.  
9.   6.   7.   12.  
4.   15.  14.   1.  
  
-->m(:,3) = []  
m =  
16.   3.   13.  
5.   10.   8.  
9.   6.   12.  
4.   15.   1.  
  
-->m(2,:) = []  
m =  
16.   3.   13.  
9.   6.   12.  
4.   15.   1.
```

Changing rows and columns

- The logic on the previous slide can be used to changing rows and columns
- We start from the previous 3x3 matrix
- First change elements in the second row to zeros
- Then we change the last column to ones (note transpose)
- These operations can also be seen as inserting a defined row/column vector in place of an existing row or column

```
m =  
16.   3.   13.  
9.    6.   12.  
4.    15.  1.  
  
-->m(2,:)=[0 0 0]  
m =  
16.   3.   13.  
0.    0.   0.  
4.    15.  1.  
  
-->m(:,3)=[1 1 1]  
m =  
16.   3.   1.  
0.    0.   1.  
4.    15.  1.
```

Addressing matrix elements by linear indexing

```
-->M=testmatrix('magi',4)
M =
16.  2.  3.  13.
 5. 11. 10.  8.
 9.  7.  6. 12.
 4. 14. 15.  1.
```

```
-->M(14)
ans =
8.
```

```
-->M([1 6 11 16])
ans =
16.
11.
 6.
 1.
```

- Scilab regards matrices as column vectors. This allows us to address matrix elements in a simplified way
- We start from the 4x4 magic square
- Then we pick out the element (2,4), which is number 14 if you count along the columns
- Next pick out elements of the main diagonal
- Finally, change the elements of the second diagonal to zeros

```
-->M([4 7 10 13]) = [0 0 0 0]
M =
16.  2.  3.  0.
 5. 11.  0.  8.
 9.  0.  6. 12.
 0. 14. 15.  1.
```

Concatenation (1/2)

- Concatenation is the process of joining small matrices to make bigger ones
- In fact, even the simplest matrix is formed by concatenating its individual elements
- The pair of square brackets, [], is the **concatenation operator**
- The examples illustrate two basic cases of concatenation (the only difference are the transposed matrices in the second case)
- Note that if a semicolon (;) is placed after a command the result is suppressed, but with a comma (,) it is displayed (top case)

```
-->A = [1 2 3]; B = [4 5 6], C = [A,B]
```

B =

4. 5. 6.

C =

1. 2. 3. 4. 5. 6.

```
-->A = [1 2 3]'; B = [4 5 6]'; C = [A,B]
```

C =

1. 4.

2. 5.

3. 6.

Concatenation (2/2)

- In this example a 4x4 matrix has been created by concatenating four 2x2 matrices
- Lines have been overlaid to highlight the fused parts
- Alternatively, we could have concatenated four row or column vectors, a 3x3 matrix plus a row and column vector, etc.
- E can be treated as a normal 4x4 matrix. For instance, the command $A = E(2:3, 2:3)$ picks out the submatrix

$$\begin{bmatrix} 22 & 23 \\ 32 & 33 \end{bmatrix}$$

```
-->A = [11 12; 21 22];
```

```
-->B = [13 14; 23 24];
```

```
-->C = [31 32; 41 42];
```

```
-->D = [33 34; 43 44];
```

```
-->E = [A B; C D]
```

E =

$$\begin{array}{cc|cc} 11. & 12. & 13. & 14. \\ 21. & 22. & 23. & 24. \\ \hline 31. & 32. & 33. & 34. \\ 41. & 42. & 43. & 44. \end{array}$$

Operators (1/4): the Colon Operator (:)

- The Colon Operator, (:), emerged in the earlier examples
- It is one of the most important operators in Scilab
- A typical use is in the form:

$0:\%pi/36:\%pi$

Meaning: "Starting at 0, step by $\pi/36$ up to π "

- The first example shows that the truncated form 1:8 produces a row vector with increment 1. The second shows how to refer to rows 3-4, column 2, of the magic square

```
-->1:8  
ans =  
  
1. 2. 3. 4. 5. 6. 7. 8.
```

```
-->M = testmatrix('magi',4)  
M =
```

16.	2.	3.	13.
5.	11.	10.	8.
9.	7.	6.	12.
4.	14.	15.	1.

```
-->K = M(3:4,2)
```

```
K =
```

7.
14.

Operators (2/4): more examples with (:)

- The second example on the previous slide was a case of **subscript** manipulation of the type $M(i:j,k)$, where $i:j$ refers to the i :th to j :th rows and k to the k :th column
- There is often need to address part of a matrix. The idea should be understood well. Below are three more examples
- Note that the Colon Operator alone refers to the **entire row or column**

```
-->M = testmatrix('magi',4);  
  
-->A = M(2:3,2:3)  
A =  
  
11. 10.  
7. 6.
```

```
-->M = testmatrix('magi',4);  
  
-->B = M(:,3)  
B =  
  
3.  
10.  
6.  
15.
```

```
-->M = testmatrix('magi',4);  
  
-->C = M(3:4,:)  
C =  
  
9. 7. 6. 12.  
4. 14. 15. 1.
```

Operators (3/4): the \$ Operator

- The \$ Operator refers to the last value, $\$-1$ to the value next to the last, etc.
- The example to the right shows some uses of the \$ Operator
- The \$ Operator can be used to flip the order of elements in a vector, as shown below (an alternative method was demonstrated in Ex 1-3, Task 1)



```
-->v = [3 4 5 6 7 8 9];  
  
-->v($:-1:1)  
ans =  
  
9. 8. 7. 6. 5. 4. 3.
```



```
-->M = testmatrix('magi',4)  
M =  
  
16. 2. 3. 13.  
5. 11. 10. 8.  
9. 7. 6. 12.  
4. 14. 15. 1.  
  
-->M($)  
ans =  
1.  
  
-->M(1:$-1,$)  
ans =  
  
13.  
8.  
12.
```

Operators (4/4): the Backslash Operator (\)

- Backslash (\) denotes left matrix division. $x=A\backslash b$ is a solution to $A*x=b$, which is important e.g. in control engineering
- If A is square and nonsingular, $x=A\backslash b$ is equivalent to $x=\text{inv}(A)*b$ but the computation burden is smaller and the result is more accurate
- Here you can see the warning given when Scilab sees singularity in left division.

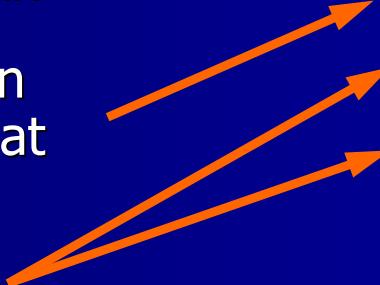
In this case Matlab produces a different answer than Scilab
(Example from book by Hunt et al.)



```
-->A = [3 -9 8; 2 -3 7; 1 -6 1]; b = [2 -1 3]';  
-->x = A\b  
Warning :  
matrix is close to singular or badly scaled. rcond = 4.1895D-18  
computing least squares solution. (see lsq).  
  
x =  
  
0.  
- 0.5641026  
- 0.3846154
```

Duplicating an $mx1$ vector to an mxn matrix

- The Colon Operator allows us to duplicate vectors to form a matrix
- Assume that we have the column vector $m = (2:2:6)'$; meaning that it has three rows
- We want to form a 3×4 matrix where each column consists of the vector m



```
-->m = (2:2:6)';
-->n = 4;
-->A = m(:, ones(n,1))
A =
 2. 2. 2. 2.
 4. 4. 4. 4.
 6. 6. 6. 6.
```

Pay attention to the command $m(:, \text{ones}(n,1))$. Verbally it can be interpreted as: "Form a matrix with the number of rows defined by the column vector m and the number of columns defined by the variable n . Fill the matrix with ones and multiply each row by the corresponding value of m . Repeat just once."

Singularities and left division

- The terms “singular” and “nonsingular” emerged on the previous slide
- A requirement of nonsingular **square matrices** is that the determinant is nonzero. Consider the following cases:

$$\begin{bmatrix} 6 & 2 \\ 5 & 3 \end{bmatrix} = 6 \cdot 3 - 2 \cdot 5 = 8 , \text{ it is therefore nonsingular}$$

$$\begin{bmatrix} 6 & 3 \\ 2 & 1 \end{bmatrix} = 6 \cdot 1 - 3 \cdot 2 = 0 , \text{ meaning that it is singular}$$

- Earlier we found that Dürer’s magic square is singular, so is e.g the matrix $A = [1 \ 2 \ 3; 4 \ 5 \ 6; 7 \ 8 \ 9]$
- Before performing left division with square matrices one should **check that the determinant of the coefficient matrix is nonzero**, e.g. by testing that $\text{clean}(\det(A)) \sim= 0$

Strings (1/6): they are matrices too

- Character (letters, text, special characters) strings can be created by using single or double quotes:

'This is a &#ck2 string', "and so is this"

- Typical use of strings is in plot commands, to define the title and x and y-labels. Other uses are interactive inputs and outputs (input(), disp(), etc.), and write commands (write(%io(2),.....))
- Strings are considered as 1x1 matrices (scalars) in Scilab, but mixed character/numeric strings are typically 1x3 matrices. It is shown on the next slide with the command

```
disp(['Was it €' string(a) 'that you said?'])
```

Elements: 1 2 3



- Example 2-4 shows an additional application of strings

Strings (2/6): disp(), string()

- The most usual string display command is `disp()`, where the text has to be in quotation marks: `disp('text')` or `disp(['text'])`
- Numeric data can be added to `disp([])`, but has to be converted to strings using the function `string()`
- Scilab knows Matlab's conversion command `num2str()`, but in the form `mtlb_num2str()`
- Leave out the square brackets and the elements are displayed as a column, starting with the last (Last In First Out)
- Commas are optional with square brackets, but not with brackets only

```
-->a = 125;  
-->disp(['Was it €' string(a) 'that you said?'])  
!Was it € 125 that you said? !
```

```
-->b = 521;  
-->disp(['No, I said €' mtlb_num2str(b) '!'])  
!No, I said € 521 ! !
```

```
-->disp('in action', 'LIFO', 'This is')  
This is  
LIFO  
in action
```

Strings (3/6): disp() vs. mprintf()

- As seen on the previous slide, disp() gives LIFO output with an empty line between the elements
- To avoid the empty line, we can use the mprintf() function with the line declaration \n . In this case the output is First In First Out. Note that the argument is a single string
- Check with the Help Browser for other applications of mprintf()

```
-->disp('in action', 'LIFO', 'This is')
```

This is

LIFO

in action

```
-->mprintf("\nThis is \nFIFO \nin action")
```

This is

FIFO

in action

Strings (4/6): write(), input()

- String arguments in the `write()` and `input()` functions allow us to build interactive codes
- In the shown example `write()` is first used to give general information to the user, after which `input()` prompts for data required in the calculation
- The `%io(2)` argument of the `write()` function tells that the target is the Console. All actions after the script is loaded into Scilab take place on the Console

```
// strings.sce           /  
  
// Demo of write() and input() functions /  
  
clear,clc;  
write(%io(2),'This is an interactive demo.');//  
write(%io(2),'You will be asked to give the base length');//  
write(%io(2),'and height of a triangle. Scilab then');//  
write(%io(2),'computes the area.');//  
write(%io(2),' '); // Empty row  
b = input('Give length of triangle base: ');  
h = input('Give height of triangle: ');  
write(%io(2),' '); // Empty row  
disp(['triangle_area = ' string(b*h/2)])
```

This is an interactive demo.
You will be asked to give the base length
and height of a triangle. Scilab then
computes the area.

Give length of triangle base: 5
Give height of triangle: 4

!triangle_area = 10 !

Strings(5/6): other useful commands

Some of the functions discussed earlier in this chapter can have string matrices (below S) as arguments:

prod(), min(), max(), mean()	Not defined for strings
size(S)	Returns the number of rows and columns in S
length(S)	Returns the number of characters in each string element
find(condition)	Returns the columnwise location of a string element in the matrix*
gsort(S)	Returns S with elements rearranged column-by-column in alphanumerically descending order*

*) See demo on the next slide

Strings(6/6): demo with find() & gsort()

- To the right is a 3x2 matrix called "cars"
- The function `find()` identifies and returns the location of a specified string within the matrix
- In case there is no match, an empty matrix is returned
- The function `sort()` orders string elements column-by-column in alphanumerically descending order (note that the number 343 is accepted without being declared string)

```
-->cars = ['Audi' 'BMW' 'Fiat'; '343' 'Saab' 'Xantia']
cars =
!Audi  BMW  Fiat  !
!
!343  Saab  Xantia !

-->find(cars=='Saab')
ans =
4.

-->find(cars=='Volvo')
ans =
[]

-->gsort(cars)
ans =
!Xantia  Fiat  Audi  !
!
!Saab    BMW   343 !
```

Symbolic computing

- Matrices of **character strings** are constructed as ordinary matrices, e.g. using square brackets
- A very important feature of matrices of character strings is the capacity to manipulate and create functions
- Symbolic manipulation of mathematical objects can be performed using matrices of character strings
- In the shown cases the function `trianfml()` performs symbolic triangularization of the matrix `sc`, and the function `evstr()` evaluates the expression `tsc`

```
-->sc = ['x' 'y'; 'z' 'v+w']
sc =
! x   y    !
!       !
! z   v+w !
-->tsc = trianfml(sc)
tsc =
! z   v+w      !
!           !
! 0   z*y-x*(v+w) !
-->x=1; y=2; z=3; v=5; w=4;
-->evstr(tsc)
ans =
 3.   9.
 0.  - 3.
```

Arrays: general

- The term “array” refers to any systematic arrangement of objects, usually in rows and columns (numeric arrays, diode arrays, antenna arrays, etc.)
- Arrays have some important uses, e.g. for **building tables**
- Arithmetic operations on arrays are done **element-by-element**, meaning that addition and subtraction are the same for arrays and matrices
- Scilab uses the **Dot Operator (.)** for array operations
- The table to the right is a list of array operators

+	addition
-	subtraction
.*	multiplication
./	right division
.\ /	left division
.^	power
.'	unconjugated array transpose

Arrays: building a table

- Assume that we have a column vector $n = (0 \ 9)'$
- We can then build a table with a simple function—in the shown case with columns for n , n^2 and 2^n
- This type of tables are useful e.g. when processing measurement data
- The second example shows that Scilab treats the created table as a normal matrix

```
-->n = (0:9)';
```

```
-->powers = [n n.^2 2.^n]  
powers =
```

0.	0.	1.
1.	1.	2.
2.	4.	4.
3.	9.	8.
4.	16.	16.
5.	25.	32.
6.	36.	64.
7.	49.	128.
8.	64.	256.
9.	81.	512.

```
-->p = powers(4:5,1:2)  
p =
```

3.	9.
4.	16.

```
-->q = powers(3,2)*powers(4,3)
```

```
q =
```

32.

Element-by-element multiplication and division

- Element-by-element multiplication with the use of the **Dot Operator** can also be performed on two-dimensional matrices
- In the first example we multiply, element-by-element, two 2x2 matrices to form a 2x2 product matrix C
- Note the different result with ordinary matrix multiplication
- And here we divide the same matrices element-by-element to form a 2x2 matrix of quotients

```
-->A = [1 2; 3 4]; B = [5 6; 7 8]; C = A.*B  
C =
```

```
5. 12.  
21. 32.
```

```
-->D = A*B  
D =
```

```
19. 22.  
43. 50.
```

```
-->E = A./B  
E =
```

```
0.2 0.3333333  
0.4285714 0.5
```

Right and left division

- As shown in the table above, Scilab allows left and right element-by-element division (\backslash and $/$ respectively)
- The difference between the two is which of the two division elements is the numerator and which the denominator
- As shown by the examples, left division means that the element in the left matrix becomes the denominator, with right division it is the nominator

The exponent function `exp()` is a special case in being defined as an element-by-element operation

```
-->A = [1 2; 3 4]
A =
1. 2.
3. 4.

-->B = [5 6; 2 -3]
B =
5. 6.
2. -3.

-->A.\B
ans =
5.           3.
0.66666667 - 0.75

-->A./B
ans =
0.2   0.3333333
1.5   -1.3333333
```

Dot Operator pitfalls

- In practical simulations Scilab often flashes error messages due to wrong use of the Dot Operator—or the absence of it
- A particular problem is division with an integer in the nominator. As shown to the here, the first case is interpreted by Scilab as $B = (1.0)/A$ and the second as $C = (1.0)./A$ **Try to remember!**
- Those with experience of **Matlab** should be aware that the priority of the Dot Operator is **different in Scilab**
- This is not an issue with multiplication

```
-->A = [1 2 3 4];  
-->B = 1./A  
B =  
  
    0.0333333  
    0.0666667  
    0.1  
    0.1333333  
  
-->C = (1)./A  
C =  
  
    1.   0.5   0.3333333   0.25  
  
-->D = 2.*A  
D =  
  
    2.   4.   6.   8.  
  
-->E = (2).*A  
E =  
  
    2.   4.   6.   8.
```

A few more functions

(1/5): modulo()

- Then command `modulo(n,m)` computes the remainder of n divided by m (where n and m are integers)
- With matrices `modulo()` computes the remainder element-by-element
- `modulo()` comes handy e.g. when we need to check if a number is even or odd (the if-then-else-end construct will be discussed in Chapter 11)
- There is a related function `pmodulo()`. Check with Help

```
-->modulo(3,2)
ans =
1.
```

```
-->n=[1,2; 10,15]; m=[2,2; 3,5];
-->modulo(n,m)
ans =
1. 0.
1. 0.
```

```
x=input('Give a number :');
if modulo(x,2)==0 then
    disp('Number is even');
else
    disp('Number is odd');
end
```

```
Give a number :24
Number is even
```

```
Give a number :1443
Number is odd
```

A few more functions

(2/5): getdate()

- We saw `getdate()` in action already in Example 1-3, where it was used to improve randomness
- Another use of `getdate()` is to put a date stamp on the printout of a simulation
- `getdate()` has numerous alternative arguments. In addition to those used in Ex. 1-3 there are e.g. the ones shown to the right. Check with Help for details
- The starting point of the “clock” of `getdate()` is UTC 00:00 on 1 January 1970

```
-->xp=getdate();
```

```
-->xp(1),xp(2),xp(3),xp(4),xp(5),xp(6),xp(7)  
ans =
```

```
2011.  
ans =
```

(1) present year

```
3.  
ans =
```

(2) present month

```
12.  
ans =
```

(3) present week

```
83.  
ans =
```

(4) day of the year

```
5.  
ans =
```

(5) weekday (Thu)

```
24.  
ans =
```

(6) day of the month

```
11.
```

(7) hour of the day



A few more functions

(3/5): unique()

- Recall that `unique()` was used in Ex. 1-3 in the condition `length(unique(numbers))<7` to ascertain that the lotto row contained only unique numbers
- Here `unique()` is used to identify generated integers in the range [1,5]
- In the second case `unique()` picks out unique rows in a matrix. Change 'r' to 'c' to find unique columns
- Compare `unique()` with `find()` that was discussed earlier

```
-->M=round(5*rand(5,1))'  
M =  
  
5. 2. 1. 5. 4.  
  
-->unique(M)  
ans =  
  
1. 2. 4. 5.
```

round() was up in Ex. 1-3

0 and 3 are absent

```
A = [0 0 1 1;  
      0 1 1 1;  
      2 0 1 1;  
      0 2 2 2;  
      2 0 1 1;  
      0 0 1 1];  
  
disp(['Unique rows are:'])  
disp(unique(A,'r'))
```

Unique rows are:
0. 0. 1. 1.
0. 1. 1. 1.
0. 2. 2. 2.
2. 0. 1. 1.

A few more functions

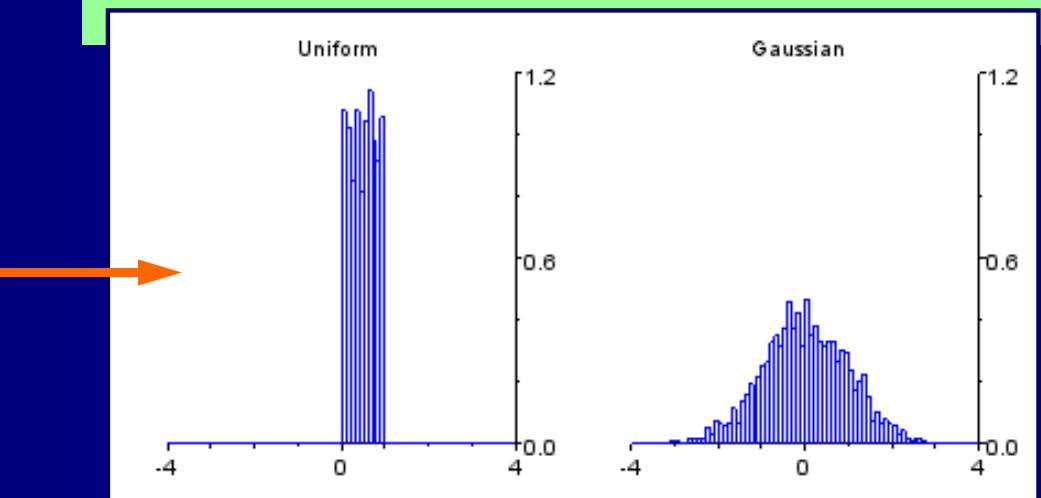
(4/5): rand()

- We have seen the random number generator `rand()` several times already
- `rand()` can generate two types of numbers, either with **uniform** or **Gaussian** distribution. Uniform is the default, Gaussian (normal) is selected with the argument 'normal' (or 'n')
- To the right are histograms of 2000 random numbers generated with uniform and Gaussian distribution (the latter with mean 0, variance 1)

```
// rand_demo1.sce
clear,clc,clf;

u_fun=rand(1,2000);
subplot(121);
histplot([-4:0.1:4],u_fun,2,'073','',[ -4,0,4,1.2],[3,3,2,3]);
xtitle('Uniform')

G_fun=rand(1,2000,'n');
subplot(122);
histplot([-4:0.1:4],G_fun,2,'073','',[ -4,0,4,1.2],[3,3,2,3]);
xtitle('Gaussian')
```



A few more functions

(5/5): grand()

```
// grand_demo.sce

// Plot histograms of Chi-square, exponential, /
// and Poisson distributions with 10^5 draws /

clear,clc,clf;

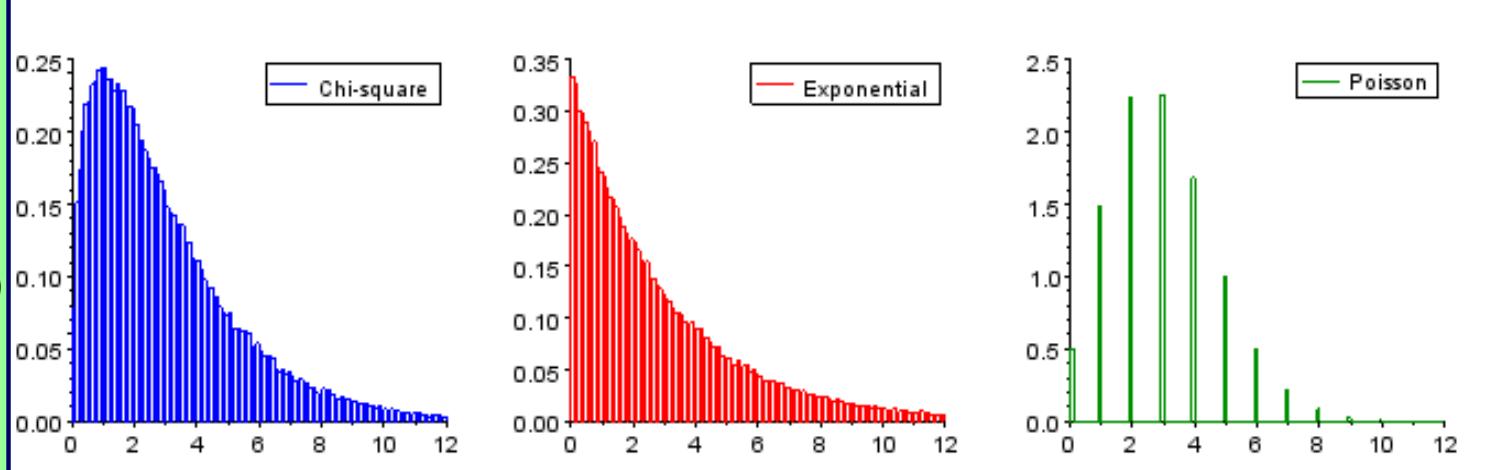
Chi=grand(100,1000,'chi',3) // Chi, 3 deg freedom
Exp=grand(100,1000,'exp',3) // Exponential, mean 3
Poi=grand(100,1000,'poi',3) // Poisson, mean 3

x = [0:.1:12];
subplot(131);
histplot(x,Chi,2)
legend(['Chi-square'])

subplot(132);
histplot(x,Exp,5)
legend(['Exponential'])

subplot(133);
histplot(x,Poi,13)
legend(['Poisson'])
```

The function `grand()` is **more versatile** than `rand()`. It allows most existing distributions to be generated. Shown here is an example with Chi-square, exponential, and Poisson distribution histograms



6. Examples, Set 2

Adds to what we have learned
so far



[Return to Contents](#)

Example 2-1: solving an equation system

- The task is to solve the following system of equations:

$$\begin{aligned}x_1 + 2x_2 - x_3 &= 1 \\-2x_1 - 6x_2 + 4x_3 &= -2 \\-x_1 - 3x_2 + 3x_3 &= 1\end{aligned}$$

- We can write it in the matrix form $Ax = b$, where:

$$A = \begin{bmatrix} 1 & 2 & -1 \\ -2 & -6 & 4 \\ -1 & -3 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

- Next we set up the equations in Scilab to find the solution x :

Ex 2-1: script & solution

The code for the equation system as entered in Editor and named algebra1.sce.
Note the Backslash Operator (\)

```
// algebra1.sce  
// Find the solution to x in /  
// Ax = b /  
  
A = [1 2 -1; -2 -6 4; -1 -3 3];  
b = [1; -2; 1];  
x = A\b
```

algebra1.sce has to be run from the Console since the script contains no disp() command

```
-->exec algebra1.sce  
  
-->// algebra1.sce /  
-->//  
-->// Find the solution x in /  
-->// Ax = b /  
-->/  
  
-->A = [1 2 -1; -2 -6 4; -1 -3 3];  
-->b = [1; -2; 1];  
-->x = A\b  
  
x =  
  
- 1.  
2.  
2.
```

The solution:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}$$

Ex 2-1: checking the result

- It is good practice to check one's solutions
- In this case it can be done by making sure that the residual $B - Ax$ is exactly zero
- The altered code is renamed `algebra1_check.sce` and saved before being executed
- The result is 0, as hoped for (note that there is no rounding error here; the result is exactly zero)

```
// algebra1.sce
// Find the solution x in /
// Ax = b
A = [1 2 -1; -2 -6 4; -1 -3 3];
b = [1; -2; 1];
x = A\b

// algebra1_check.sce
// Make sure that b - Ax = 0
residual = b - A*x
```

x =
- 1.
2.
2.

residual =
0.
0.
0.

Ex 2-1: what should have been done before

Problem: The determinant of the coefficient matrix A must be non-zero

```
-->A = [1 2 -1; -2 -6 4; -1 -3 3];  
-->det(A)  
ans =  
- 2.
```

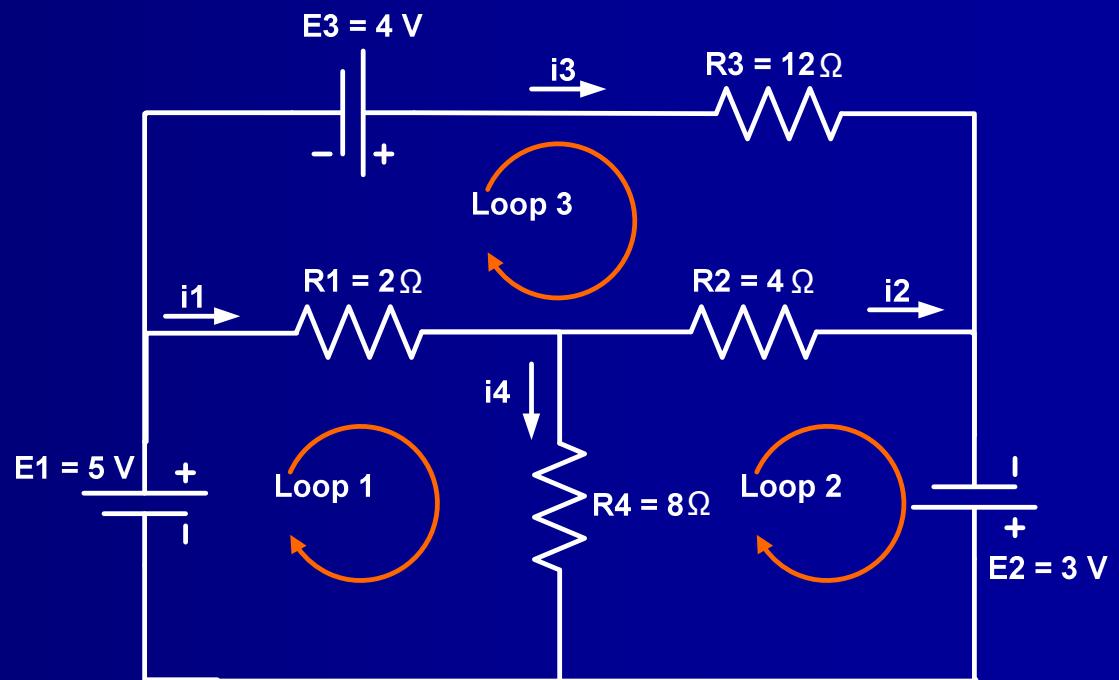


- In line with what has been said earlier, we should start by checking that the determinant of the coefficient matrix A is nonsingular (ok, Scilab would have yelled if that had been the case)
 - We can test it in hindsight and see that this is not the case
 - When writing a program for practical applications we must include the zero check in the script. This, however, requires **flow control** (conditional branching) that will be discussed in Chapter 11

Example 2-2: solving currents in a DC circuit

Task: Determine the four currents i_1 , i_2 , i_3 , and i_4 for the shown DC circuit

As drawn, the figure allows Kirchhoff's voltage law to be applied. However, the method leads to a **non-square matrix** and tools like the Backslash Operator (\) and multiplication with inverse matrices cannot be applied



Ex 2-2: mesh-currents

Instead, superposition of currents with **mesh-current equations** can be used. Along the current loops the diagonal term resistances are:

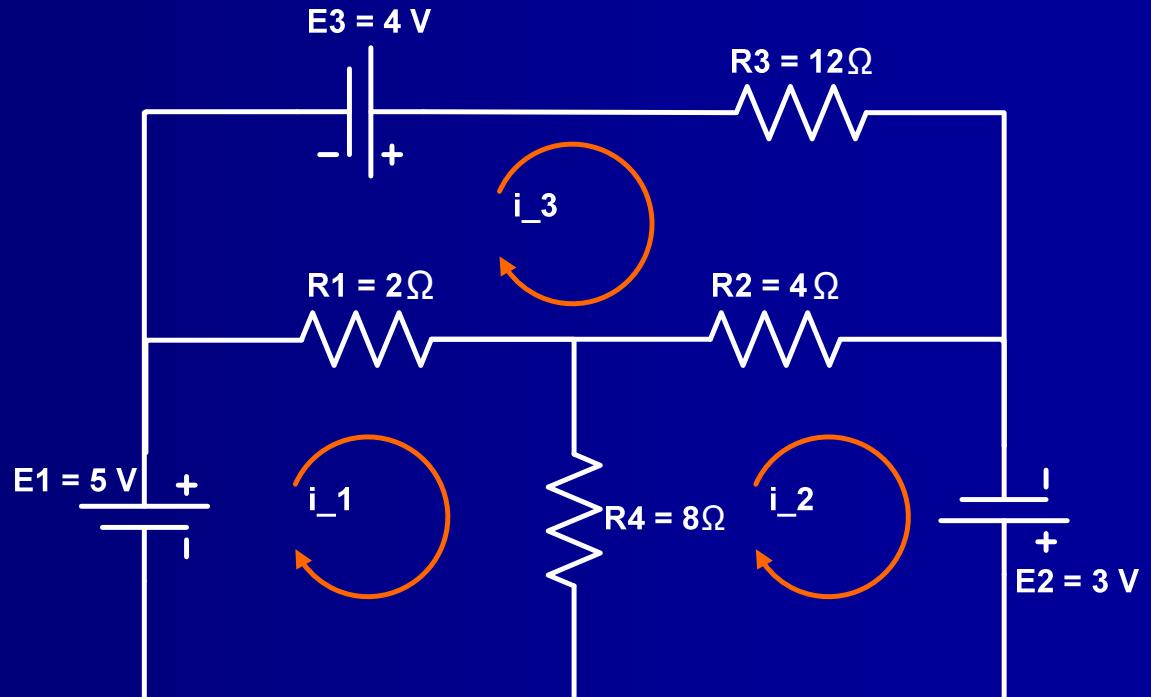
$$R_{11} = 10 \Omega$$

$$R_{22} = 12 \Omega$$

$$R_{33} = 18 \Omega$$

The common (off-diagonal) resistances are:

$$R_{12} = -8 \Omega, R_{13} = -2 \Omega, R_{21} = -8 \Omega, R_{23} = -4 \Omega, R_{31} = -2 \Omega, R_{32} = -4 \Omega \text{ (You should be able to figure out the logic)}$$



Ex 2-2: solution

These values allow us to write the following mesh-current equations:

$$\begin{bmatrix} 10 & -8 & -2 \\ -8 & 12 & -4 \\ -2 & -4 & 18 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 4 \end{bmatrix}$$

We execute the script in the Console and compute manually the current values that we are looking for:

```
i1 = i_1 - i_3 = 1.5 A
i2 = i_2 - i_3 = 1.25 A
i3 = i_3           = 1 A
i4 = i_1 - i_2 = 0.25 A
```

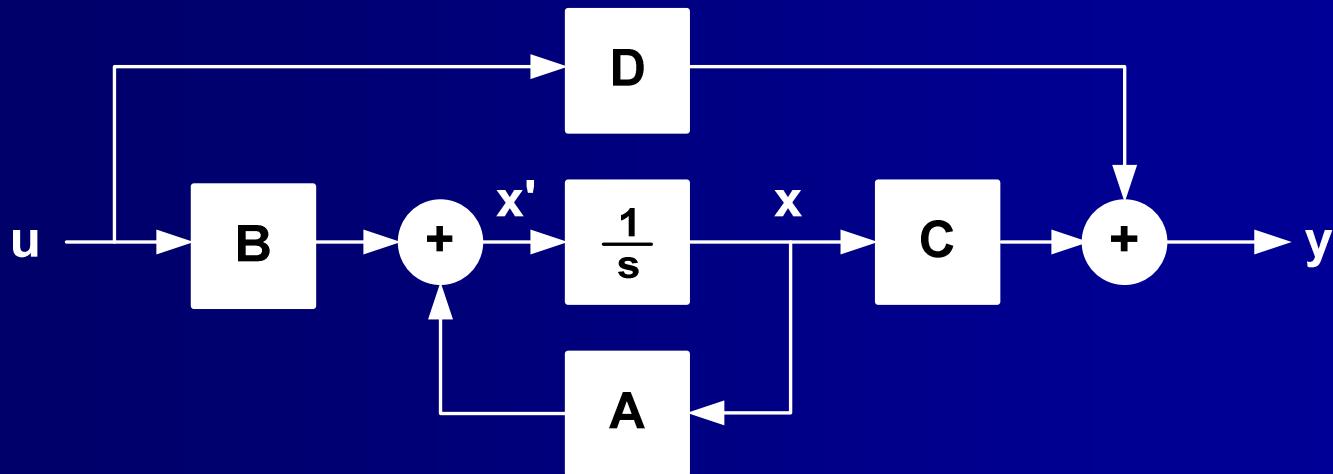
```
// circuit1.sce
// Mesh-current solution for Example 4 /
R = [10 -8 -2; -8 12 -4; -2 -4 18];
u = [5 3 4]';
i_n = R\u
residual = clean(u - R*i_n) // Check
```

```
i_n =
2.5
2.25
1.
residual =
0.
0.
0.
```

Ex 2-2: comments

- The example shows that we have to find the right method to be able to use matrix operations
- Is there reason to use matrices, which are the alternatives?
- The first alternative would be to proceed from the initial diagram and apply Kirchhoff's voltage law, and solve the problem manually. It is a quite tedious task
- Another alternative is to start manual calculations from the set of mesh-current equations by using **Cramer's rule**. However, it also requires a good dose of algebra since we have to compute determinants for several equations before we can divide the results to find the solutions
- In short, using Scilab to manipulate matrices simplifies the undertaking. With more complicated circuits the difference is even more pronounced

Example 2-3: continuous-time state-space model



- The figure shows a typical a continuous-time state-space model, defined by the matrix equations

$$\begin{aligned}x' &= Ax + Bu \\y &= Cx + Du\end{aligned}$$

where

A = system matrix
 B = input matrix
 C = output matrix
 D = feedforward matrix
 x = state vector
 x' = dx/dt
 u = input vector
 y = output vector

Ex 2-3: the task

- Assume a system given by:

$$A = \begin{bmatrix} 0 & 1 \\ -1 & -0.5 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 \end{bmatrix}$$

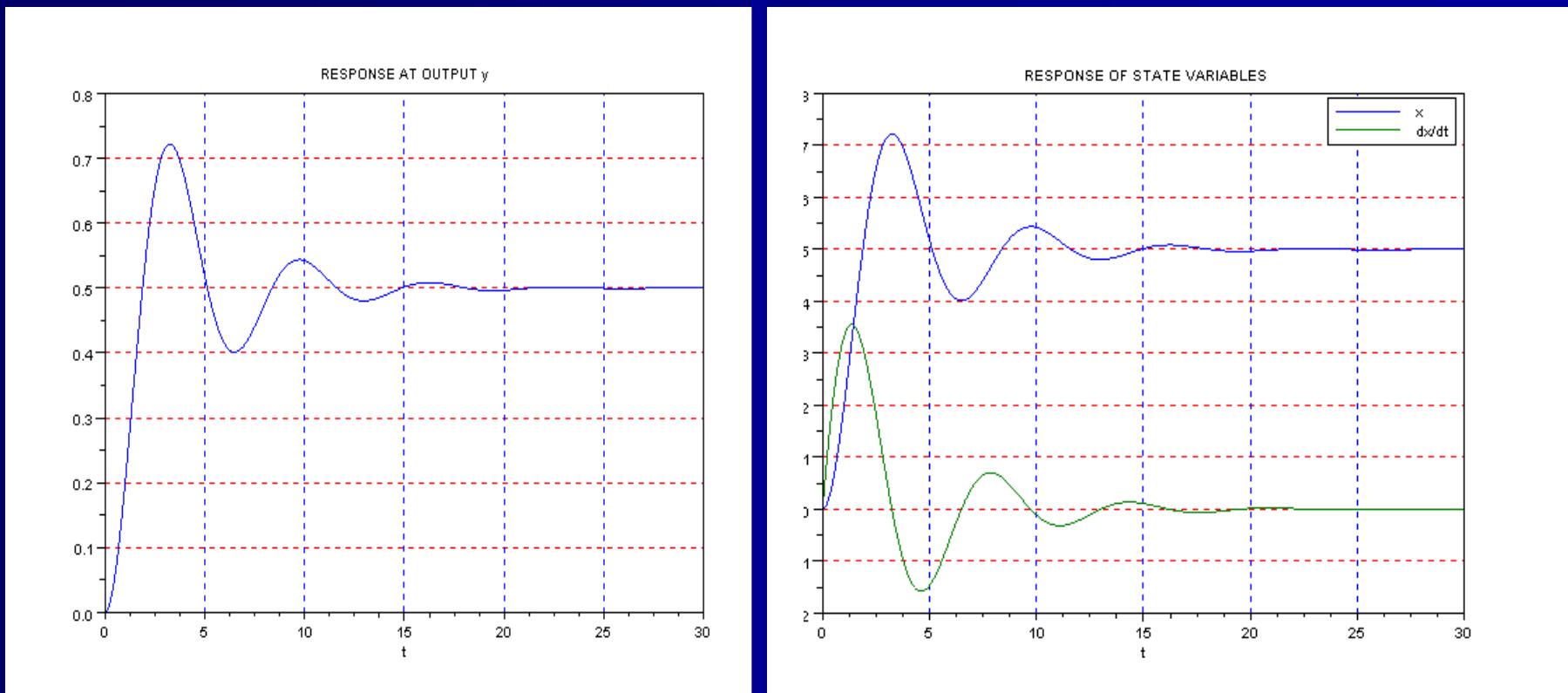
- The input u is constant at 0.5
- The initial state vector $x_0 = [0 \ 0]$, i.e., $x = 0$ at $t = 0$
- The task is to plot the output y and state variable responses (x, x') for $t = 0 \dots 30$

Ex 2-3: script

- First the state-space model is defined
- Note the syslin() function that defines a linear system
- Next, the responses due to initial state and external input signal u are simulated using csim()
- To finish, the responses at output y and state variables x and x' are plotted in separate windows

```
// state_space.sce  
  
// Simulates a continuous-time state-space //  
// system model //  
  
clear,clc;  
A=[0,1;-1,-0.5]; // System matrices  
B=[0;1];  
C=[1,0];  
D=[0];  
x0=[0;0]; // Initial state  
sys=syslin('c',A,B,C,D,x0); // Create cont.-time ('c') system model  
t=[0:0.1:30]; // Time vector  
u=0.5*ones(1,length(t)); // Create constant input signal  
  
[y,x]=csim(u,t,sys); // Compute with u=input, y=output, x=states  
  
scf(1); clf; // Open and clear figure 1  
plot(t,y); // Plot response in y  
xtitle('RESPONSE AT OUTPUT y','t');  
ax1=gca(); ax1.grid=[2,5]; // Handle: add grid to y-plot  
scf(2); clf; // Open and clear figure 2  
plot(t,x); // Plot response in x  
xtitle('RESPONSE OF STATE VARIABLES','t');  
legend('x','dx/dt',1); // Add legend to x-plot  
ax1=gca(); ax1.grid=[2,5]; // Handle: add grid to x-plot
```

Ex 2-3: plots



Note the use of the function `scf(number)` (set current figure) to produce two plots

Ex 2-3: comments (1/3)

- Apart from demonstrating matrix operations, this example introduced a number of new concepts:
 - definition of a linear system with the `syslin()` function, in which the string ‘c’ as input argument denotes “continuous.” The initial state `x0=[0;0]` is not needed since `x0=0` is the default value, but it is there if we want to make changes
 - Scilab **lacks a unit step function**; the constant input signal is constructed with a unit vector (using `ones()`) of length = t
 - simulation of the defined system was done by the `csim()` function, with u, t, and sys as input arguments
 - `csim()` produces the output arguments y and x, which are used by the plotting commands. Check Help for a detailed explanation
 - two plots are created since, with this particular system, x and y would otherwise overlap. x and x' are plotted automatically
 - the `ax1=gca()` and `ax1.grid=[2,5]` pair of commands tells that we want a grid with blue vertical and red horizontal lines

Ex 2-3: comments (2/3)

For a linear system, we can use either a transfer function or state-space representation. Their differences:

Transfer function	State-space
External description	Internal description
Input/Output descriptions	State descriptions
Frequency response method	Time response method
Laplace transform	Matrix
Some internal couplings are hidden	State variables are shown
System representation becomes more compact with fewer parameters	Representation with more parameters
Single input / Single output	Multiple input / Multiple output

In common are block diagrams and their manipulations, poles and zeros

Ex 2-3: comments (3/3)

- It is possible to shift between transfer functions and state-space representation:
 - `tf2ss()`, transfer function to state-space
 - `ss2tf()`, state-space to transfer function
- These functions are needed e.g. when discretizing continuous-time models, for which Scilab has the function `dscr()` but which is valid only for state-space models
- See tutorial by Haugen, section 9.6, for a brief discussion. A detailed discussion is given in the obsolete *Signal Processing With Scilab*, sections 1.5, 1.7, and 2.1 (you can access both through <<http://wiki.scilab.org/Tutorials>>)

Example 2-4: string functions, script

This example relates to the discussion on strings in Chapter 5

String as `input()` argument

The `if...else...end` construct will be discussed in Chapter 11

Note interplay between `floor()` and `modulo()`

Strings as `disp()` arguments

```
// conv_seconds.sce
```

```
// The script asks for a number of seconds, /  
// checks that the given number is positive, /  
// then converts the number into hours, /  
// minutes, and seconds /
```

```
clear,clc;
```

```
time = input("Give time in seconds: ");
```

```
if time < 0
```

`// Check if time >= 0`
`// Display error message`
`// and abort execution`

```
    disp("ERROR, negative number")
```

```
    abort
```

```
else
```

```
    minut = floor(time/60);
```

`// Convert to minutes`
`// Remaining seconds`

```
    seconds = modulo(time,60);
```

`// Convert to hours`
`// Remaining minutes`

```
    hours = floor(minut/60);
```

`// Display answer`

```
    minutes = modulo(minut,60);
```

```
    disp(string(hours)+" hour(s) ...")
```

```
    +string(minutes)+" minute(s) ...")
```

```
    +string(seconds)+" second(s) ")
```

```
end
```

Ex 2-4: string functions, execution & comments

Below is the result of
three different runs



```
Give time in seconds: 0
```

```
0 hour(s) 0 minute(s) 0 second(s)
```

```
Give time in seconds: -3600
```

```
ERROR, negative number
```

```
Give time in seconds: 7465.33
```

```
2 hour(s) 4 minute(s) 25.33 second(s)
```

In the script, the initial cleaning command is `clear,clc;`. If `clf` was included it would cause the Graphics Window to pop up unnecessarily (in Ex 2-3 it would have produced an extra empty window)

In this example we for the first time use a **sanity check** (`if time < 0 ...`) to make certain that the user does not cause problems by wrong inputs

In a case like this **it is irritating** that the Console does not become active after the execution command is given on the Editor. You automatically begin to type in the response once the string command pops up, but the cursor is still on the Editor...

7. Graphics & plotting

2D & 3D plots, subplots & other types of plots; editing plots

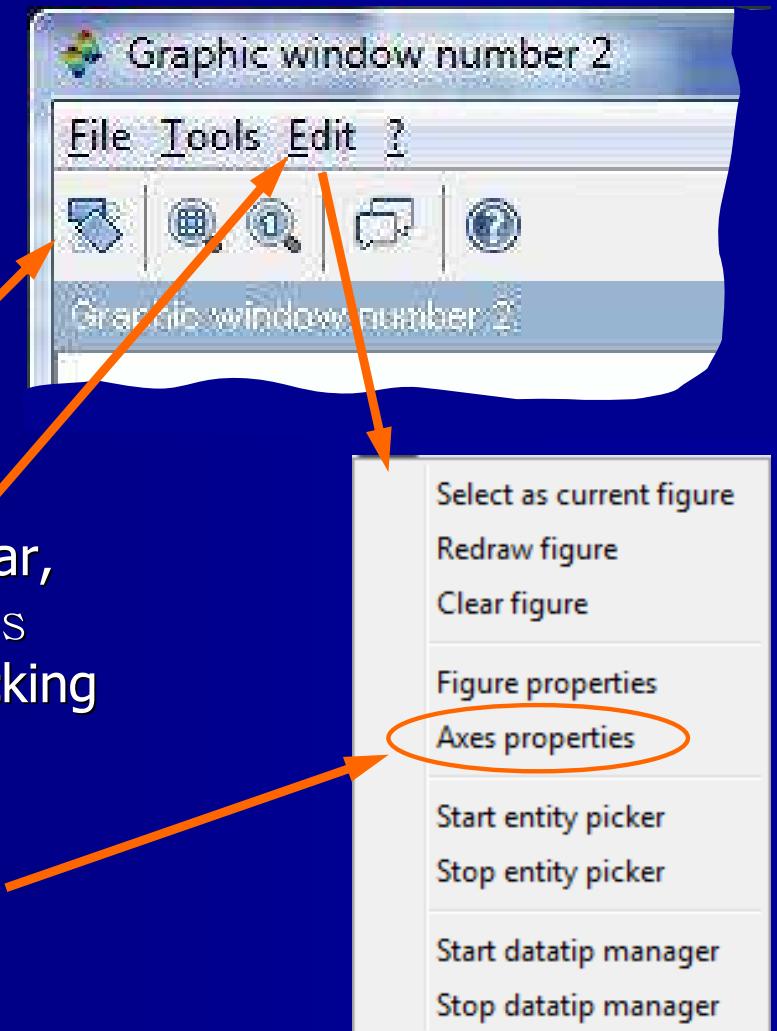


[Return to Contents](#)

The Graphics Window

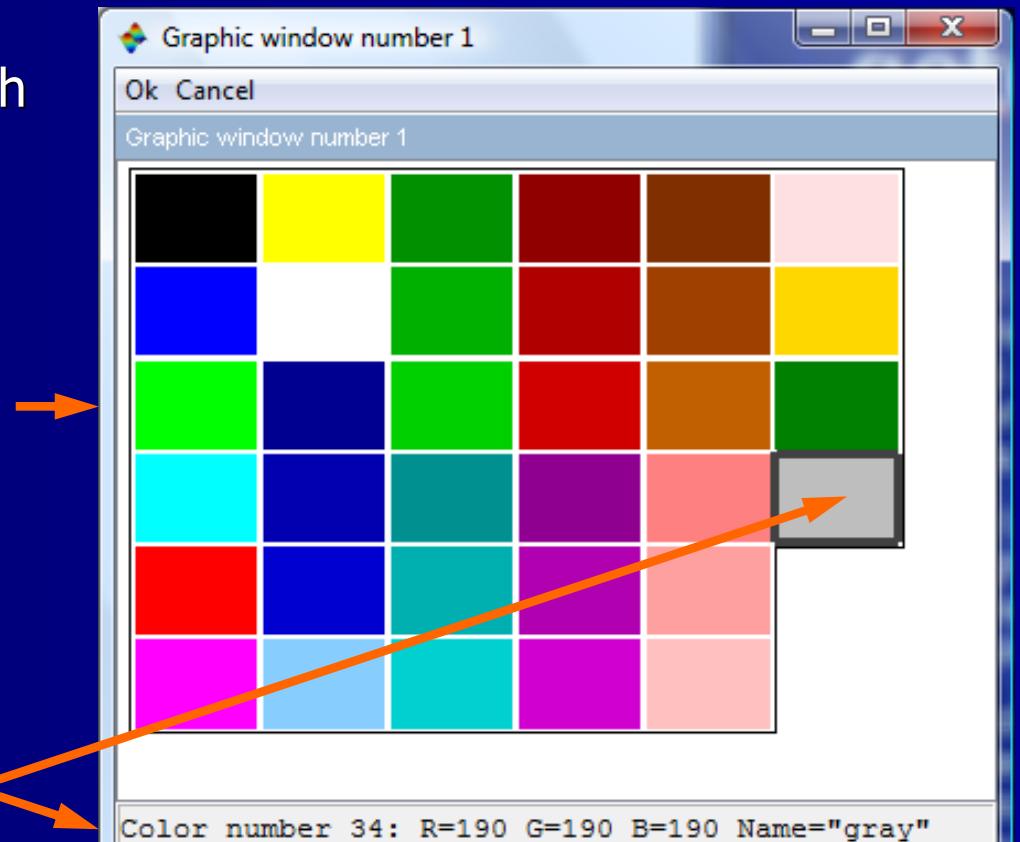
Note: The Demonstration feature has a good presentation of plotting functions, but it also contains obsolete ones

- The toolbar allows rotation and zoom of a plot
- Of real interest is Edit in the menu bar, and the Figure properties and Axes properties, that are shown when clicking on Edit
- However, Figure properties is ambiguous, all options can be found under Axes properties



getcolor()

- When working with graphics you may want to check which colors are available in Scilab and which their codes or names are
- Scilab's color palette can be brought up by entering the command `getcolor()` on the Console
- By clicking on a color in the palette its number, RGB composition, and name are displayed at the bottom of the window (Scilab 5.3.x does not display the last two, 33 "green" and 34 "grey").*

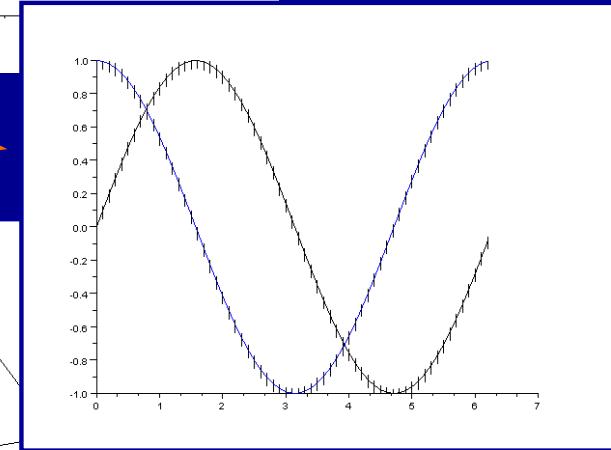
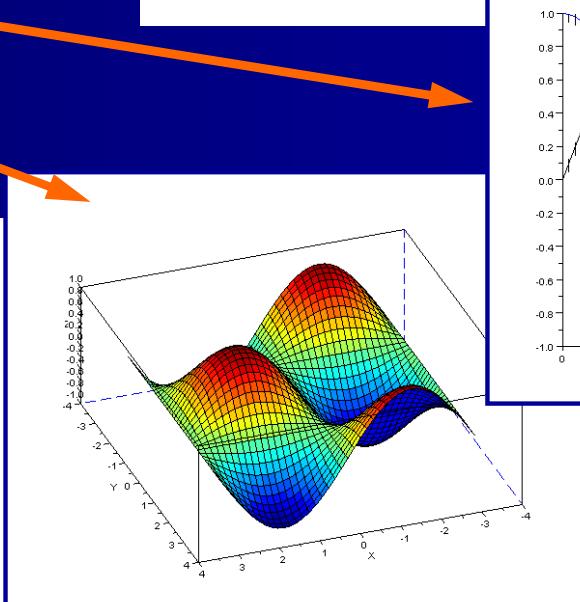
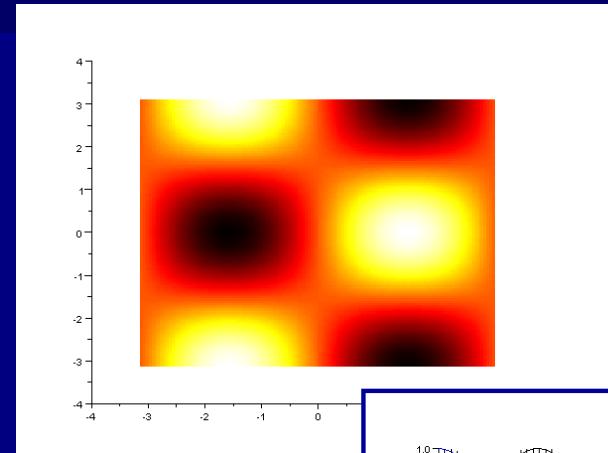
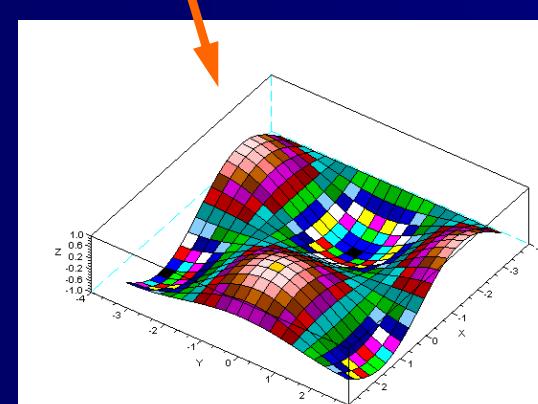


*) I miss more light colors for use as plot backgrounds.

Plot function demos

- You get a demo of certain plot functions by entering the function name on the Console.
Examples:

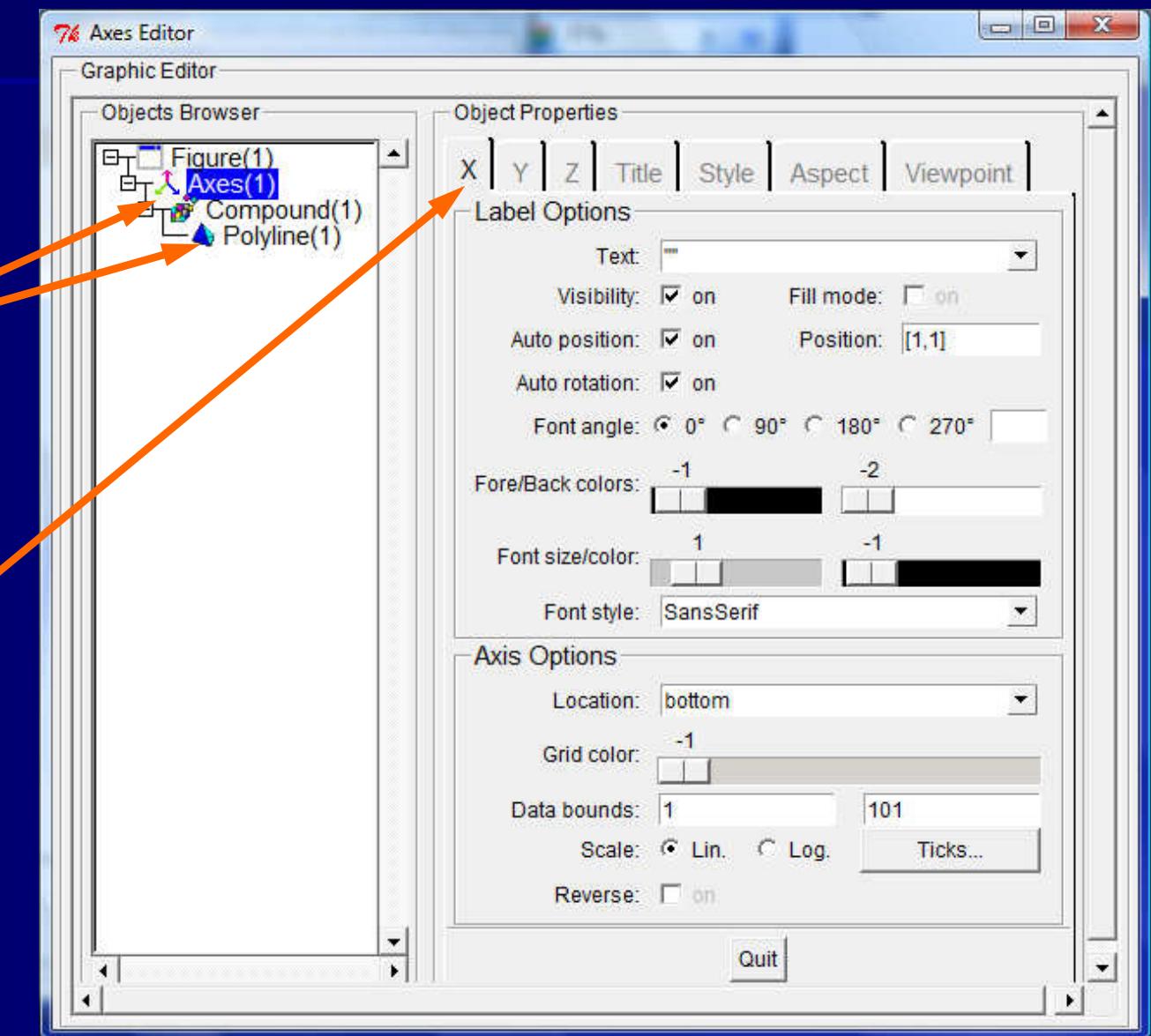
- grayplot()
- errbar()
- plot3d()
- fplot3d1()



Axes Editor

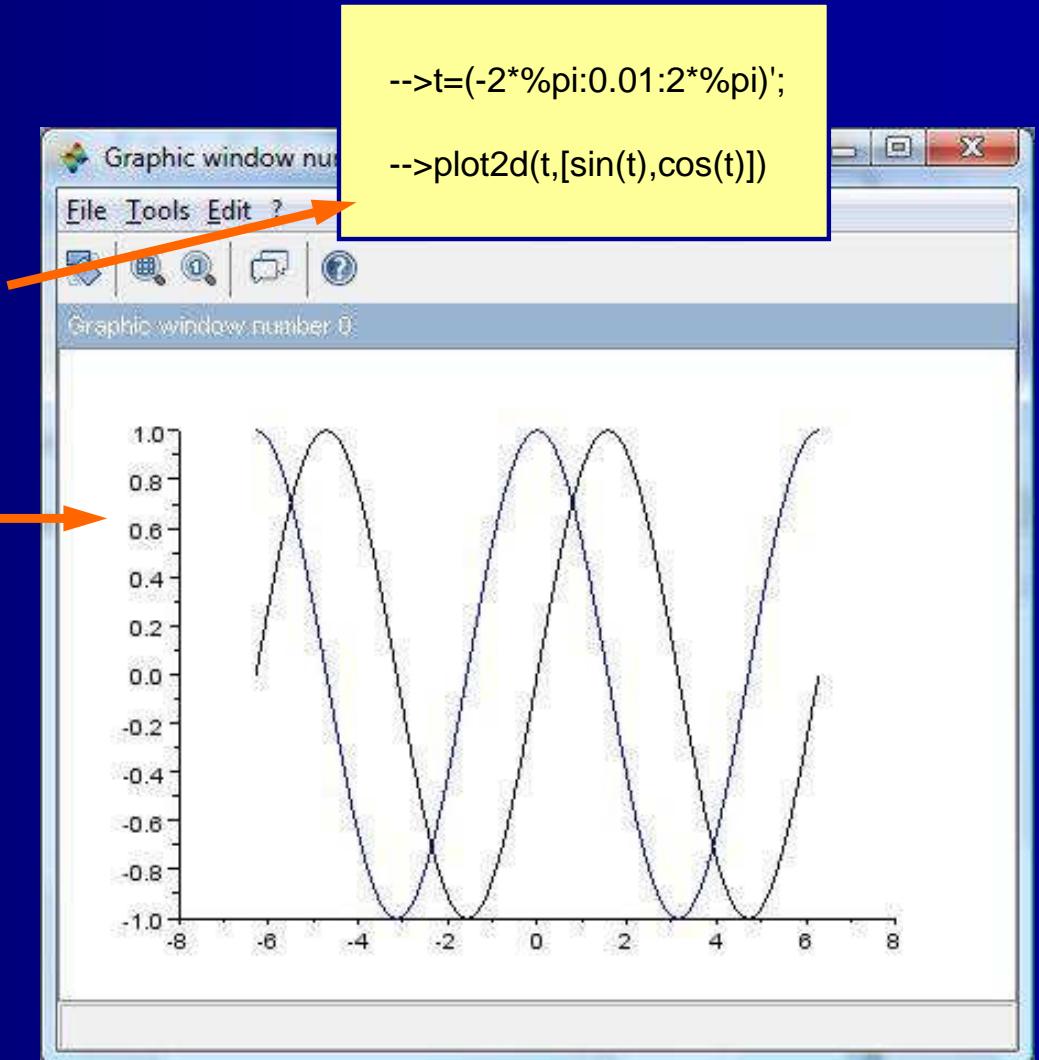
The most useful editing objects are `Axes()` and `Polyline()`. Here the Axes window is open

There are seven object properties that can be played with, the one for the x-axis is shown



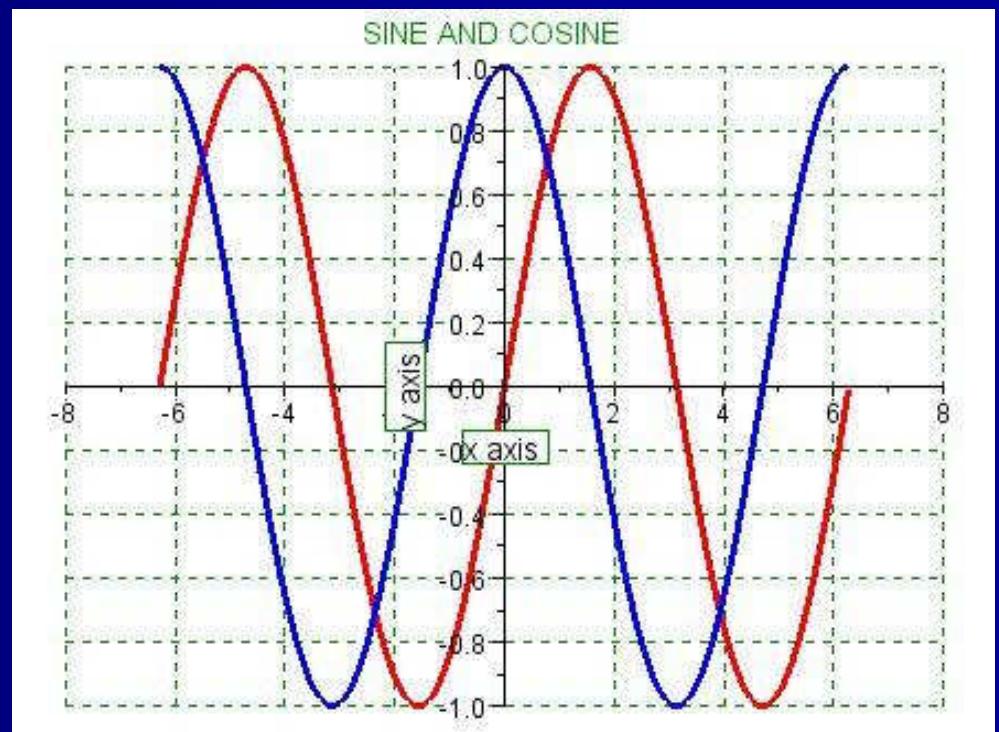
Plot editing demo, starting point

- Let's start by plotting a sine and cosine curve on the same frame
- The resulting plot is not very sexy
- So let's do some editing to give it a more attractive appearance
- Start by clicking Edit\Axes properties



Plot editing demo, edited plot

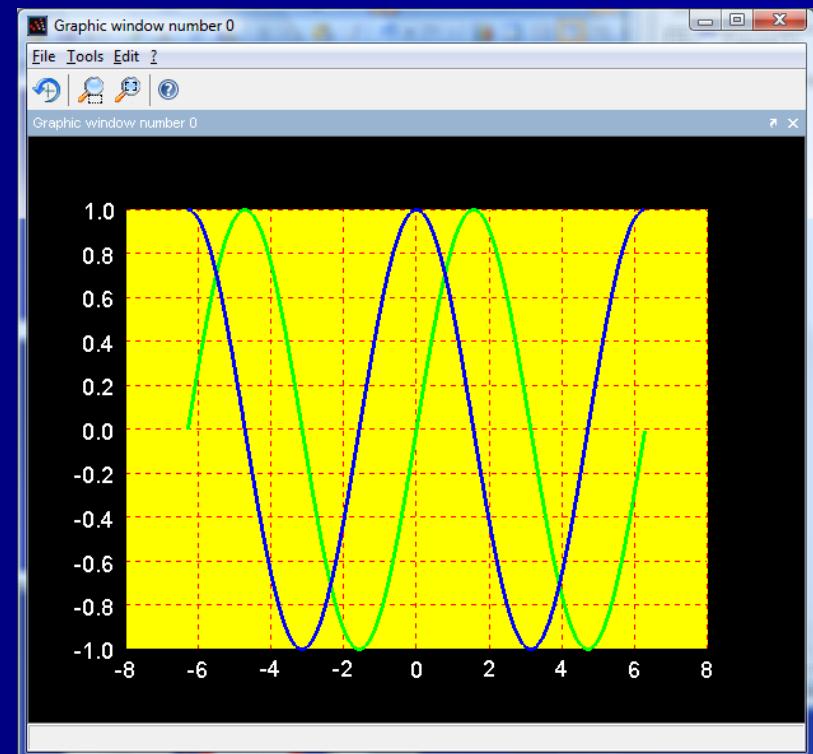
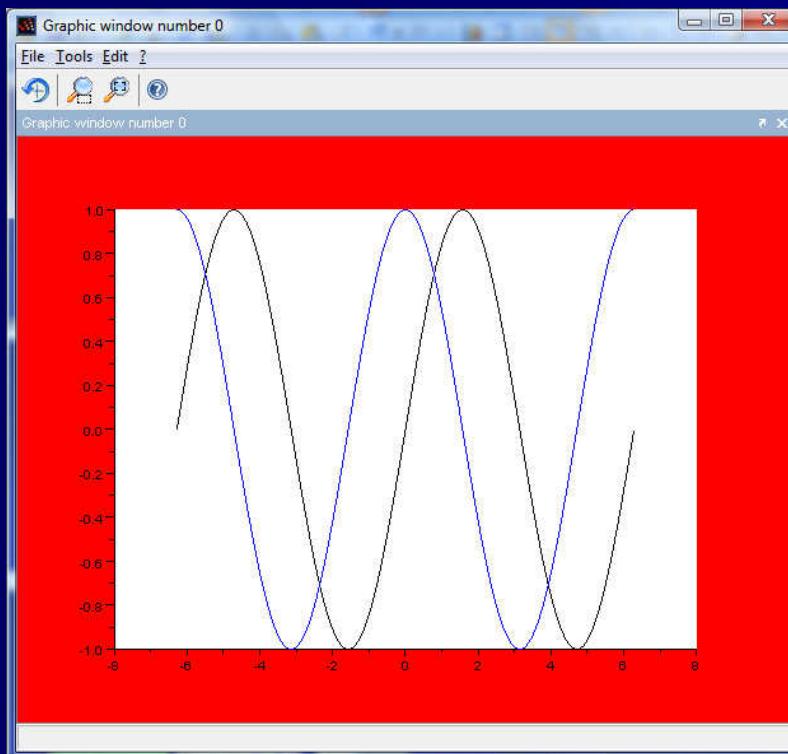
- To change sine and cosine colors, Click: Figure object\Colormap, mark a one (1) for: 1 RED, 2 BLUE
- Sine/cosine style: Axes\Compound\Polyline, select Line solid **3** for both
- x/y axes: Axes\Text "x/y axis", File mode **on**, Fore color **13**, Font size **3**, Axis location **middle**, Grid color **13**
- Title: Text "**SINE AND COSINE**", Font size **3**, Font Color **13**



- Style: Font size **2** (Axes labels)

Editing the Graphics Window

- The Figure Editor allows us to give the Graphics Editor a more colorful appearance. Play for a while with the Editor's object properties and you can find e.g. the following alternatives:



Graphics Window commands

- The command for **creating** a new Graphics Window for plots is:*

`scf()`

for figures:

`show_window()`

and the obsolete:**

`xset()`

- Windows-related clear/delete commands are e.g.:

`clf()`

`xdel()`

`delete()`

obsolete are:

`xclear()`

`xselect()`

`xbasc()` (Removed)

*) Single plots can be created without the `scf()` command.

**) Obsolete functions can be seen in most Scilab tutorials, but they should be avoided.

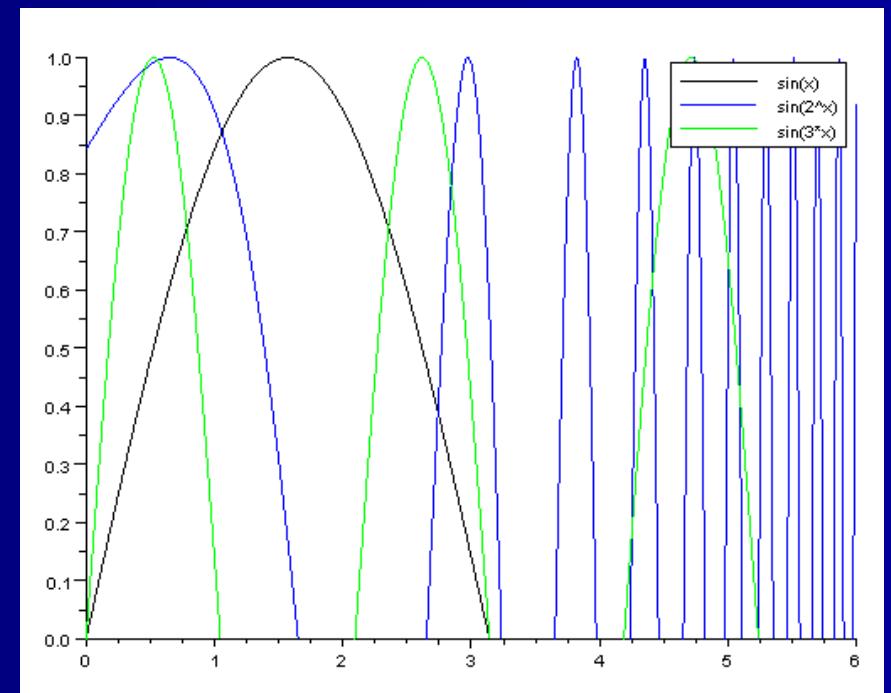
Scilab commands starting with `x` are usually associated with Graphics Window. The history of the `x` goes back to the **X window system** in Unix

Why plot() and plot2d()?

- Both plot() and plot2d() create 2D plots
- plot() is borrowed from Matlab. Persons with Matlab experience may want to use it (and frankly, **the benefits of plot2d() are doubtful**)
- Scilab has the added plot2d() function. It offers more options to tailor the plot. Multiple plots, for instance (recall however that multiple plots were done with plot() in Ex 1-2):

```
// multiple_plot.sce
// Demonstrates one alternative offered  /
// by the plot2d() function           /
clear,clc,clf;

x = [0:0.01:2*pi]';
plot2d(x,[sin(x) sin(2^x) sin(3*x)],rect=[0,0,6,1])
legend('sin(x)','sin(2^x)','sin(3*x)')
```



plot2d(): syntax

Note: plot2d() has also a slightly different old syntax

- The plot2d() syntax can be used as a guide for some other plot commands, e.g. for fplot2d() and histplot()
- plot2d() has the following arguments:

plot2d(logflag,x,y,optional arguments)

- x and y can be either vectors or matrices but with different outcomes for the plot. logflag is used only with logarithmic plots, we'll see it in a demo later
- The set of optional arguments is:

style, strf, leg, rect, nax

Graph style
(numeric)

Control of display captions
(by default "081")

Legend (string,
often seen
empty (''))

Axes label and tick
definitions (vector)

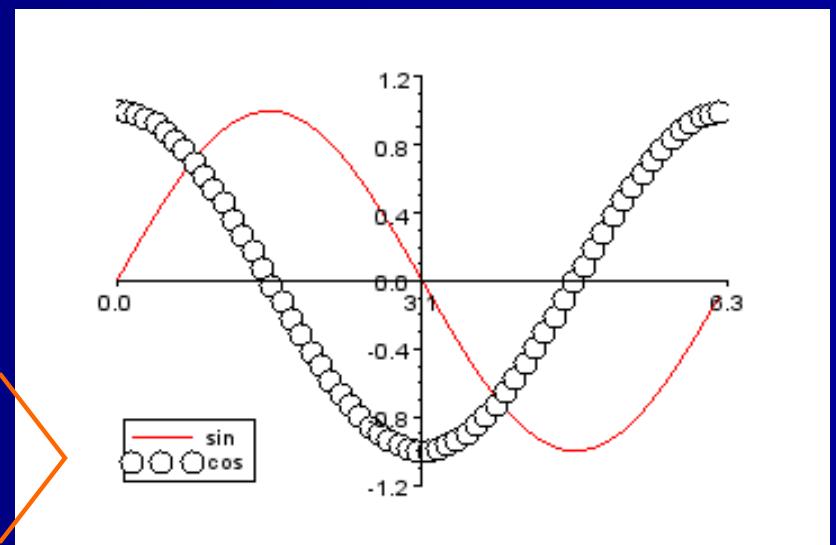
Minimum bounds for
the plot (vector: [xmin,
ymin, xmax, ymax])

plot2d(): syntax demo

```
// plot2d_demo.sce  
  
clear,clc,clf;  
  
x = 0:0.1:2*pi;  
y1 = sin(x);  
y2 = cos(x);  
  
style1 = 5;  
strf1 = '174';  
rect = [0,-1.2,2*pi,1.2]; // "rect"  
nax = [4,%pi,4,7]; // "nax"  
plot2d(x,y1,style1,strf1,'',rect,nax)  
  
style2 = -9;  
strf2 = '000';  
leg = 'sin@cos';  
plot2d(x,y2,style2,strf2,leg)
```

Scilab may not accept the legend command as it has done here (bug?)

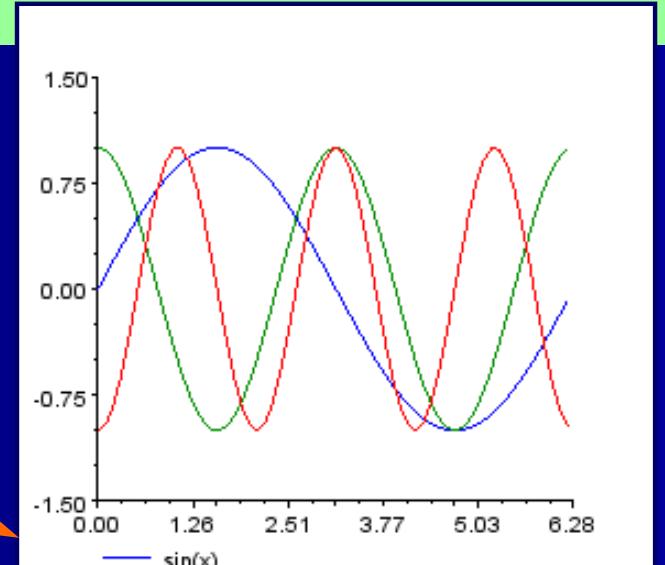
- linspace() is not accepted here
- style = 5 produces a red graph
- leg is empty (' ') in sine plot
- style=-9 produces circle marks
- A legend is added to the figure with the second plot command



plot2d(): multiple plots

- The previous slide showed how to create multiple graphs in a single window with two separate plot2d() commands
- Multiple graphs can be declared in a single plot2d() statement using a **vector argument**
- The case shown here also differs from the previous one by having argument declarations 'in situ'
- Scilab does not properly adjust the plot to the window; only the first legend shows

```
// plot2d_multiple.sce  
  
// Multiple graph declarations in a single /  
// plot2d() command /  
  
clear,clc,clf();  
  
x=[0:0.1:2*pi]';  
plot2d(x,[sin(x) cos(2*x) sin(3*x-%pi/2)],...  
[2,13,5],... // Graph colors  
leg="sin(x)@cos(x)@sin(3x)",... // Legend  
nax=[3,6,2,5],... // Ticks & marks  
rect=[0,-1.5,2*pi,1.5]); // Axes
```



plot2d(): style codes

- We have several times come across number codes for graph colors (style, the number after the x and y arguments in plot2d())
- Color codes are those that can be found with the getcolor() command on the Console. The most important ones are 1=black, 2=blue (9=dark blue), 3=green (13=dark green), 5=red, 8=white, and 25=brown
- On the previous slide we saw that the code -9 creates circles. Plug in getmark() on the Console to see the whole list, including codes for mark sizes that you can use with handle commands. There are in all 15 of these marks (always black):

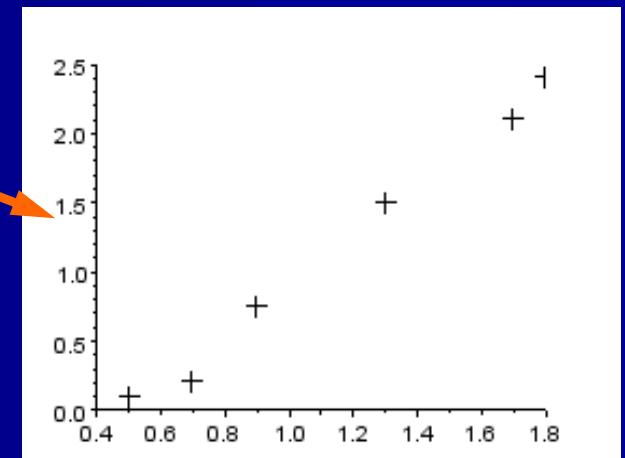
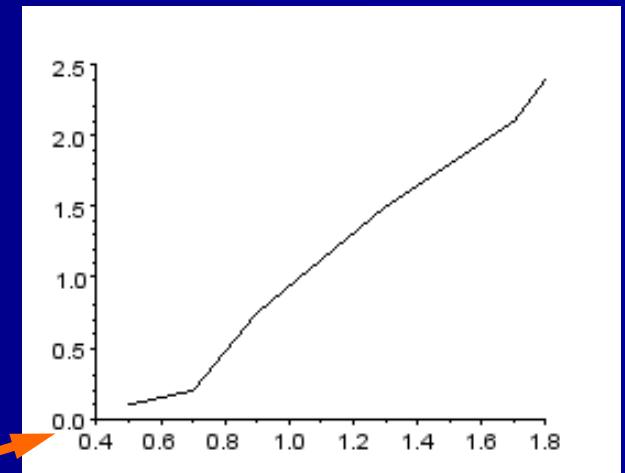
0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14
•	+	×	⊕	◆	◇	△	▽	◇	○	*	□	▷	◁	☆

plot2d(): demo with matrices

The simple script below demonstrates the plot2d() command when arguments x and y are matrices, and the style is 1 and -1

scf() is used to open a new Graphics Window. Otherwise the + marks of the second plot2d() command would be on top of the first one

```
-->x = [.5 .7 .9 1.3 1.7 1.8]';  
-->y = [.1 .2 .75 1.5 2.1 2.4]';  
-->plot2d(x,y, style=1)  
  
-->scf();  
-->plot2d(x,y, style=-1)
```



The command is clearer if arguments are written in plain (style=-1) but, as shown in earlier demos, the number alone is enough

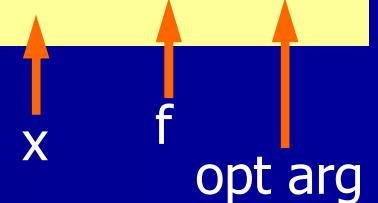
fplot2d()

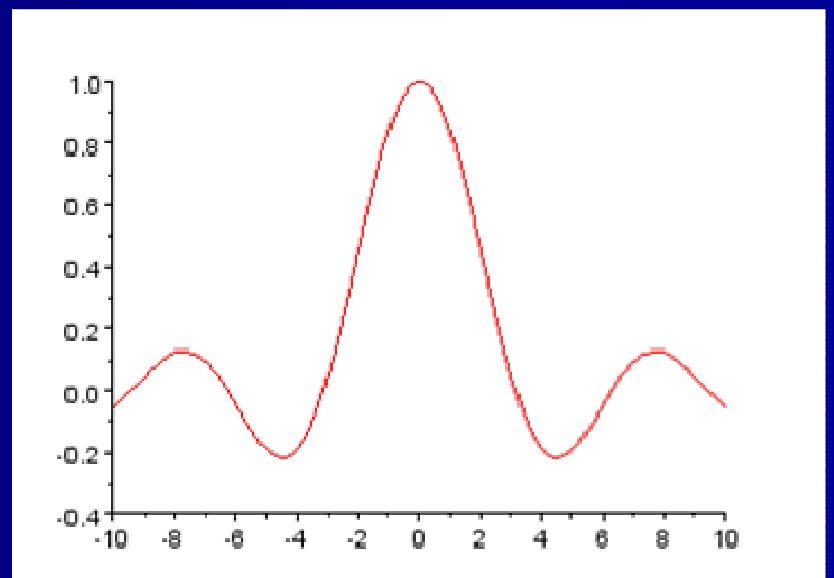
- fplot2d() is a variant of plot2d()
- With fplot2d() a **function** and its definitions can be **included in the arguments**
- The general form of fplot2d() is:

fplot2d(x,f,opt arguments)

- The demo to the right shows a case where
 - $x = \text{linspace}(-10,10,100)$
 - $f = \text{Scilab's in-built sinc function}$
 - $\text{style}=5$ is an optional argument
- There is also a **3D alternative**, fplot3d()

```
-->fplot2d(linspace(-10,10,100),sinc,style=5)
```



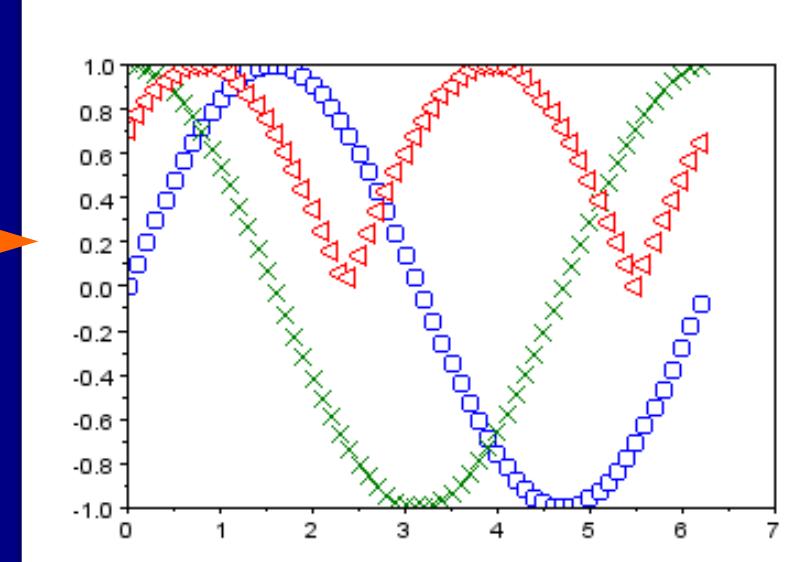


plot(): the beauty of simplicity (1/2)

- Matlab/Scilab's `plot()` function* offers a simpler way to distinguish between multiple plots than does `plot2d()`. It is by using keyboard characters, the way it was done on teleprinters half a century ago
- Here three graphs are plotted with one `plot()` command. The style definitions are '`'o'`', '`'x'`' and '`'<'`'. Note that `t` is repeated for each graph
- It can be seen that '`'<`' (red) gives a triangle that points in the direction of the path of the line

*) Scilab's `plot()` function does not support all properties of its Matlab counterpart

```
// plot()_demo.sce
// Demonstration of plot() syntax /
clf();
t=0:0.1:2*pi;
plot(t,sin(t),'o',...
      t,cos(t),'x',...
      t,abs(sin(t+pi/4)),'<')
// Plot with 'o'
// Plot with 'x'
// Plot with '<'
```



plot(): the beauty of simplicity (2/2)

The following list contains main line style codes for plot():

-	Solid line (default)	x	Cross
--	Dashed line	'square' or 's'	Square
:	Dotted line	'diamond' or 'd'	Diamond
- .	Dash-dotted line	^	Upward-pointing triangle
+	Plus sign	v	Downward-pointing triangle
o	Circle	>	Right-pointing triangle
*	Asterisk	<	Left-pointing triangle
.	Point	'pentagram'	Five-armed star

Color arguments are: k – Black, w – White, r - Red, g - Green, b – Blue, c – Cyan, m – Magenta, y – Yellow (the letter should be in front of the style code, inside single or double quotes, e.g. 'r+')

3D graphs: plot3d()

- The syntax of plot3d() is quite similar to that of plot2d(). In addition to the mandatory x,y,z arguments, the plot 3d() function can—among other possibilities—have following arguments:

plot3d(x,y,z,theta,alpha,leg,flag,ebox)

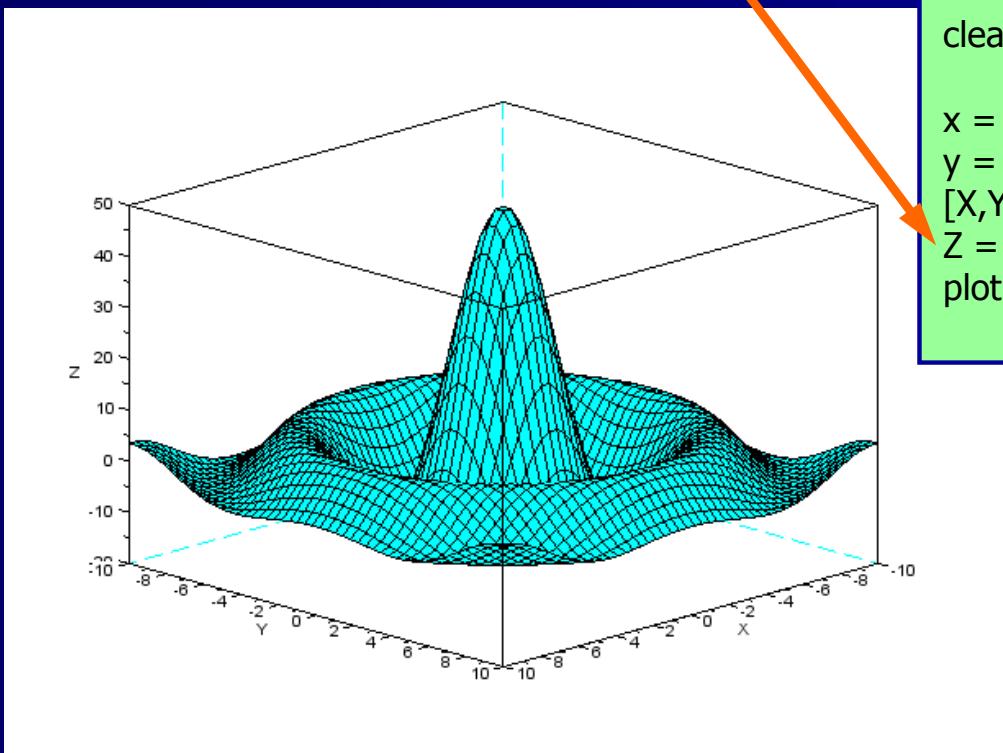
- Check with Help for an explanation
- Below we'll plot a 3D graph of the sinc function $\sin(x)/x$, using some of the surface definition capabilities of plot3d()
- Scilab defines only the 2D $\text{sinc}(x)$ function so to shift to 3D we will apply the expression

$$r = \sqrt{x^2 - y^2}$$

- Of the above mentioned arguments we'll use `leg="X@Y@Z"` to label x,y , and z axes and `flag=[mode,type,box]` to define surface color, scaling and frame of the plot

3D graphs: plot3d(), script & plot for 3D sinc()

Pay attention to $[X,Y] = \text{ndgrid}(x,y)$
& use of the Dot Operator in Z



```
// sinc3D.sce  
  
// Plot the sinc function (sin(x)/x) using plot3d() /  
// with surface definition arguments /  
  
clear,clc,clf;  
  
x = linspace(-10,10,50);  
y = linspace(-10,10,50);  
[X,Y] = ndgrid(x,y);      // Create array for xy grid  
Z = 50*sin(sqrt(X.^2 + Y.^2))./sqrt(X.^2 + Y.^2);  
plot3d(x,y,Z,leg="X@Y@Z",flag=[4,2,4])
```

Change plot3d() for plot3d1()
to get a different texture

A different approach to this
task is shown in Example 3-5.
There is a **bug** in the script
given in Help/meshgrid

3D graphs: surf(), task & script

- Write a script that plots the function
$$z = (2*x^2 - y^2)\exp(-x^2 - 0.5*y^2),$$
 where $-2 \leq x \leq 2$ and $-3 \leq y \leq 3$
- The function `linspace(a,b,m)` creates linearly spaced `x` and `y` row vectors ("from `a` to `b` with `m` equal increments")
- Using vectors `x` and `y`, the `[X,Y] = meshgrid(x,y)` command creates a 2D matrix in the `xy`-plane
- Generate `Z`-values for each element of the 2D matrix
- Plot the resulting 3D function

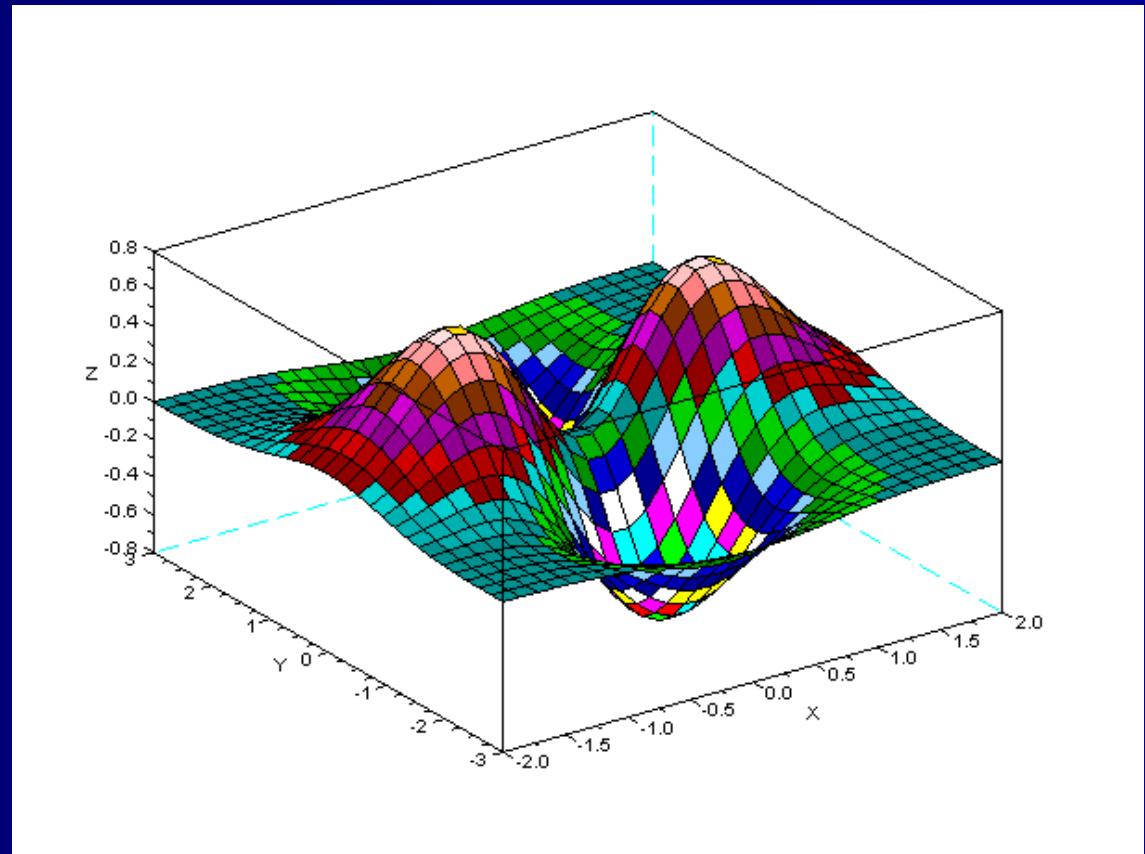
```
// surf_ex1.sce
// Plot the function
// z=(2x^2 - y^2)exp(-x^2 - 0.5y^2)
// for -2<x<2, -3<y<3, where < indicates
// "less than or equal to"
clear,clc,clf;
x=linspace(-2,2,30); // Linear spacing
y=linspace(-3,3,30);
[X,Y]=meshgrid(x,y); // Surface mesh
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);
surf(X,Y,Z) // Plot 3D surface
```

3D plots: surf(), plot

Ain't that cute!

The colors may not be all that great but they can be changed with handle commands.
This will be shown in Example 3-5

surf() has a **parallel form** called mesh()
that is used in the same way as surf()
but it lacks shading



If you click on the display button for surf() in the Help Browser, Scilab first displays a number of alternatives and then **crashes**.

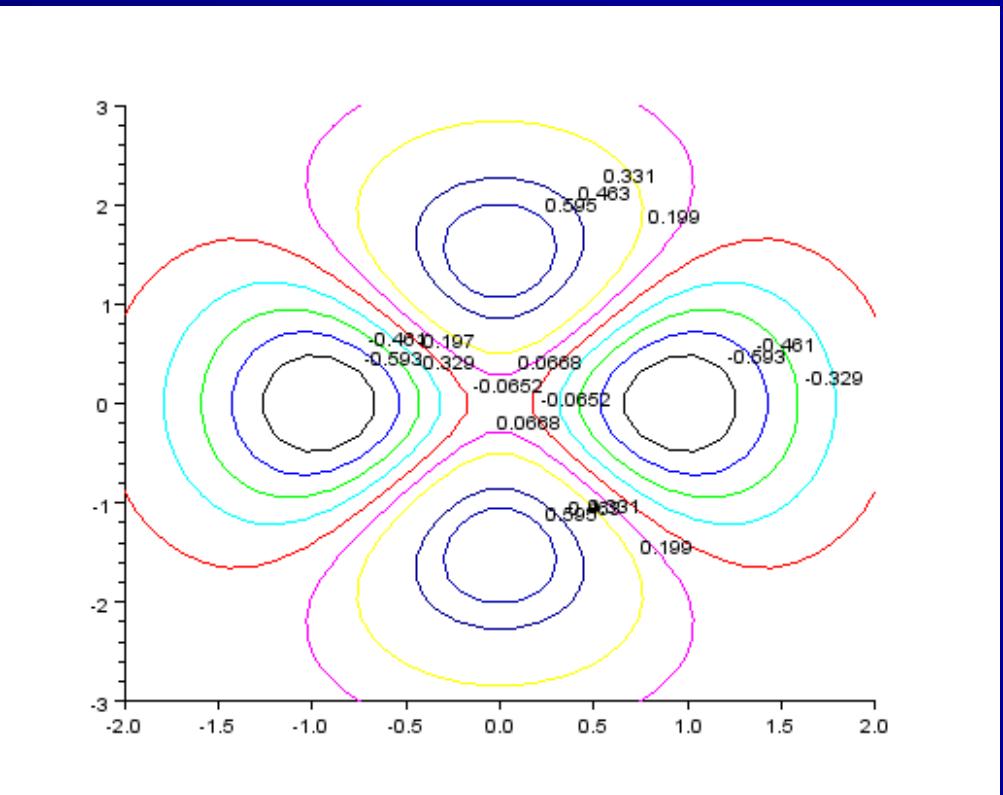
Contour plots: contour()

- Let's return to the expression $z = (2*x^2 - y^2)\exp(-x^2 - 0.5*y^2)$, and plot its 2D contour (level/height curves)
- It only requires the script's plot command to be changed

```
// contour.sce

// Plot the 2D height curves for the      /
// function                                /
// z=(2x^2 - y^2)exp(-x^2 - 0.5y^2)      /
// for -2<x<2, -3<y<3, where < indicates   /
// "less than or equal to"                  /

clear,clc,clf;
x=linspace(-2,2,30);
y=linspace(-3,3,30);
[X,Y]=meshgrid(x,y);
Z=(2*x.^2-y.^2).*exp(-X.^2-0.5*Y.^2);
contour(x,y,Z,10)
```



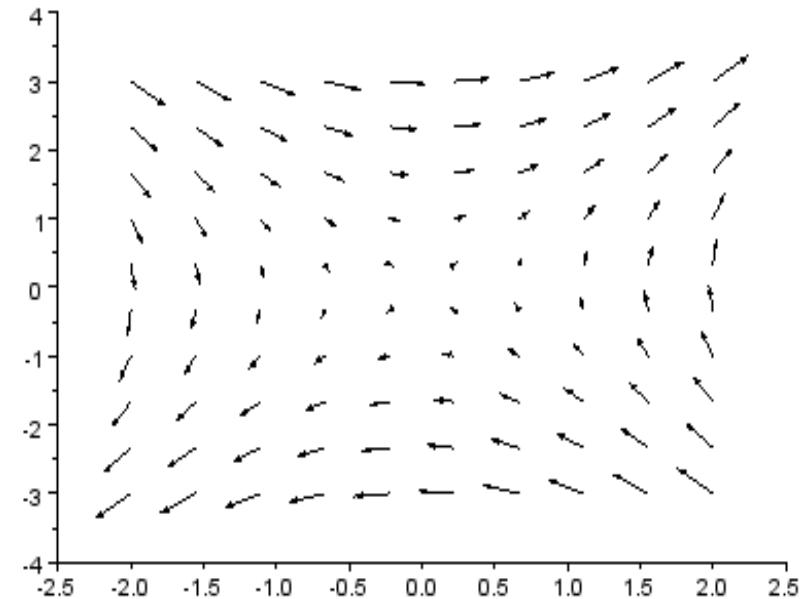
Vector fields: champ()

- The 2D vector field for the expression $z = (2x^2 - y^2)\exp(-x^2 - 0.5y^2)$ can be visualized by changing the plot expression to `champ()`, and adjusting the intervals in the `linspace()` functions:

```
// vector_field.sce

// Plot the 2D vector fields for the function /
// z=(2x^2 - y^2)exp(-x^2 - 0.5y^2) /
// for -2<x<2, -3<y<3, where < indicates /
// "less than or equal to" /

clear,clc,clf;
x=linspace(-2,2,10);
y=linspace(-3,3,10);
[X,Y]=meshgrid(x,y);
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);
champ(x,y,X,Y)
```



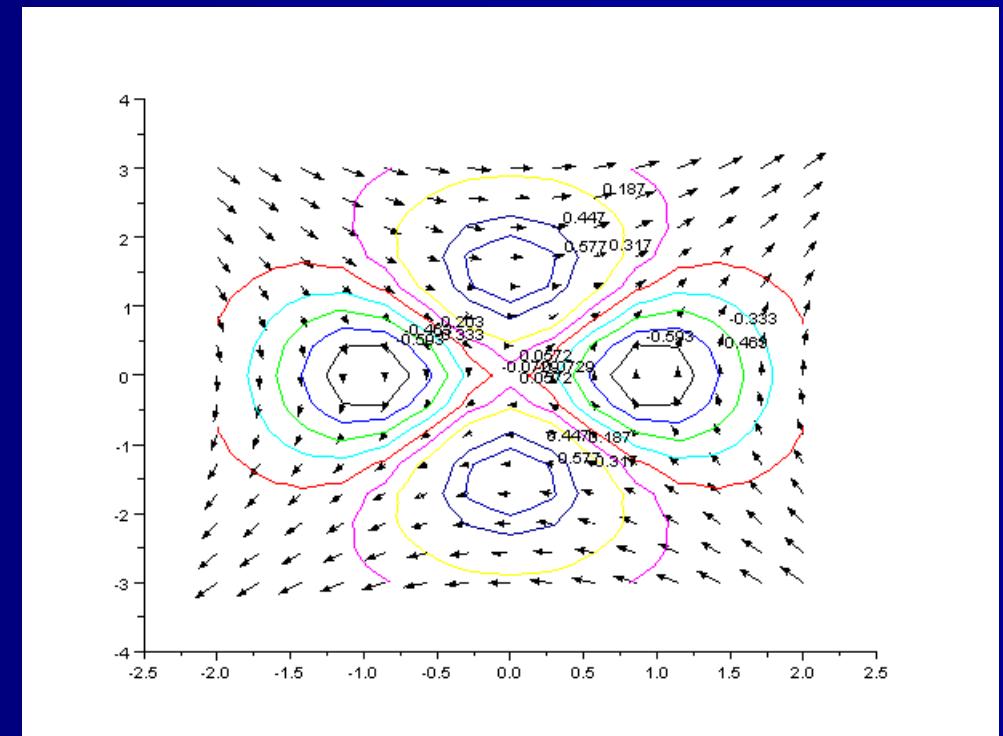
Mixed contours and vector fields

- Vector fields are not very informative per se, but the situation improves when they are fused with contours
- In the previous case, just insert the `champ()` and `contour()` commands into the same script and you get them in one plot:

```
// contour-vector.sce

// Plot the combined contour and vector      /
// fields for the function                  /
// z=(2x^2 - y^2)exp(-x^2 - 0.5y^2),      /
// for -2<=x<=2, -3<=y<=3              /

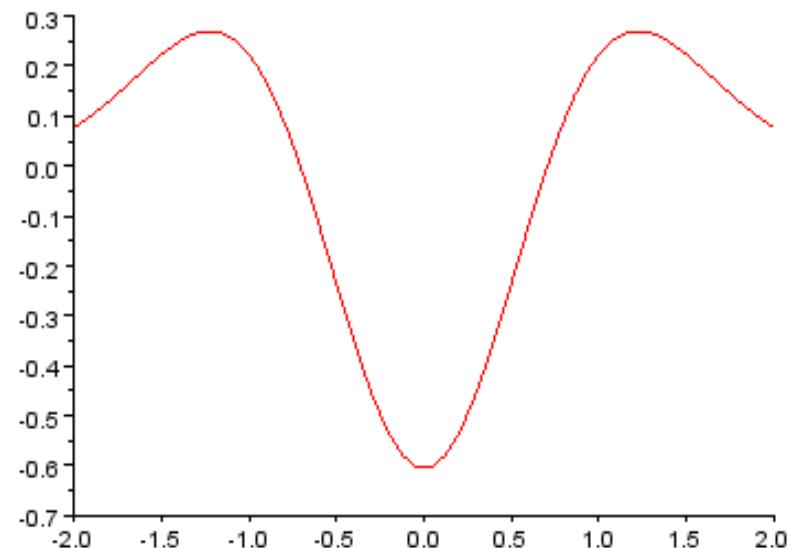
clf;
x=linspace(-2,2,15);
y=linspace(-3,3,15);
[X,Y]=meshgrid(x,y);
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);
champ(x,y,X,Y)
contour(x,y,Z,10)
```



Cutting a 3D surface

- We can see the outline of the 3D surface $z = (2*x^2 - y^2)\exp(-x^2 - 0.5*y^2)$ at a certain plane by defining the plane in case (below $y = -1$) and by returning to 2D plotting:

```
// cutting.sce
// Cut the the function
// z=(2*x^2 - y^2)exp(-x^2 - 0.5*y^2)      /
// along the plane y = -1                      /
clf;
x=linspace(-2,2,50);
y=linspace(-1,-1,0);  
[X,Y]=meshgrid(x,y);
Z=(2*x.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);
plot2d(X,Z,5)
```



Mixed 2D/3D plots (1/2): script

Scilab has its own ideas of what it should do if a `contour()` command is added to the script of a 3D plot command (`plot3d()`, `surf()`). Trial and error is needed

Question: Should `contour()` come before or after `plot3d()`?

Answer: Scilab accepts both alternatives, but with dramatically different results

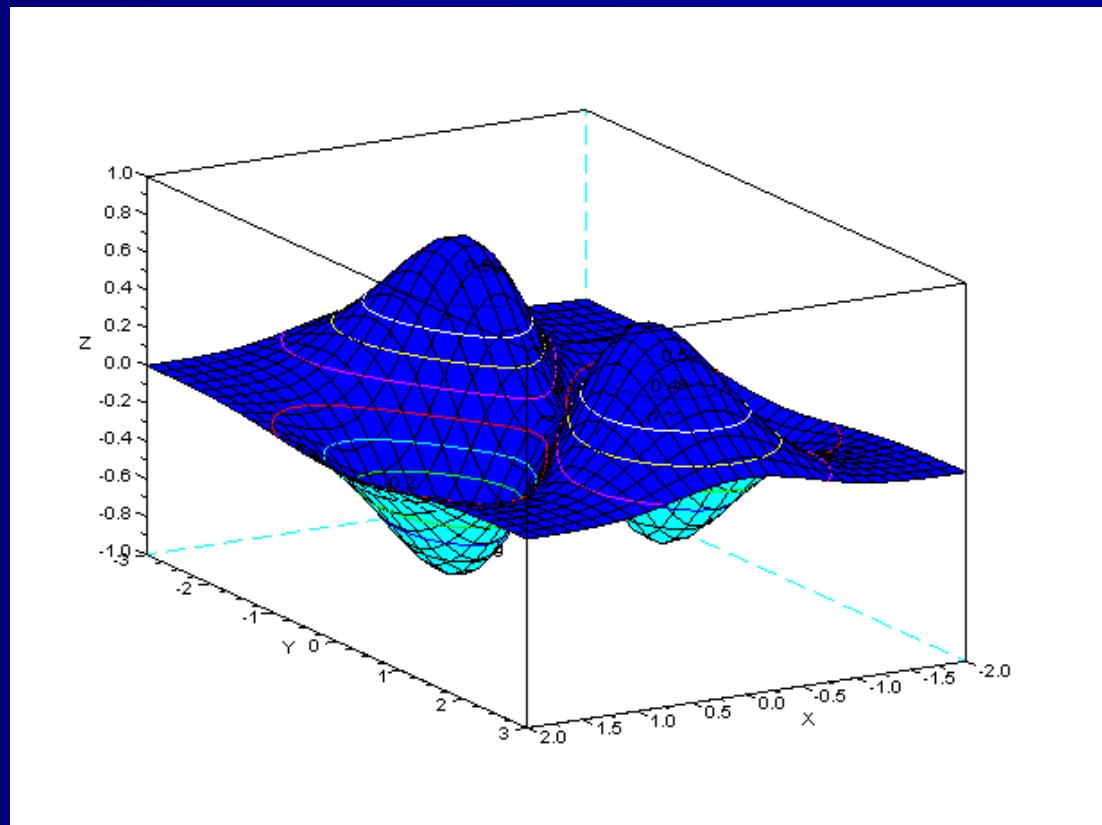
Only the first flag [] argument of `contour()` has an influence on the plot

```
// plot3d-contour.sce
// Plot the combined 3D graph and contour /
// of the function
// z=(2x^2 - y^2)exp(-x^2 - 0.5y^2), /
// for -2<=x<=2 and -3<=y<=3 /
clear,clc,clf;
x=linspace(-2,2,30);
y=linspace(-3,3,30);
[X,Y]=meshgrid(x,y);
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2); // Same as before
contour(x,y,Z,10,flag=[0,0,0]); // First flag[] argument
plot3d(x,y,Z,theta=60,alpha=80); // Turn 60 and 80 deg
```

Mixed 2D/3D plots (2/2): plot

The surface looks different from when it was plotted using `surf()`. The reason is that the **x** and **y axes are inverted** compared with the earlier case

No point in denying, there remains unsolved issues regarding the behavior of Scilab in this case



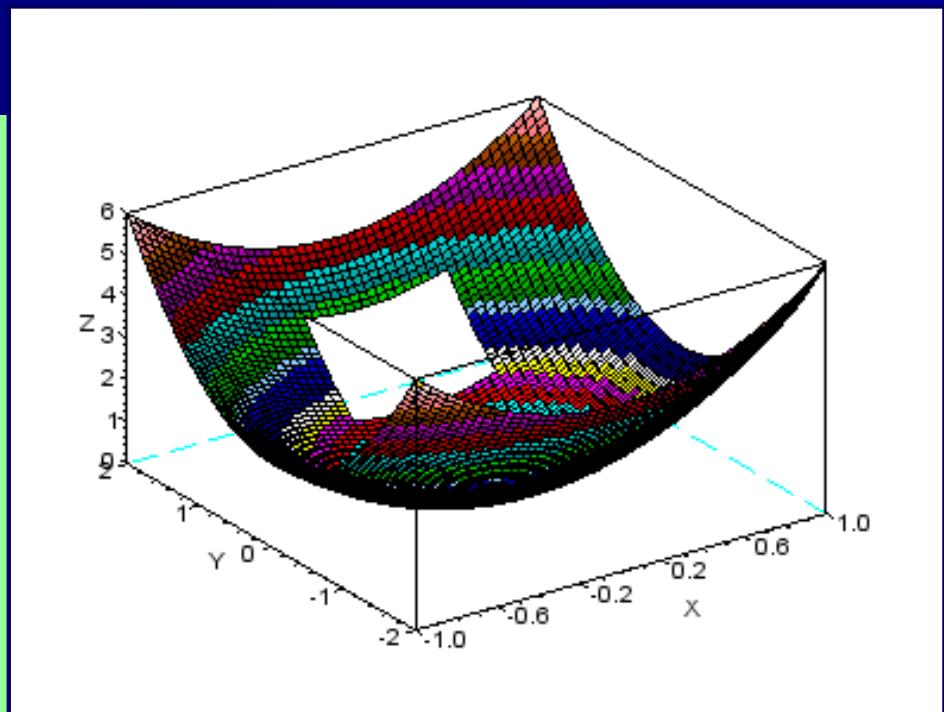
3D plot with hole

The `%nan` function allows certain z values to be excluded from a 3D plot:

```
// hole.sce
// 3D surface with a hole punched  /
// into it with the %nan command  /
// (z values not to be represented)  /
clear,clc,clf;

function z = f(x, y)
    z=2*x^2+y^2;
endfunction

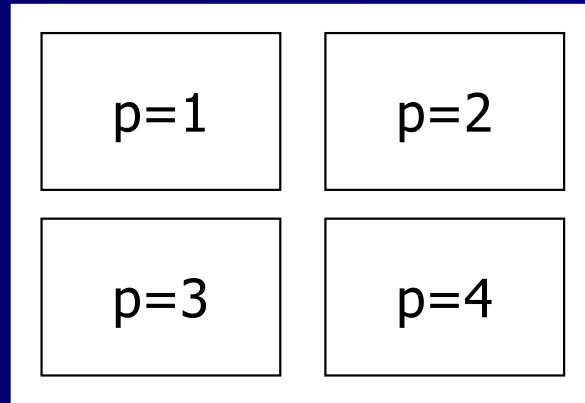
x = linspace(-1,1,50);
y = linspace(-2,2,100);
z = (feval(x,y,f))';
z(75:90,20:35) = %nan; // Evaluate function
// Definition of hole
surf(x,y,z) // Plot surface
```



There is “Polish logic” behind the z arguments that asks for trial & error to get it right

subplot()

- Subplots are a way of presenting multiple graphs on a single frame
- The function subplot(m,n,p), or (mnp), splits the Graphics Window into m rows and n columns, and the subplot in case occupies position p. In the case of four subwindows, subplot(22p), the position of p is as shown:



- We'll do it for the $z = (2*x^2 - y^2)\exp(-x^2 - 0.5*y^2)$, by fusing the four earlier cases into a single frame

subplot(): demo script

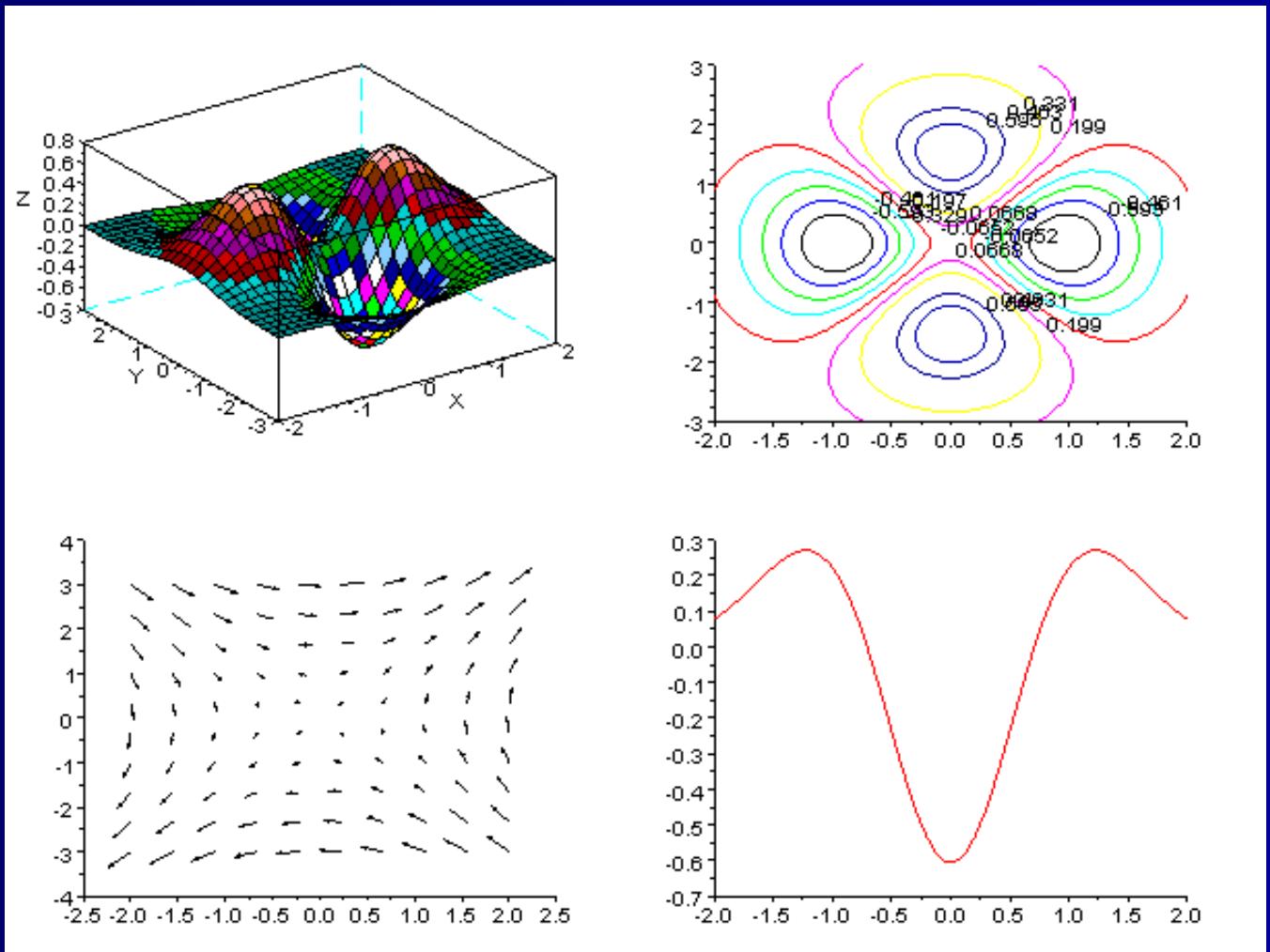
```
// subplot.sce  
  
// Presents different aspects of          /  
// the function                         /  
// z=(2x^2 - y^2)exp(-x^2 - 0.5y^2)    /  
// in four subplots                      /  
  
clear,clc,clf;  
x=linspace(-2,2,30);  
y=linspace(-3,3,30);  
[X,Y]=meshgrid(x,y);  
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);  
subplot(221)  
surf(X,Y,Z)  
  
subplot(222)  
contour(x,y,Z,10)
```

```
x=linspace(-2,2,10);  
y=linspace(-3,3,10);  
[X,Y]=meshgrid(x,y);  
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);  
subplot(223)  
champ(x,y,X,Y)  
  
x=linspace(-2,2,50);  
y=linspace(-1,1,0);  
[X,Y]=meshgrid(x,y);  
Z=(2*X.^2-Y.^2).*exp(-X.^2-0.5*Y.^2);  
subplot(224)  
plot2d(X,Z,5)
```

Note that only the plot function has been repeated for (222)

subplot(): demo plot

There is another function for subplots:
xsetech(). Check with Help for details



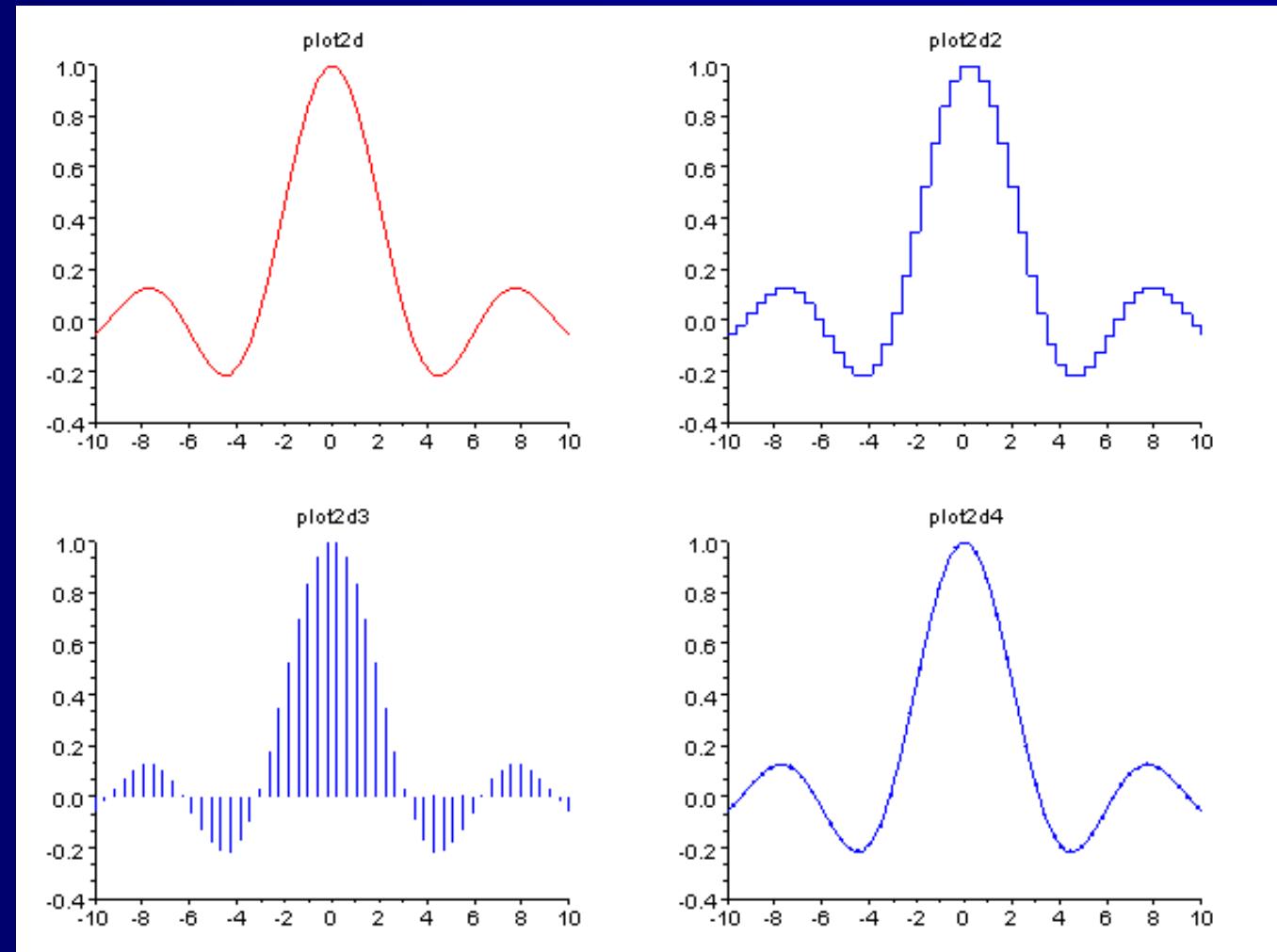
plot2d2(), plot2d3(), plot2d4(): demo, script

- The plot2d() function has three variants:
- plot2d2() for step functions
- plot2d3() for vertical bars
- plot2d4() for arrow style lines
- The effect of these plotting commands on the sinc() function is shown on the next slide

```
// plot2dx.sce  
  
// Demonstration of the basic sinc function plotted /  
// with plot2d(), plot2d2(), plot2d3(), and plot2d4() /  
  
clear,clc,clf;  
x = linspace(-10,10,50);  
  
subplot(221);  
plot2d(x,sinc(x),style=5) // Plot continuous line  
xtitle('plot2d')  
  
subplot(222);  
plot2d2(x,sinc(x),style=2) // Plot with steps  
xtitle('plot2d2')  
  
subplot(223);  
plot2d3(x,sinc(x),style=2) // Plot vertical bars  
xtitle('plot2d3')  
  
subplot(224);  
plot2d4(x,sinc(x),style=2) // Plot arrow style  
xtitle('plot2d4')
```

plot2d2(), plot2d3(), plot2d4(): demo, plot

Note: You
can still see
the obsolete
plot2d1() in
manuals.
plot2d()
should be
used instead
(In contrast,
plot3d1() is
not declared
obsolete)



Histograms: functions to create them with

- Histograms are graphical presentation—typically rectangles—of one-dimensional data
- Scilab's main function for plotting histograms is:
`histplot(x,data,opt_arguments)`
- Bar diagrams, a common form of histograms, are given by:
`bar(x,y,width,color,style)`
or, for horizontal bars:
`barh(x,y,width,color,style)`
- 3-dimensional bar diagrams can be created by the command:
`hist3d(z,opt_arguments)`
and with added x and y vectors:
`hist3d(list(z,x,y),opt_arguments)`
- Check Help for more details

Histograms: demo, script

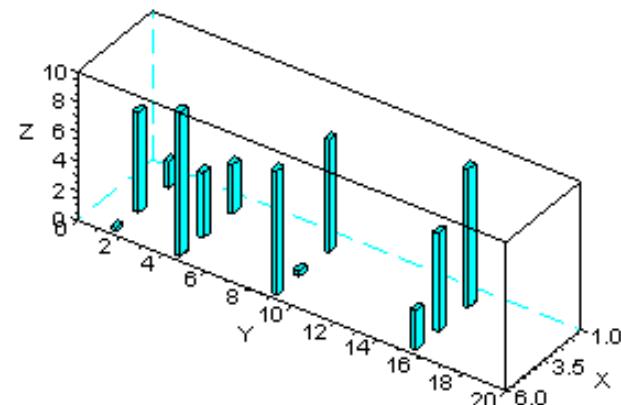
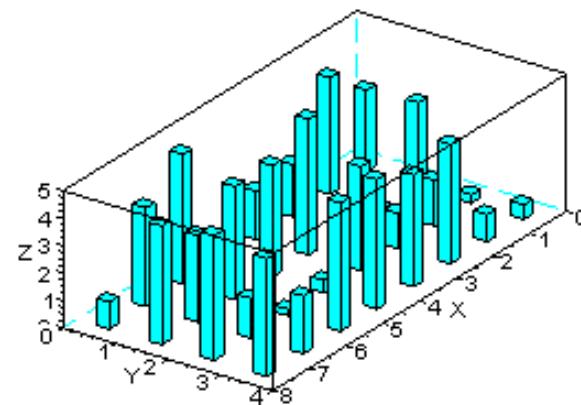
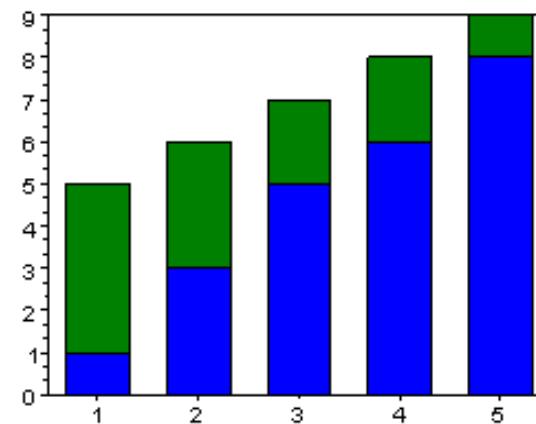
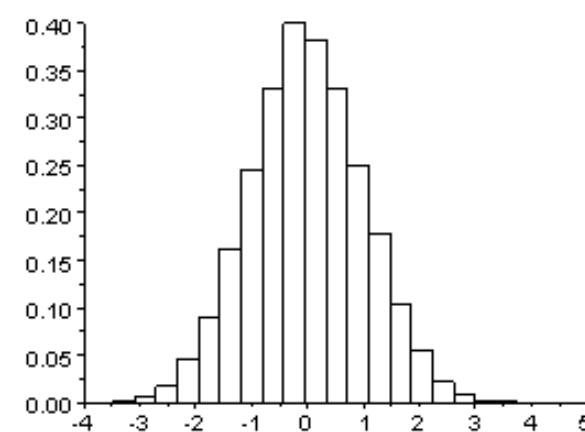
- The script(s) below are intended to demonstrate different types of histograms, presented as (22p) subplots

```
// histogram_subplot.sce  
  
// Demonstration of histogram types /  
// using subplots /  
  
clear,clc,clf;  
subplot(221)  
data=rand(1,10000,'normal');  
histplot(20,data) // Traditional histogram  
  
subplot(222)  
y=[1 3 5 6 8];  
z=[y;4 3 2 2 1]'; // Transpose necessary!  
bar(z,0.7,'stacked') // "on top of each other"
```

```
subplot(223)  
hist3d(5*rand(8,4)) // 3D histogram  
  
subplot(224)  
z=10*rand(3,4);  
x=[1 3 5 6];  
y=[1 2 7 11 20];  
hist3d(list(z,x,y)) // 3D hist, add x/y vectors
```

The list() argument defines the distribution of random z values over the x,y plane

Histograms: demo, plot



Old graphics syntax (1/2): demo, script

Scilab's graphics syntax changed with version 3.1. This demo shows the old `plot2d()` syntax for a case with three plots,

$y_1 = f(x_1)$, $y_2 = f(x_2)$ and $y_3 = f(x_3)$,

in the same frame

Note the frame definition and compare with the method used in Example 1-2

```
// multiple_plots2.sce

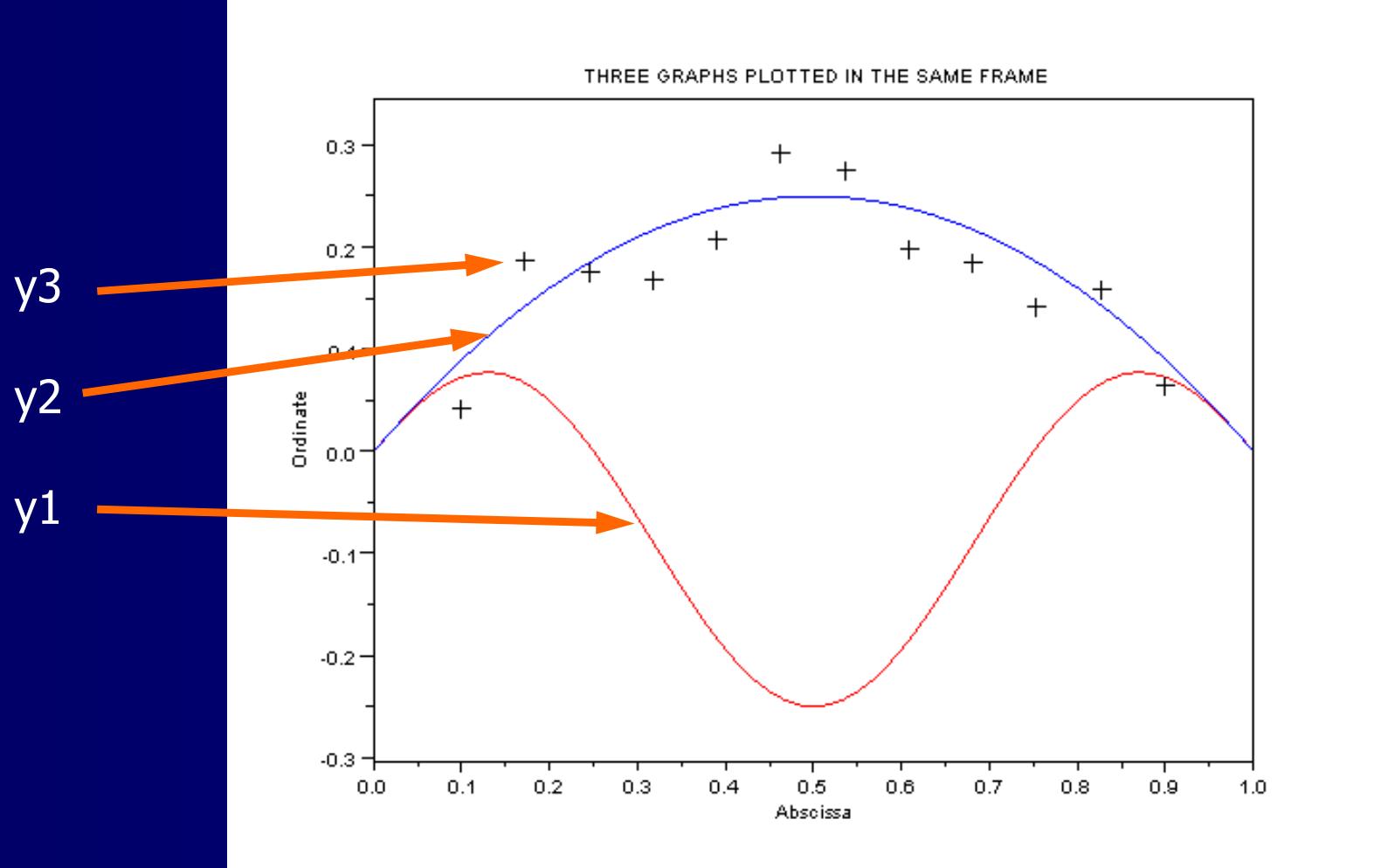
// Demonstration of a method for producing /
// three plots  $y_1 = f(x_1)$ ,  $y_2 = f(x_2)$ ,  $y_3 = f(x_3)$  /
// in the same frame. Note how the frame /
// is defined

clear,clc,clf;
x1 = linspace(0,1,61);
x2 = linspace(0,1,31);
x3 = linspace(0.1,0.9,12);
y1 = x1.* (1-x1).* cos(2*%pi*x1);      // First graph
y2 = x2.* (1-x2);                      // Second graph
y3 = x3.* (1-x3) + 0.1*(rand(x3)-0.5); // Third, as  $y_2$  with disturbance

ymin = min([y1,y2,y3]);      // Select minimum to define frame bottom
ymax = max([y1,y2,y3]);      // Select maximum to define frame top
dy = (ymax - ymin)*0.1;     // Border for min/max
rect = [0,ymin - dy,1,ymax+dy];   // Frame limits, start at 0

plot2d(x1,y1,5,"011","",rect)    // First call with frame definitions
plot2d(x2,y2,2,"000")           // Second call, only type/color (2) definition
plot2d(x3,y3,-1,"000")          // Third call, defines marks(-1)
xtitle("THREE GRAPHS PLOTTED IN THE SAME FRAME","Abscissa","Ordinate")
```

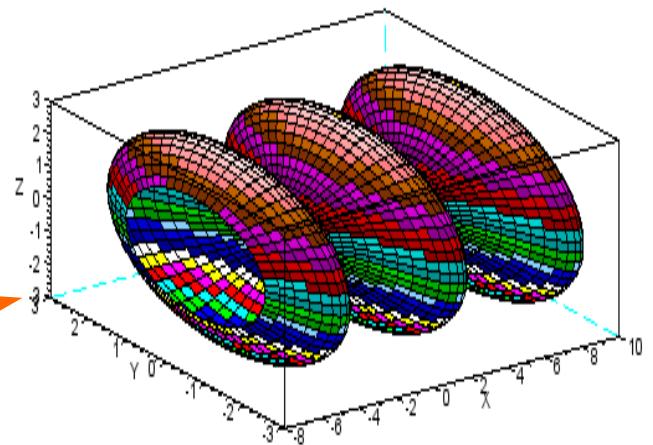
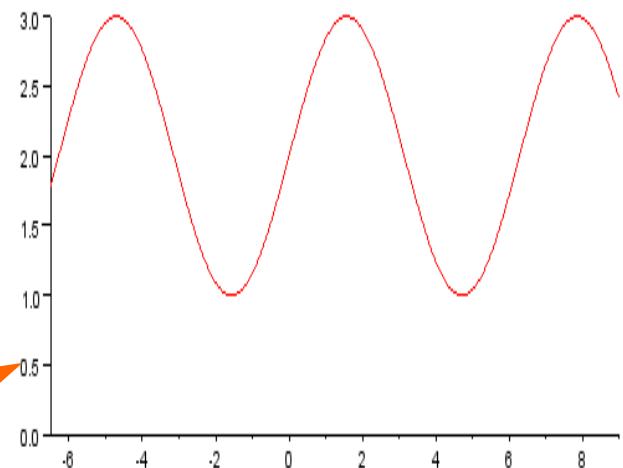
Old graphics syntax (2/2): demo, plot



Rotation surfaces

The rotation surface is created by multiplying the original function, which is redefined as $2+\sin(T)$, by $.*\sin(\text{PHI})$ and $.*\cos(\text{PHI})$

```
// rotation_surface.sce  
  
// Plot the rotation surface created by /  
// the function y=2+sin(x) as it rotates /  
// around the x-axis /  
  
clear,clc,clf;  
  
// Define function to rotate:  
//-----  
x=-10:.01:10;  
subplot(211)  
plot2d(x,2+sin(x),5,rect=[-6.5,0,9,3])  
  
// Rotate 2+sin(x) around y-axis:  
//-----  
t=linspace(-6.5,9,60);  
phi=linspace(0,2*%pi,60);  
[T,PHI]=meshgrid(t,phi); // Create mesh  
X=T;  
Y=(2+sin(T)).*sin(PHI);  
Z=(2+sin(T)).*cos(PHI);  
subplot(212)  
surf(X,Y,Z)
```



Logarithmic scale: task & script

- Plot the Bode diagram for the function

$$G(s) = \frac{100}{(s-10)(s-90)}$$

where $s = i\omega$ and the angular frequency $\omega = 0.1 \dots 1000$

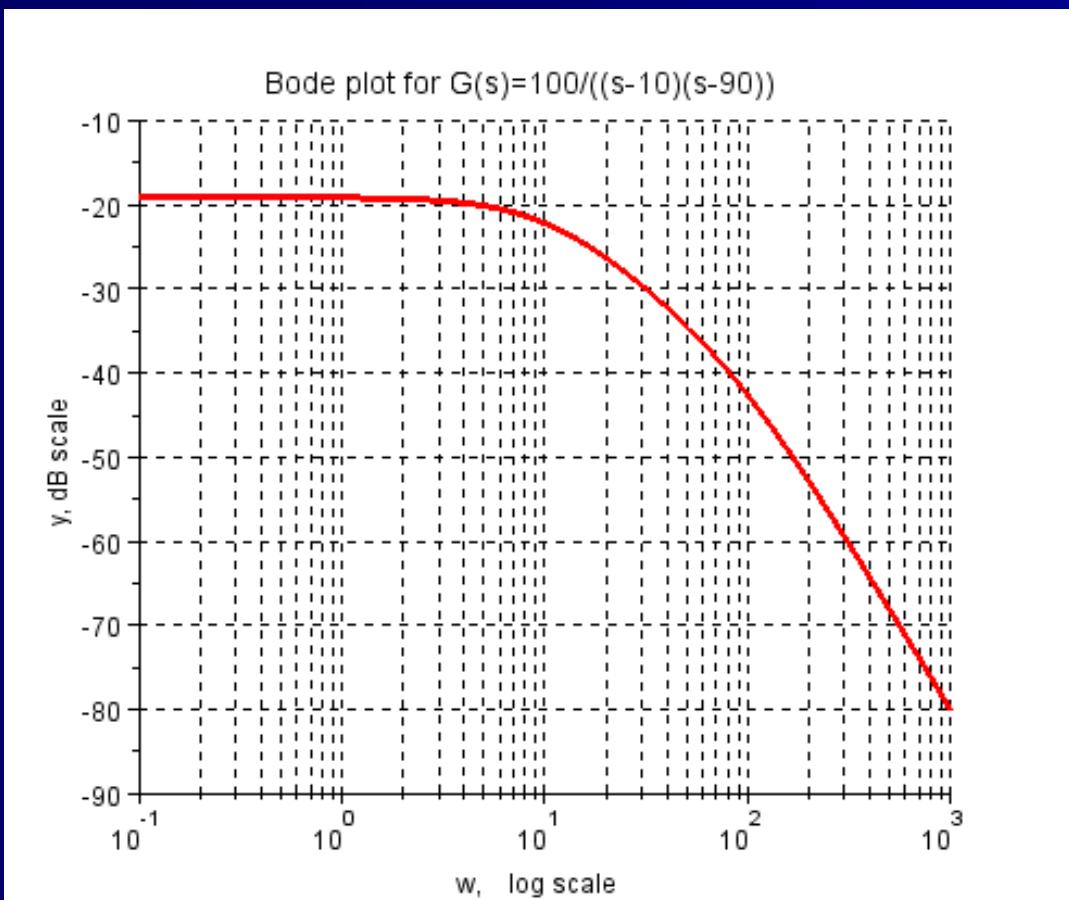
- Note double dots $100../(s-10)$ in the G command. First dot is a decimal point, then comes the Dot Operator
- Put the logarithmic ω -axis horizontally and the decibel scale $y=20(\log|G(\omega)|)$ vertically

// log_plot.sce

```
// Plot the Bode diagram for the function /  
// G(s) = 100/((s-10)(s-90)). Use the normal /  
// logarithmic x-axis and decibel scale on /  
// the y-axis /  
  
clear,clc,clf;  
w = logspace(-1,3,100); // Define log scale for w  
s = %i*w; // Define imaginary s  
G = 100../((s-10).*(s-90)); // Define G(s)  
y = 20*log10(abs(G)); // Define dB scale for y  
plot2d(w,y,5,logflag="ln") // Plot y=f(w)  
xtitle("Bode plot for G(s)=100/((s-10)(s-90))","w,...  
log scale","y, dB scale")  
xgrid() // Add grid
```

logspace(-1,3,100) = "from 10^{-1} to 10^3 in 100 logarithmically spaced increments"

Logarithmic scale: the plot



The graph has been edited after plotting

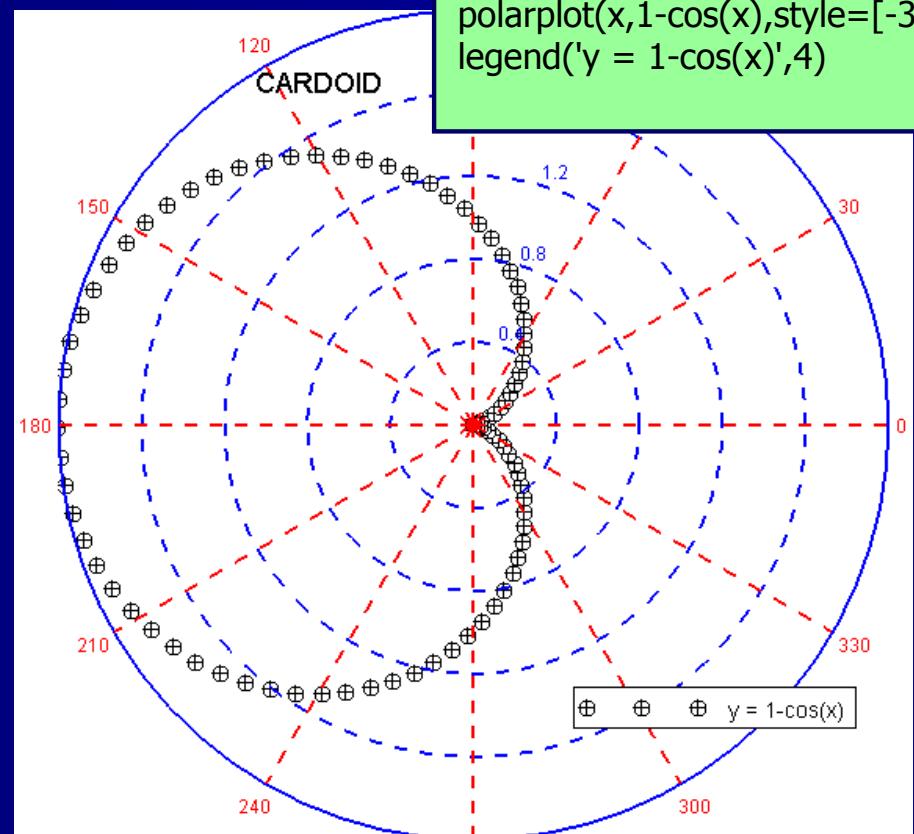
We have not before mentioned **the argument logflag** 'ln' in `plot2d()`. Change 'ln' to 'nn' ('ll' is not possible here) and see how the plot changes (n=normal, l=logarithmic)

Note: Scilab has a special **function for Bode plots**, `bode()`. See [Example 3-1](#)

Polar coordinates

- Polar coordinates are used frequently in some areas of engineering, e.g. to present antenna lobe diagrams
- Plotting in polar coordinates is done by the command `polarplot()`
- The demo shows the simple script for a cardioid, $y = 1 - \cos(x)$, and its plot
- Note the markers related to the `style= [-3]` argument
- The plot has been edited, which is time consuming for polar plots

```
//cardioid.sce
// The script plots the cardioid   /
// r = 1 - cos(x), for x = 0...2pi  /
clear,clc,clf;
x = 0:0.07:2*pi;
polarplot(x,1-cos(x),style=[-3])
legend('y = 1-cos(x)',4)
```



Exporting plots

- Scilab plots can be exported in various picture formats (PNG, SVG, GIF, Bitmap, etc.) for use in documents
- To export, Click File/Export to... in the Graphics Window and select the target file as well as the wished format
- An alternative way is to use the `xs2*()` function which for PNG takes the form

```
xs2png(window_number, file_name);
```

- The following **vectorial** and **bitmap** formats are possible:

<code>xs2png()</code>	export to PNG
<code>xs2pdf()</code>	export to PDF
<code>xs2svg()</code>	export to SVG
<code>xs2eps()</code>	export to EPS
<code>xs2ps()</code>	export to Postscript
<code>xs2emf()</code>	export to EMF (Windows)

<code>xs2fig()</code>	export to FIG
<code>xs2gif()</code>	export to GIF
<code>xs2jpg()</code>	export to JPG
<code>xs2bmp()</code>	export to BMP
<code>xs2ppm()</code>	export to PPM

Handles (1/12): introduction*

- Handles are a thorny subject to Scilab newbies. Existing texts give only an incoherent treatment of the topic. The user is left with the option "try and cry"
- We shall limit this discussion to the most essential handle properties, aiming at gaining a basic understanding of how plots are edited with handles
- It may help to view handles as an alternative to the Figure Editor that we already have used. The idea is the same in both
- The Help Browser discusses the subject under the heading `graphics_entites`. Check also `object_editor`

*) Recall the introduction to handles in [Chapter 2](#). Handles were already used in [Example 2-3](#) and when discussing [polylines](#). This discussion is based on Kubitzki: *Grafik, Eigenschaften verwalten in Scilab*, section 2.4.3 in Campbell et al., and Steer: *Scilab Graphics*, 2007.

Handles (2/12): introduction*

- The Graphics Window is built as a **hierarchy** of objects. See the hierarchic tree presented on the next slide, which also gives typical commands for each entity
- The topmost object in the window is called **Figure**. We use the function `gcf()`, *get current figure*, to influence the window as it pops up on the screen. This is done with the handle `f = gcf()`
- Figure has a child called **Axes**, which in turn has several **children**, and these again may have own children. Axes is called by the function `gca()`, *get current axes*. The handle in case is `a = gca()`, but the alternative `a = f.children` also works
- Pay attention to **Compound**, which has the important children **Label** and **Polyline**. The latter refers to the actual graph that we plot
- Figure can have other children beside Axes. These are created by Scilab when we use certain commands

Handles (3/12): basic graphics hierarchy

Compare with the structure of the Graphics Editor's Object Browser that we have used to edit plots

Figure

```
h = gcf(); // Get figure handle
```

Axes

```
a = gca(); // Get axes handle or  
a = h.children;
```

x_label

```
x = a.x_label // Get x_label handle
```

y_label

```
y = a.y_label // Get y_label handle
```

z_label

```
z = a.z_label // Get z_label handle (3D graphics)
```

title

```
t = a.title // Get title handle
```

compound

```
c = a.children // Get compound handle
```

legend

```
leg = c.children(1) // Get legend handle (if any)
```

polyline

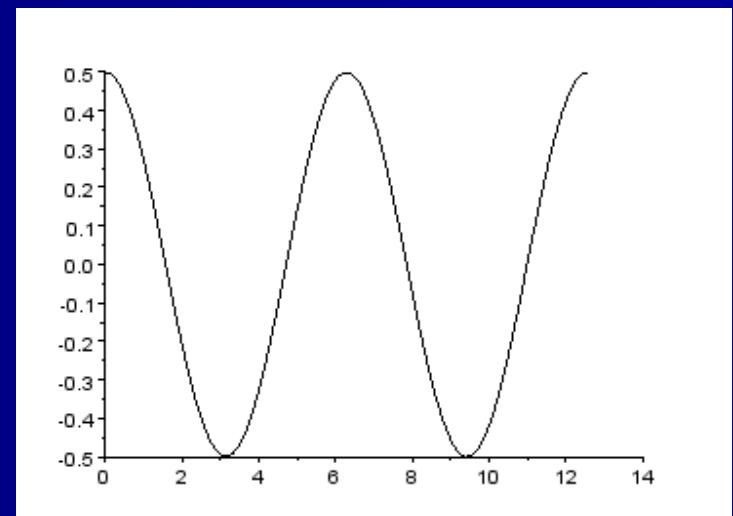
```
pol = c.children(2) // Get polyline handle  
(first graph in plot)
```

etc.

Handles (4/12): demo, starting point

As a first exercise, let's start from the script that was used in the introduction to handles in Chapter 2:

```
// handles_demo1.sce  
  
// Basic script to demonstrate handles /  
  
x = linspace(0, 4*pi, 100);  
plot2d(x, 0.5*cos(x))
```



Lessons learned:

- 1) You have to be systematic when working with handles
- 2) The existing literature is not always correct. For instance, the method suggested by Steer for changing axes ticks & marks simply does not work (took me hours to figure out)

Handles (5/12): demo, behind the scene

```
-->gca()
ans =
Handle of type "Axes" with properties:
=====
parent: Figure
children: "Compound"
...
```

When we call up the Axes handle on the Console, it turns out to be really long. On the top of the list we find that **Axes** has a child, **Compound**

A check with `gce()` reveals that **Compound** in turn has a child, **Polyline**. This matches the hierarchy that we have seen on the Figure Editor

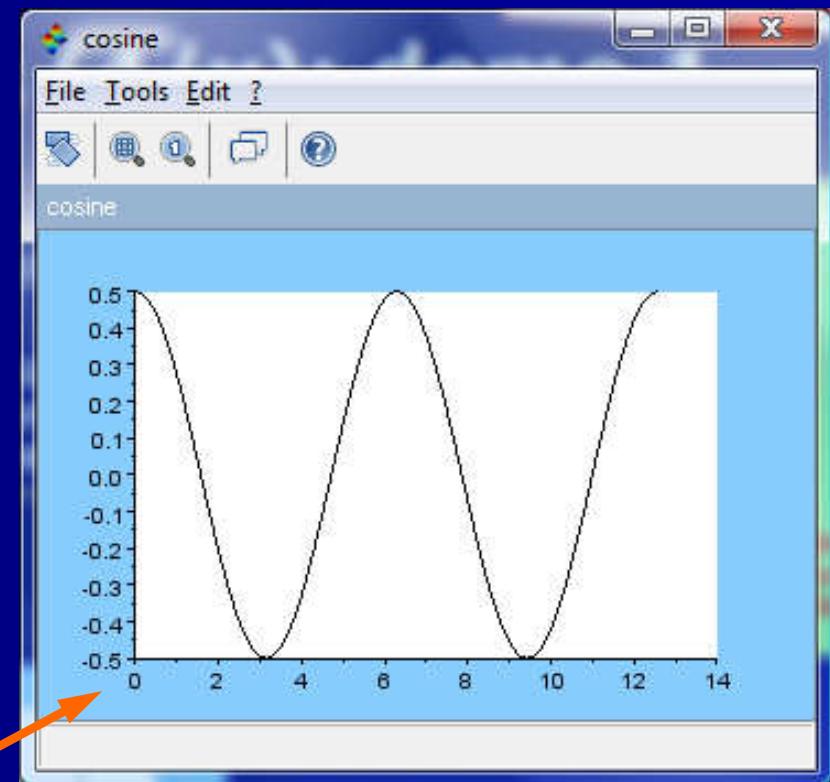
```
-->gce()
ans =
Handle of type "Compound" with properties:
=====
parent: Axes
children: "Polyline"
visible = "on"
user_data = []
```

Handles (6/12): demo, step 1

We first define some changes to the window:

- Adjust the window size
- Add background color
- Give the window a name

```
// handles_demo1.sce  
  
// Basic script to demonstrate handles /  
  
clear,clc,clf;  
  
x = linspace(0, 4*%pi, 100);  
plot2d(x, 0.5*cos(x))  
  
f=gcf(); // Get Figure (window) handle  
f.figure_size = [500,400]; // Adjust window size  
f.background = 12; // Add background color  
f.figure_name= "cosine"; // Name window
```



Check for details under figure_properties in the Help Browser

Handles (7/12): demo, step 2

In this step we

- Move along the hierarchy ladder
- Edit the plot

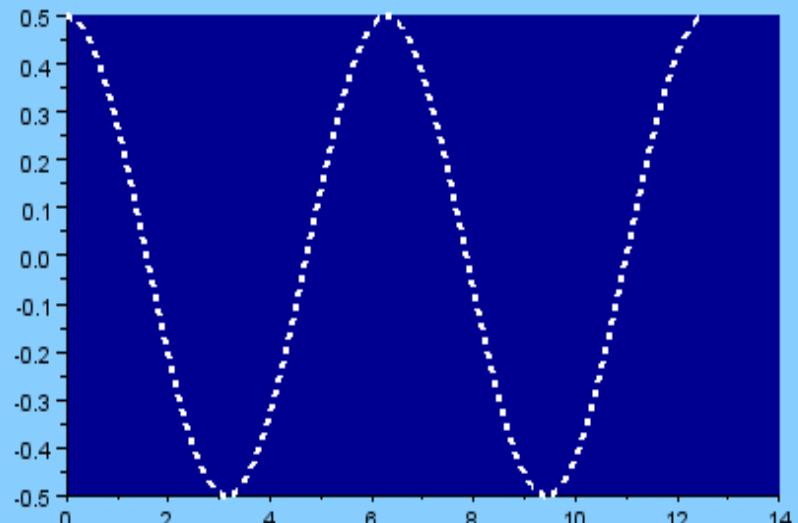
by adding these lines to the
script (you can bypass the p1
definition stage and write
c.children.foreground ... etc.)

Note that we move down the
hierarchy ladder: Figure ->
Axes -> Compound -> Polyline

Check for details under
polyline_properties in
the Help Browser

Change p1.line_style to
p1.polyline_style to get a
different plot

```
a=gca();           // Get Axes handle
a.background = 9;  // Change background
c=a.children;     // Get compound handle
p1=c.children;   // Get polyline (plot) handle
p1.foreground = 8; // Change line color
p1.line_style = 7; // Change line style
p1.thickness = 3; // Line thickness
```



Handles (8/12): demo, step 3

As shown earlier, the Entity handle was quite empty. We need to add labels that can be edited. For that we add the following command to the script:

```
xtitle('COSINE PLOT',...  
      'X-axis','Y-axis');
```

And now the Entity handle has undergone a dramatic change (this is only the beginning of the list) →

```
-->gce()  
ans =  
  
Handle of type "Axes" with properties:  
=====  
parent: Figure  
children: "Compound"  
  
visible = "on"  
axes_visible = ["on","on","on"]  
axes_reverse = ["off","off","off"]  
grid = [-1,-1]  
grid_position = "background"  
x_location = "bottom"  
y_location = "left"  
title: "Label"  
x_label: "Label"  
y_label: "Label"  
z_label: "Label"  
auto_ticks = ["on","on","on"]  
x_ticks.locations = [0;2;4;6;8;10;12;14]  
y_ticks.locations = matrix 11x1  
z_ticks.locations = []  
x_ticks.labels = ["0";"2";"4";"6";"8";"10";"12";"14"]  
y_ticks.labels = matrix 11x1  
z_ticks.labels = []
```

Handles (9/12): demo, step 4

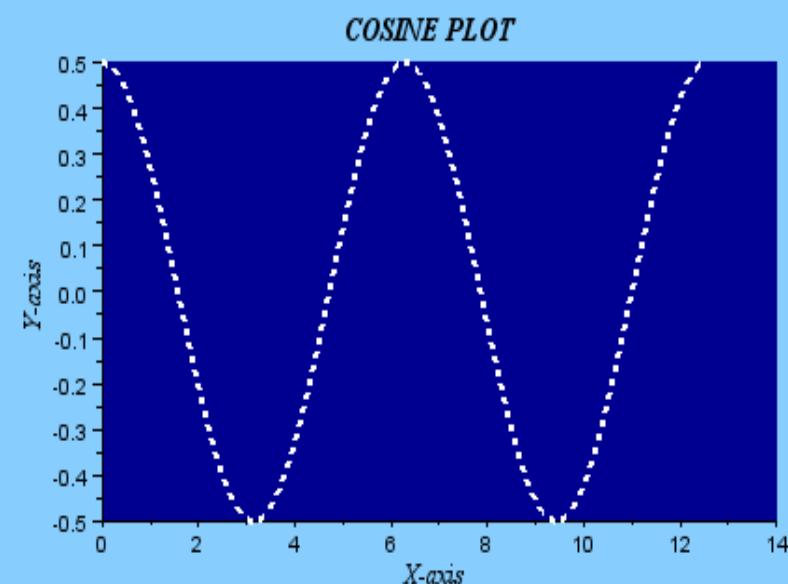
Title and axis labels have been added, the next step is to edit them

In each case we must first call the respective handle (or skip this stage by writing editing commands in the form `a.title.font_style ... etc.`), then edit the handle properties

Check for details under
label_properties in
the Help Browser

The plot isn't exactly a beauty, but we'll add a grid and edit axes ticks & marks

```
xtitle('COSINE PLOT','X-axis','Y-axis'); // Add title & labels
t=a.title;
t.font_style = 5; // Get title handle
t.font_size = 3; // Times bold, italic
xL=a.x_label;
xL.font_style = 5; // Increase font size
xL.font_size = 2; // Get x_label handle
yL=a.y_label;
yL.font_style = 5; // Times bold, italic
yL.font_size = 2; // Increase font size
// Get y_label handle
// Increase font size
// Get y_label handle
// Times bold, italic
// Increase font size
```



Handles (10/12): demo, step 5

Add grid



```
xgrid(5); // Add grid
```

Change x-axis ticks & marks



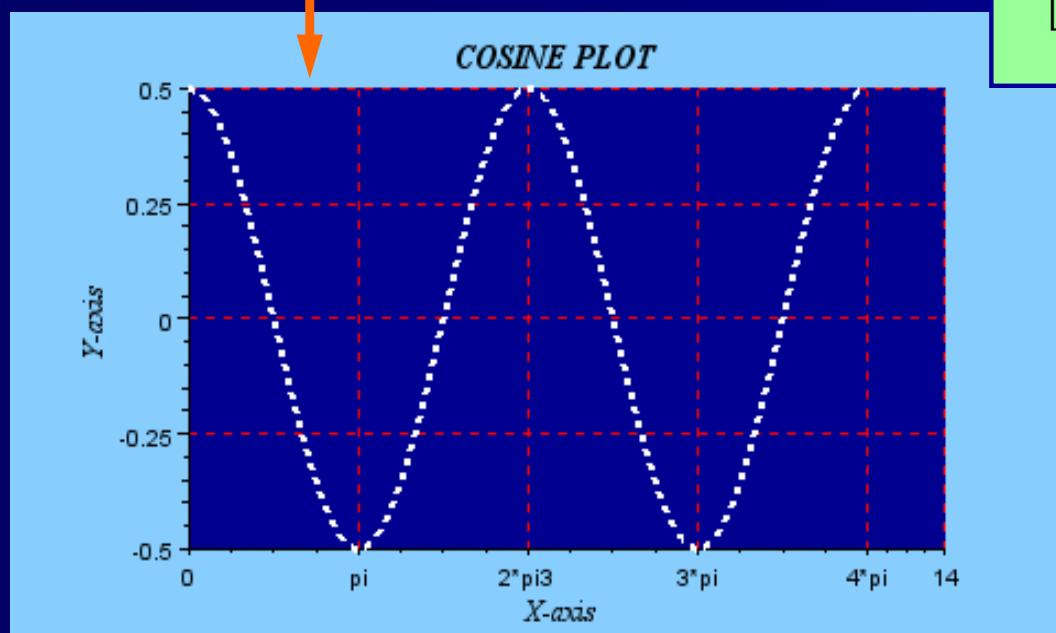
// Change x/y ticks & marks:

```
a.x_ticks = tlist(['ticks','locations','labels'],...  
[0,%pi,2*%pi,3*%pi,4*%pi,14],...  
['0','pi','2*pi3','3*pi','4*pi','14']);  
a.y_ticks = tlist(['ticks','locations','labels'],...  
[-0.5,-0.25,0,0.25,0.5],...  
['-0.5','-0.25','0','0.25','0.5']);
```

Change y-axis ticks & marks



Final plot:

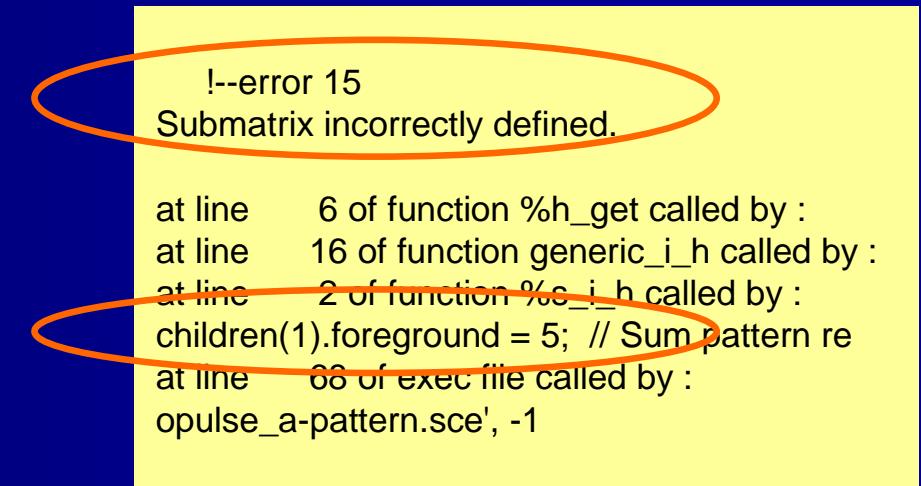


Note: There were problems with ticks & marks. Only the presented syntax worked

Handles (11/12): comments (1/2)

- With handles we must observe the order of Scilab commands. For instance, a script of the following type causes an error message from Scilab:

```
plot(...);  
legend("alpha", "beta");  
.....  
a=gca();  
a.children(1).foreground=5;  
.....
```



The image shows a yellow rectangular window with a black border, representing a Scilab error message. The text inside is as follows:

```
!--error 15  
Submatrix incorrectly defined.  
  
at line    6 of function %h_get called by :  
at line   16 of function generic_i_h called by :  
at line   2 of function %c_i_h called by :  
children(1).foreground = 5; // Sum pattern re  
at line   66 of exec file called by :  
opulse_a-pattern.sce', -1
```

Two parts of the text are circled with orange ellipses: "Submatrix incorrectly defined." and the entire stack trace starting with "at line".

- The error message confuses by referring to a submatrix
- The real reason is that we try to change the color of the plotted graph after the legend was declared. Scilab cannot jump back to the legend and change it. **The legend command has to come after related handle declarations.** But there exceptions....

Handles (12/12): comments (2/2)

- Handle commands are valid only specific levels (Figure, Axes, Entity, etc.). Help/axes_properties gives some hints but mostly you try & cry and get error messages



--error 999

This object has no auto_clear property.
at line 4 of function generic_i_h called by :
at line 2 of function %c_i_h called by :
e2.auto_clear = "on"; at line
71 of exec file called by :
examples\planet_moon1.sce', -1

- Scilab has a **hidden agenda** when it comes to handles. For instance, the polyline numbering works in quite strange ways...
- Visual edition with handles undoubtedly improves the look of a figure, but is the method an “overkill?” The amount of code needed to edit the plot can be larger than used to create the actual plot
- We should consider that **time = money**. The important thing is to come up with a script that is “**fit for purpose**.” The rest is luxury
- It is possible to change Scilab’s default settings, but information on the subject is hard to come by (Kubitzki discusses it briefly)

Polyline (1/3): xpoly(), script

- This is an attempt to see how well we can work without ordinary plot functions
- Compare with the xpoly() example given in Help and which uses the obsolete xset() function
- The xpoly() function draws a polyline; the polyline is a closed polygon if the numeric argument of xpoly() is >0
- Note the e.parent.... definition that refers one step up in the hierarchy, to Figure
- With e.children.... we move one step down in the hierarchy

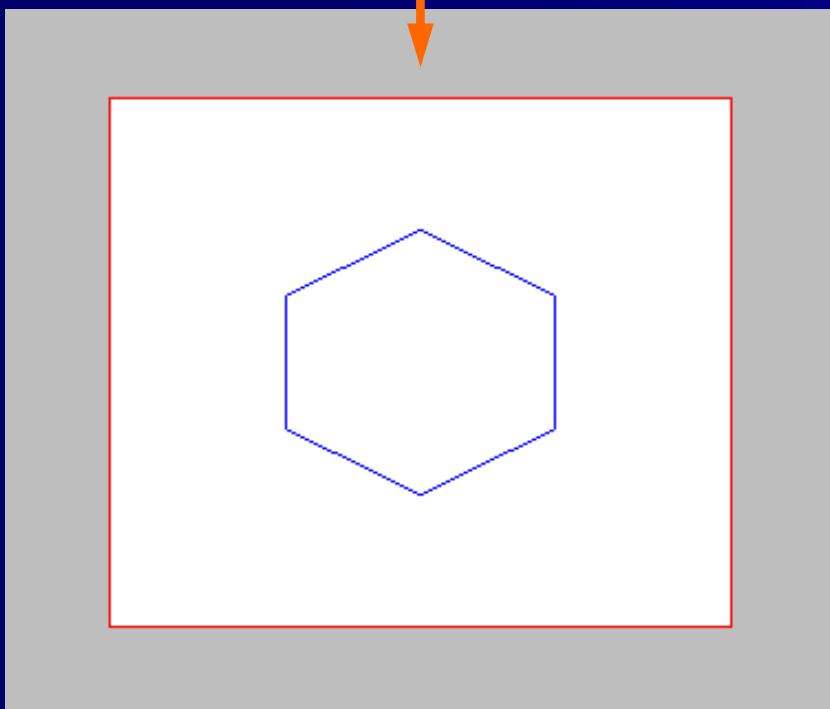
```
// xpoly.sce
// Attempt to plot a hexagon with xpoly() & edit      /
// with handles. Causes erroneous behavior in      /
// Scilab. The script must be closed to get rid of      /
// the grey background color                         /
clear,clc,clf;

x = sin(2*%pi*(0:5)/6);           // Define hexagon
y = cos(2*%pi*(0:5)/6);           // - " -
xpoly(x,y,'lines',1);            // Draw polygon

e=gca();                          // Get Axes handle
e.parent.background =...          // Get Figure handle
color('grey');                   // & set background
e.box='on';                       // Switch frame on
e.foreground=5;                  // Red frame color
e.data_bounds=[-2,-2;2,2];        // Frame size
e.children.foreground = 2;         // Blue graph color
```

Polylines (2/3): xpoly(), plot & discussion

And this is the polygon that we have created:



The unedited hexagon can also be drawn with the following script:

```
x = sin(2*pi*(0:6)/6);
y = cos(2*pi*(0:6)/6);
plot2d(x,y,strf='011',rect=[-2,-2,2,2])
```

It is left open if we do a small change to the x/y arguments:

```
x = sin(2*pi*(0:5)/6);
y = cos(2*pi*(0:5)/6);
plot2d(x,y,strf='011',rect=[-2,-2,2,2])
```

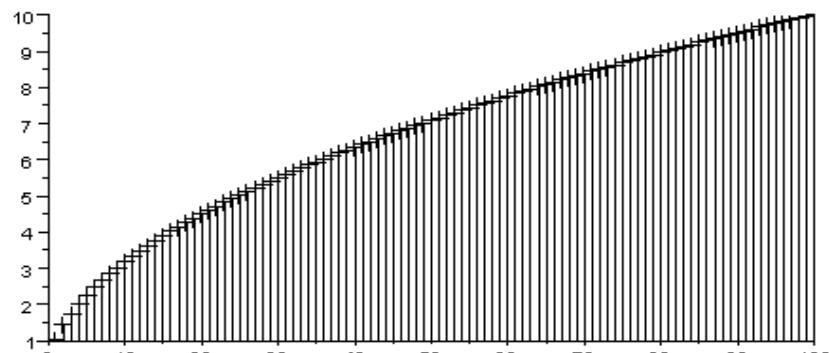
Polyline (3/3): xpoly(), lessons learned

- Scilab showed unexpected behavior with this script:
 - The background color could **turn black** with the command `e=gce(); e.parent.background=34`. The `gcf()` handle revealed that the setting **was** `background=-2` and the handle command had no effect. The **definition** `color('grey')` feels more stable than its numeric counterpart
 - The Graphics Window did not always change when the script was changed and executed. The background stayed grey even if `e.parent.background=color('grey')` was deleted. When shifting between two scripts on the Editor, the background color was exported to the second script. The script had to be closed to get rid of the gray color
 - I found **no way to add ticks & marks** to the box. The Axes handle `gca()` showed them as defined, but for some reason they are suppressed. Help/axes_properties gives no explanation
- **Lessons learned:** Do not exaggerate the extent to which you trade ordinary plot functions (`plot()`, `plot2d()`) for handle commands

Programming pitfalls: don't forget clf;

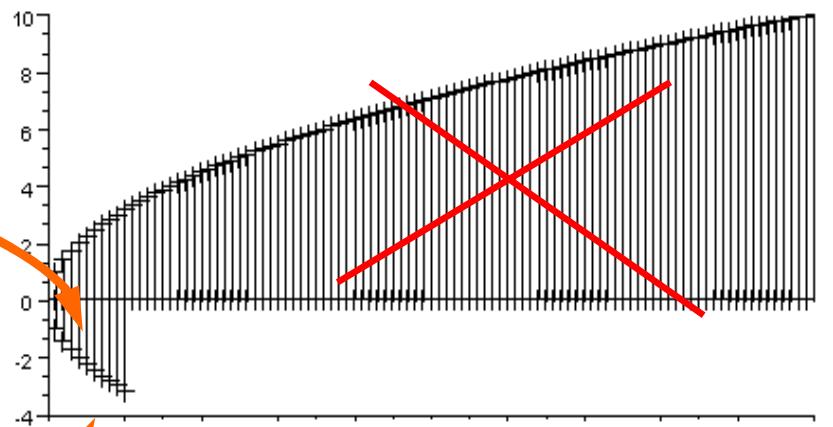
Change
and
rerun

```
// ptifalls_1.sce  
  
// Clear commands /  
  
K = 100; a = 0; b = 0;  
x = zeros(1,K); y = zeros(1,K);  
for k = 1:K  
    x(k) = a+k;  
    y(k) = b+k^(0.5);  
end  
plot2d3(x,y,style=-1)
```



```
// ptifalls_1.sce  
  
// Clear commands /  
  
K = 10; a = 0; b = 0;  
x = zeros(1,K); y = zeros(1,K);  
for k = 1:K  
    x(k) = a+k;  
    y(k) = b-k^(0.5);  
end  
plot2d3(x,y,style=-1)
```

Plots are
superposed
without the
clf command

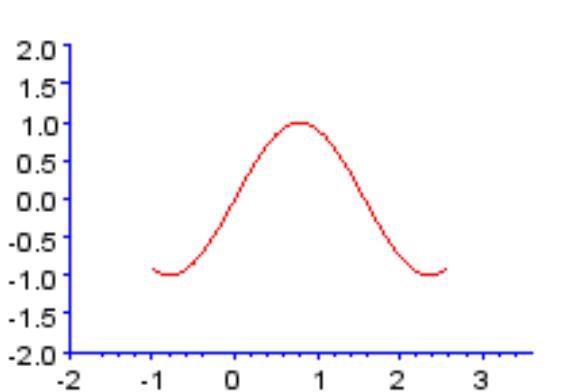


What to do with xset()?

- Examples in Scilab literature—and in blog discussions—frequently use the function `xset()`. It's **handy but obsolete**, so what should we do about it?
- The Help Browser recommends using the graphic objects representation instead (`set()`, `get()`, handle commands)
- Below are examples of how to substitute `xset()`. Note that `xset()` **operates on the current Entity level** and gives the blue axes color, not the red graph

1. Initial script
with `xset()`

```
x=-1:0.1:2.6
plot2d(x,sin(2*x),5,rect=[-2,-2,3.6,2])
xset("color",2)
```



2. Modified script
with Axes handle
command

```
x=-1:0.1:2.6
plot2d(x,sin(2*x),5,rect=[-2,-2,3.6,2])
a=gca();
a.foreground=2
```

3. Modified script
with `set()` and
handle argument

```
x=-1:0.1:2.6
plot2d(x,sin(2*x),5,rect=[-2,-2,3.6,2])
a=gca();
set(a,"foreground",2)
```

xset(): a practical case

Example 6-2 (last set of examples) is adapted from Pinçon. The original contained obsolete commands, in particular xset(). I substituted the xset() commands with the following handle commands:

xset() command	Handle graphics command
	h=gca()
xset('background',1) black	h.background = 1
xset('color',2) blue fill	h.foreground = 2
xset('thickness',3) line thickness	h.thickness = 3
xset('color',5) red border	h.foreground = 5

But frankly, it can be a pain and you want to throw the computer out the window. If so, **check if the gca() handle has any children at all...**

Flawed error messages

- Scilab's debugger shows strengths and flaws in the error messages that invariably pop up before one's plot commands are right
- Here are two error messages that I have got:

```
plot2d(x1,y1,style=5) // Function
!--error 999 plot2d:
first and second arguments have incompatible dimensions.
at line    16 of exec file called by :
exec("H:/Dr.EW/Writings/Scilab examples/dervative_2.sce");
while executing a callback
```

The real problem was that I had not used the Dot Operator in the equation for y1

```
clc();
!--error 13
Redefining permanent variable.

while executing a callback
```

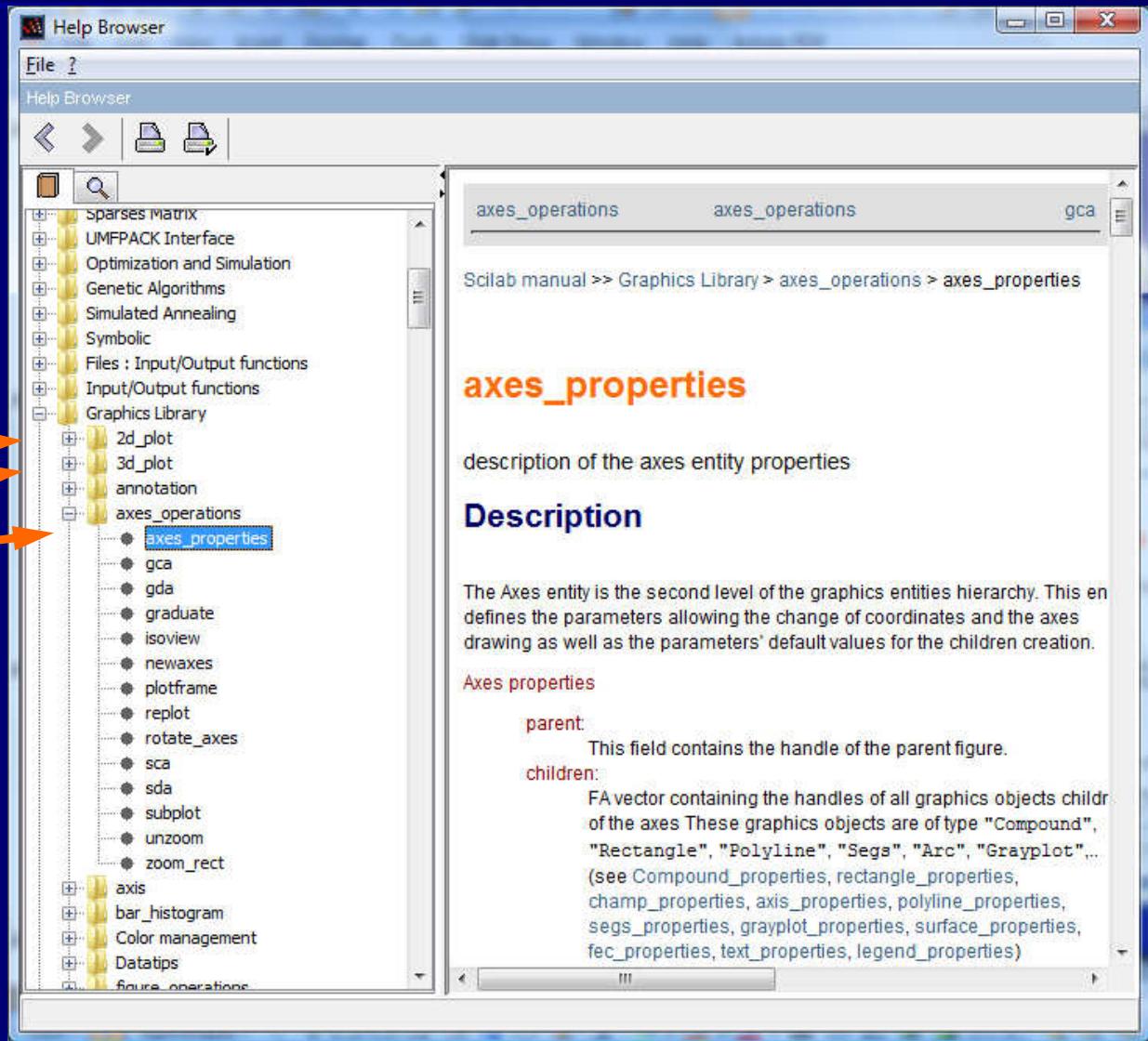
Bogus warning, Scilab crashed and had to be reloaded to erase the Console. This occurred when I used two deff() functions in tandem

More info on plotting

In the Help Browser, Click:
Graphics Library,
and under it you find
info on e.g.

- 2D plots
- 3D plots
- axes_operations/
axes_properties
etc.

Now at least is the
time to get familiar
with the Help
Browser



8. Examples, Set 3

On plotting, handles, control
engineering, and user defined
functions



[Return to Contents](#)

Example 3-1: More control engineering plots

- Example 2-3 and the log scale demo were typical control engineering tasks. Recall also the pages on polynomials in Chapter 3
- Here we'll look at examples with Bode and Nyquist plots, Nichols chart (Black's diagram), and an Evans root locus plot
- The first cases use the second-order transfer functions

$$G_2(s) = \frac{s^2 + 20s + 100}{s^2 + 6s + 100} * \frac{s^2 + 3s + 220}{s^2 + 25s + 225}$$

- The Evans root locus is plotted for

$$G_4(s) = 352 * \frac{5 + s}{2000s^2 + 200s^3 + 25s^4 + s^5}$$

Ex 3-1: script

- The first two gain equations are given as ordinary polynomial expressions
- The third gain equation, to be used in plotting Evans root loci, is defined through its roots
- The Bode plot is only for the gain, later the alternative bode() will be demonstrated
- Scilab talks about Black's diagram rather than Nichols chart. Example 3-2 highlights the difference between the two

```
// control_eng.sce

// Plot Bode, Nyquist, Nichols & Black's, /
// and Evans for defined equations      /

clear,clc,clf;
// Definition of systems:
//-----
s = poly(0,'s'); // Polynomial seed
Gain1 = syslin('c',(s^2+20*s+100)/(s^2+6*s+100));
Gain2 = Gain1*syslin('c',(s^2+3*s+220)/(s^2+25*s+225));
Gain3 = poly(-5,'s')/poly([0,0,2000,200,25,1],'s','c');
Gain4 = syslin('c',352*Gain3);

// Bode plot:
//-----
subplot(221)
gainplot([Gain2;Gain1],0.01,100) // Magnitude plot

// Nyquist plot:
//-----
subplot(222)
nyquist([Gain2;Gain1]) // Plot with Re and Im axes

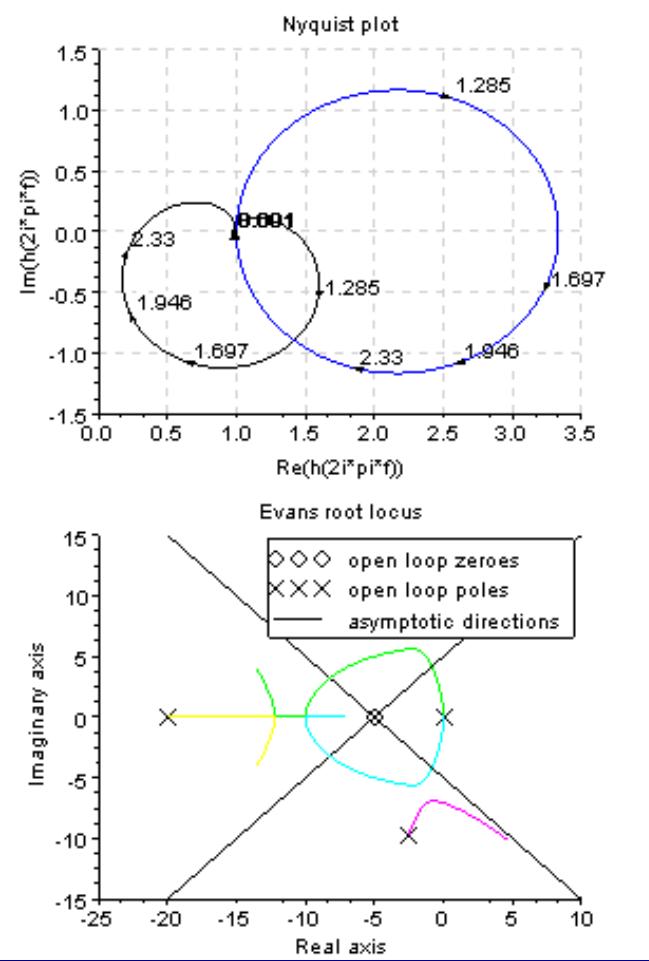
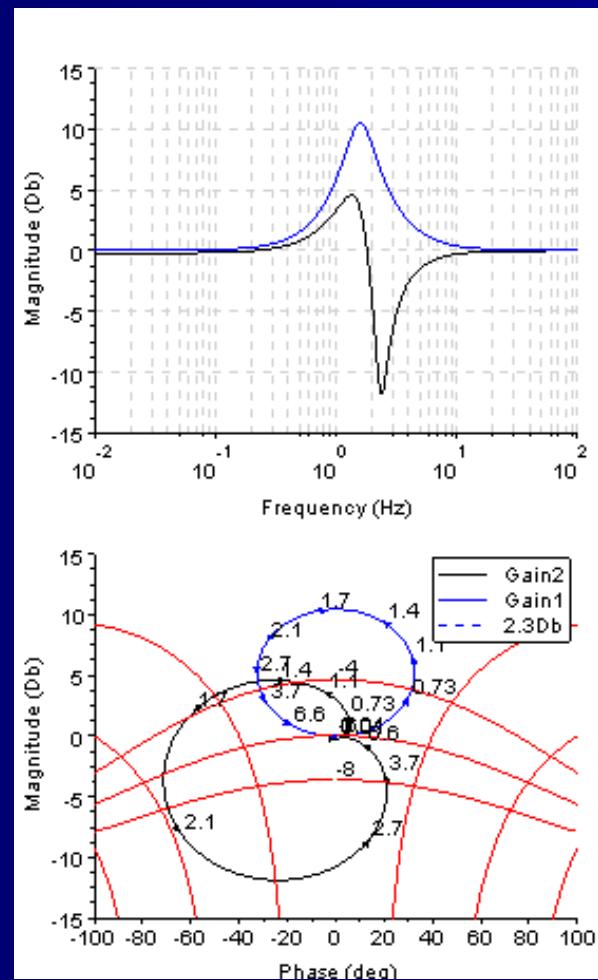
// Nichols chart (Black's diagram + iso-plots):
//-----
subplot(223)
black([Gain2;Gain1],0.01,100,['Gain2';'Gain1'])
chart([-8 -6 -4],[20 50 80],list(1,0,5))

// Evans root locus:
//-----
subplot(224)
evans(Gain4,100) // Evans root locus for sys4
```

Ex 3-1: plot

The plot has not been edited, everything shown is the result of the script. Note the red iso-curves on the Bode-Nichols subplot

The next slide looks at how alternative Bode plot commands operate



Ex 3-1: alternative Bode plot functions

```
// bode_comparison.sce

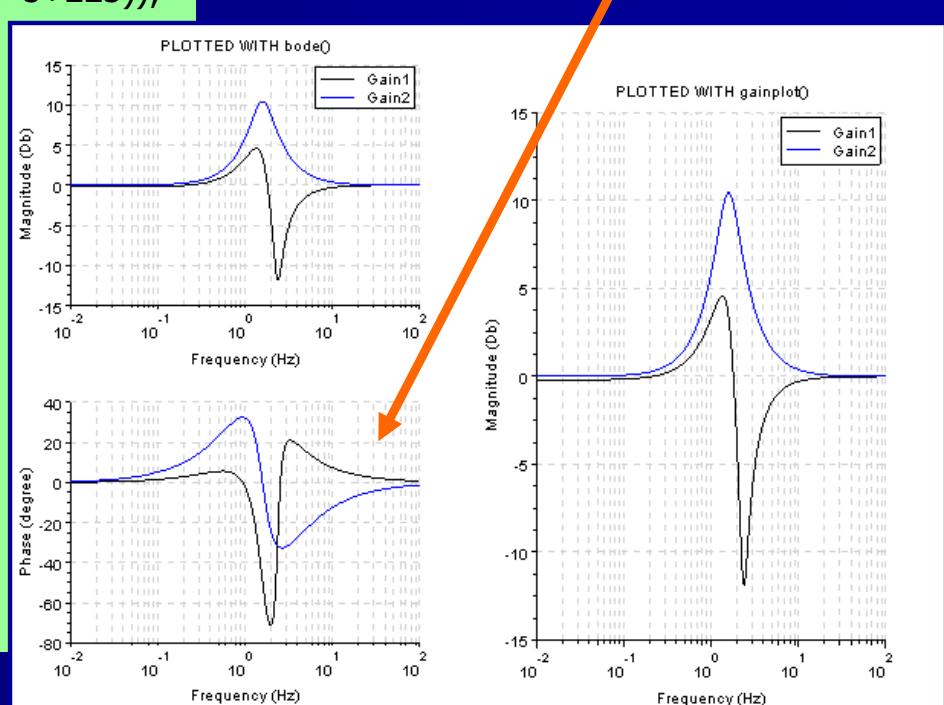
// Compare the how the bode() and gainplot()  /
// functions operate in Example 7           /

clear,clc,clf;
s = poly(0,'s'); // Polynomial seed
Gain1 = syslin('c',(s^2+20*s+100)/(s^2+6*s+100));
Gain2 = Gain1*syslin('c',(s^2+3*s+220)/(s^2+25*s+225));

// Plot with the bode() function:
//-----
subplot(121)
bode([Gain2;Gain1],0.01,100)
legend('Gain1','Gain2')
xtitle('PLOTTED WITH bode()')

// Plot with the gainplot() function:
//-----
subplot(122)
gainplot([Gain2;Gain1],0.01,100)
legend('Gain1','Gain2')
xtitle('PLOTTED WITH gainplot()')
```

This example demonstrates the `bode()` and `gainplot()` functions when operating on the earlier `Gain1` and `Gain2` expressions. `bode()` plots also the phase



Ex 3-1: comments

- The script was modified after being copied from the Scilab Group User's Manual and pasted into Editor. When copy-pasting, Editor tends to interpret citation marks ('c','s', etc.) wrongly and they have to be corrected manually
- Scilab is strict with the arguments for polynomial expressions. If, for instance, the 'c' is left out from the expression `poly([0,0,2000,200,25,1],'s','c')`, it will be translated into $10000000s^2 - 10455000s^3 + 455225s^4 - 2226s^5 + s^6$. **Be careful!**
- There is an advantage in using self-documenting expressions, here exemplified by naming the polynomials Gain1, Gain2, etc.
- The separate Bode plot demo showed that the `bode()` function has an advantage in providing also the phase of the system of interest
- The difference between Black's diagram and Nichols chart will be demonstrated in Example 3-2

Example 3-2: Black vs. Nichols

This example, adapted from Povy's tutorial, p. 78, shows what the chart() command adds to Black's diagram

The first vector argument of chart() defines the iso-gain curves to be plotted

The second argument defines iso-phase curves

list() defines plotting properties (the last argument does not have any effect)

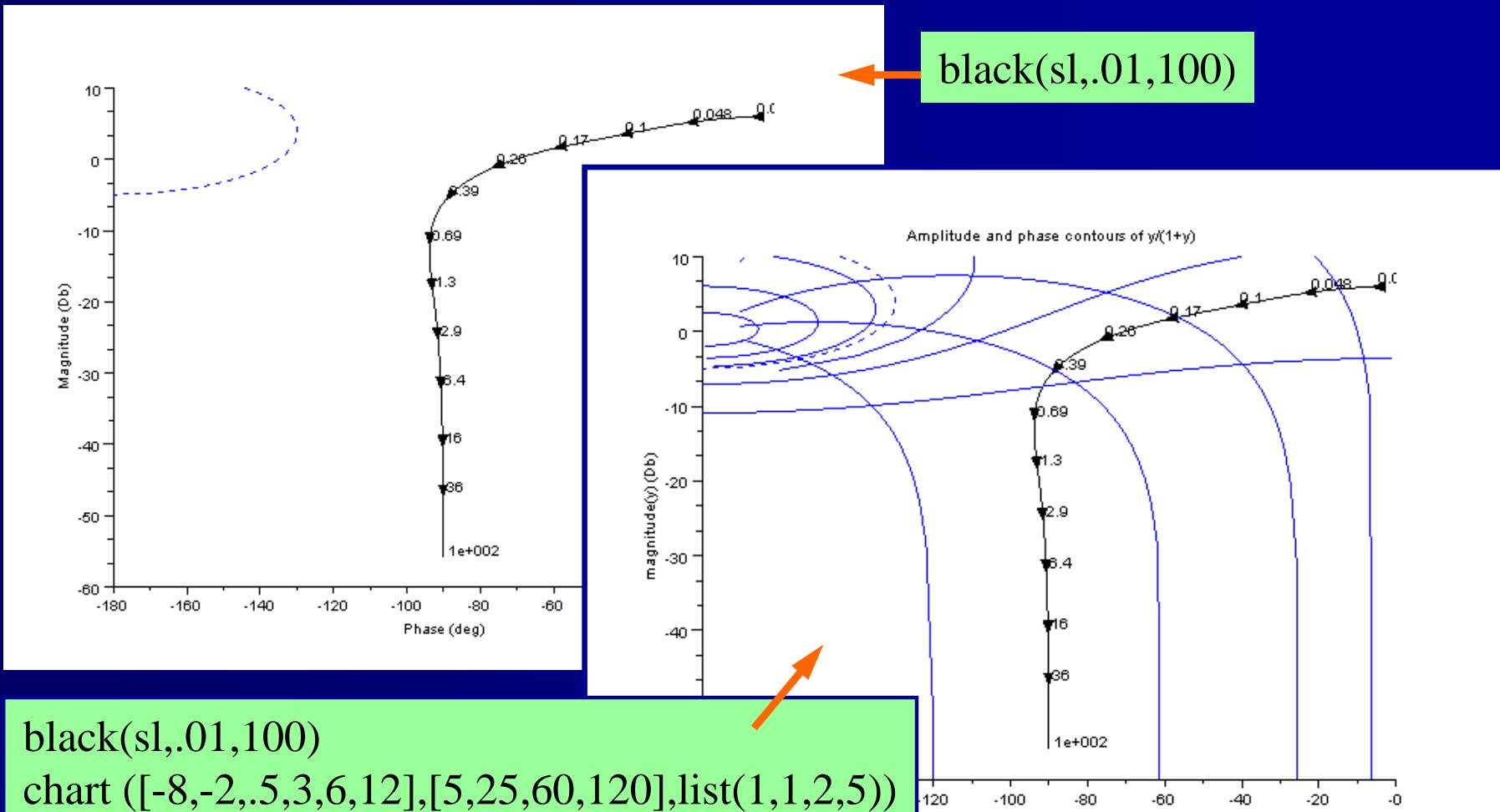
Check with Help for details

```
// black_nichols.sce
// Demonstration of black() and chart() functions
clear,clc,clf;
s = %s;
Gain = (2+3*s+s^2)/(1+3*s+2.5*s^2+s^3);
system = syslin('c',Gain);

black(system,.01,100) // Plot Black's diagram
chart([-8,-2,.5,3,6,12],[5,25,60,120],list(1,1,2,5))
// chart() adds iso-graphs
```

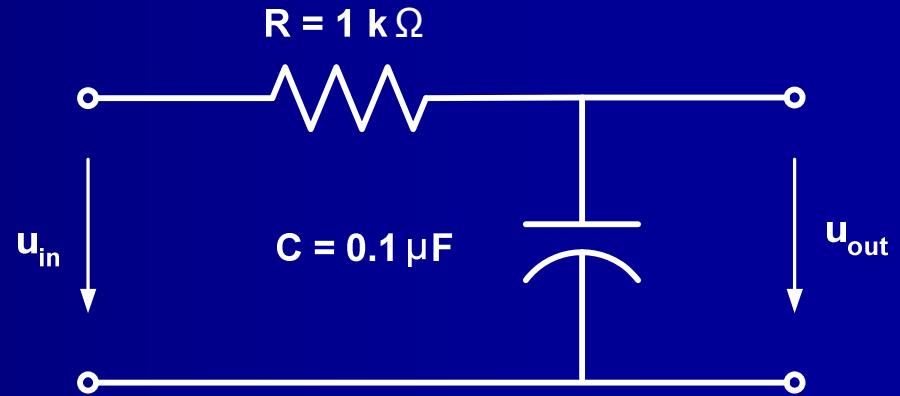
History Quiz: Did Nichols base his chart on work by Black or by Hall?

Ex 3-2: plots



Example 3-3: an RC circuit

- Let's do a Bode plot using just basic circuit theory and no Laplace rubbish
- The case is the simple RC circuit to the right (first-order low-pass filter)
- The task is to plot both the magnitude (gain) and phase
- The `bode()` function is not suitable for this case, instead we'll used `plot2d()` and define it separately for magnitude and phase



$$G = \frac{u_{\text{out}}}{u_{\text{in}}} = \frac{1}{1 + i2\pi f RC}$$

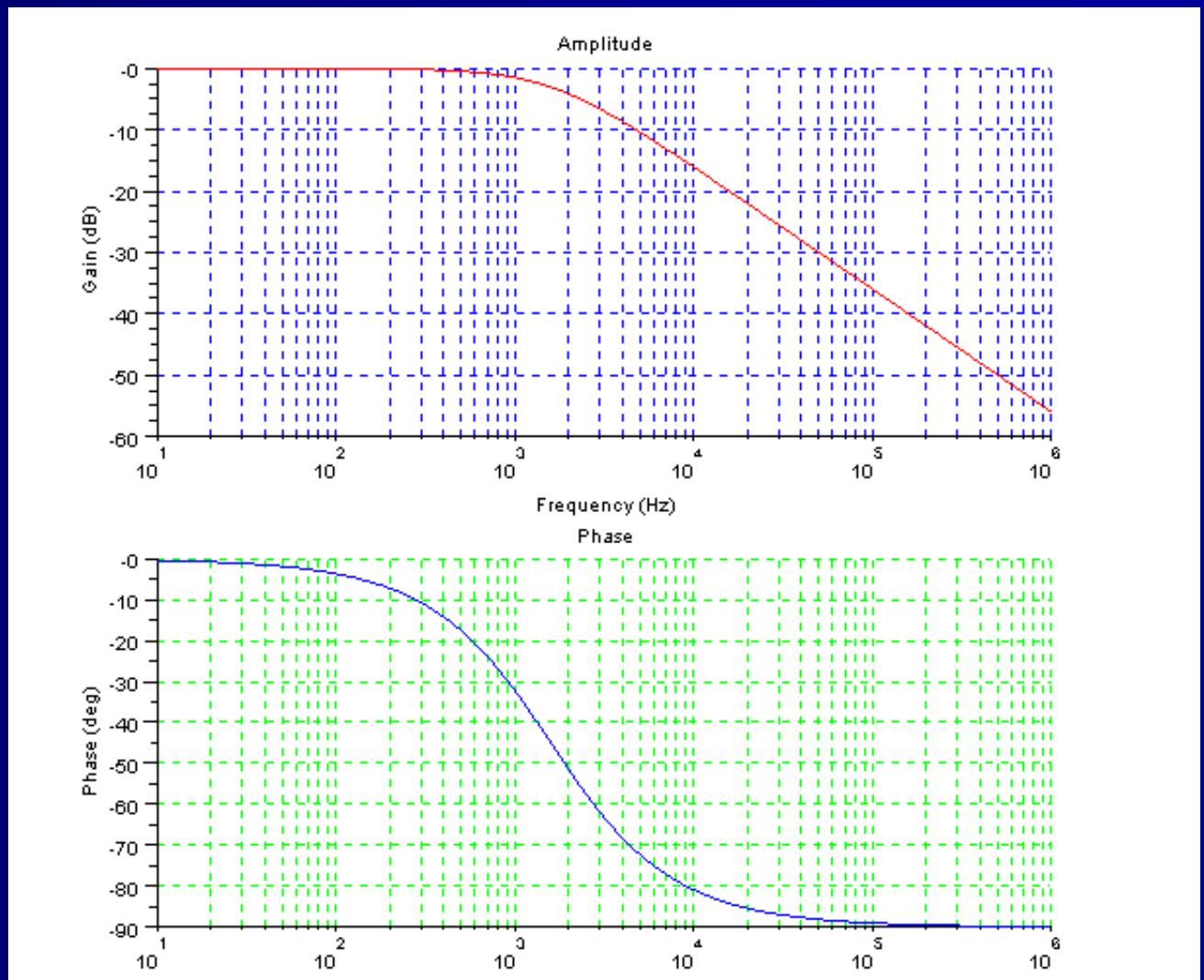
Ex 3-3: script

- The `logspace(1,6,60)` command means starting point 10^1 , end point 10^6 , in 60 steps and logarithmic scale
- Trigonometric phase definition and conversion to degrees
- The `logflag = 'ln'` argument defines a logarithmic x-scale and linear (normal) y-scale
- Different styles and `xgrid()` arguments have been used to demonstrate their effect

```
// bode_RC.sce  
  
// Bode diagram for an RC circuit /  
// (first-order low-pass filter) /  
  
clear,clc,clf;  
  
R = 1e+3;      // Resistance in ohm  
C = 1e-7;      // Capacitance in farad  
freq = logspace(1,6,60); // Frequency range, logarithmic  
G = 1 ./ (1 + %i*2*pi*freq*R*C); // Transfer function  
G_dB = 20*log10(abs(G));          // Logarithmic scale  
phase = ((atan(imag(G),real(G)))/(%pi))*180; // Phase  
  
subplot(211);                      // Amplitude plot  
plot2d(freq,G_dB,logflag='ln',style=5)  
xgrid(2)                           // Blue grid  
xtitle('Amplitude','Frequency (Hz)','Gain (dB)')  
  
subplot(212);                      // Phase plot  
plot2d(freq,phase,logflag='ln',style=2)  
xgrid(3)                           // Green grid  
xtitle('Phase','Frequency (Hz)','Phase (deg)')
```

Ex 3-3: plot

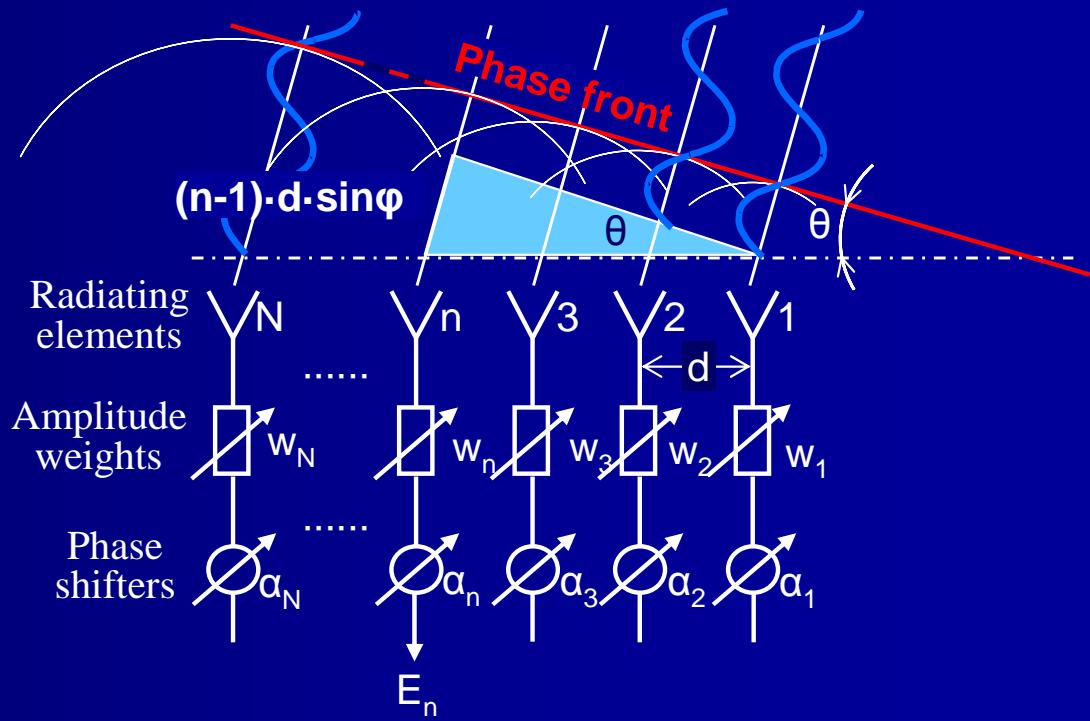
Note that the x-axis label is missing for the phase plot, although it was specified. Scilab does not repeat it since it is the same as the top one. Change the subplot declarations to (121) and (122) and the x-axis label is given for both parts



Example 3-4: linear antenna array

The task is to investigate the behavior of the Array Factor, AF (also known as field intensity pattern), of a linear antenna array with $N = 10$ isotropic radiating elements, when the main beam is scanned at $\theta_0 = 60^\circ$ and element spacing $d = 0.45$ and 0.55 wavelengths

(For a discussion on antenna arrays, see Brookner, E. (ed): Practical Phased Array Antenna Systems, Artech House, 1991)



$$| \text{AF} | = \left| \frac{\sin [N\pi (\frac{d}{\lambda}) \cdot \sin \theta]}{\sin [\pi (\frac{d}{\lambda}) \cdot \sin \theta]} \right|$$

Ex 3-4: beam scanning

The previous expression for AF is valid when the beam is normal to the array axis (broadside case). If the beam is deflected, the scan angle θ_o must be included in the equation

(In a more complete simulation we must also include element factors, tapering, and mutual coupling between the array elements)



Scan angle

Scan angle

$$| \text{AF} | = \left| \frac{\sin [N\pi (\frac{d}{\lambda}) \cdot (\sin \theta - \sin \theta_o)]}{\sin [\pi (\frac{d}{\lambda}) \cdot (\sin \theta - \sin \theta_o)]} \right|$$

Ex 3-4: script

This script is for the previous expression for AF, but normalized (divided by N) to keep the main beam value at unity. The phase { functions have been defined separately in order to shorten the expression for AF_norm

The array factor is → plotted both in linear and polar presentation

```
// array_factor.sce
//
// Plots the array factor of a linear antenna array with N elements,
// spaced at d wavelengths, and main beam scanned at +60 degrees
//
clear,clc,clf;

// Variables:
N = 10;           // Number of radiating elements
d1 = 0.45;        // Element spacing in wavelengths
d2 = 0.55;        // Ditto
theta = [-%pi/2:0.01:%pi/2]'; // Half space, +/-90 deg
theta_z = %pi/3;   // Scan angle

// Define array factors:
f_phase1 = %pi*d1*(sin(theta)-sin(theta_z)); // Phase function
f_phase2 = %pi*d2*(sin(theta)-sin(theta_z)); // Ditto
AF_norm1 = abs((sin(N*f_phase1)./sin(f_phase1))/N); // Normalized array factor (d=0.45)
AF_norm2 = abs((sin(N*f_phase2)./sin(f_phase2))/N); // Normalized array factor (d=0.55)

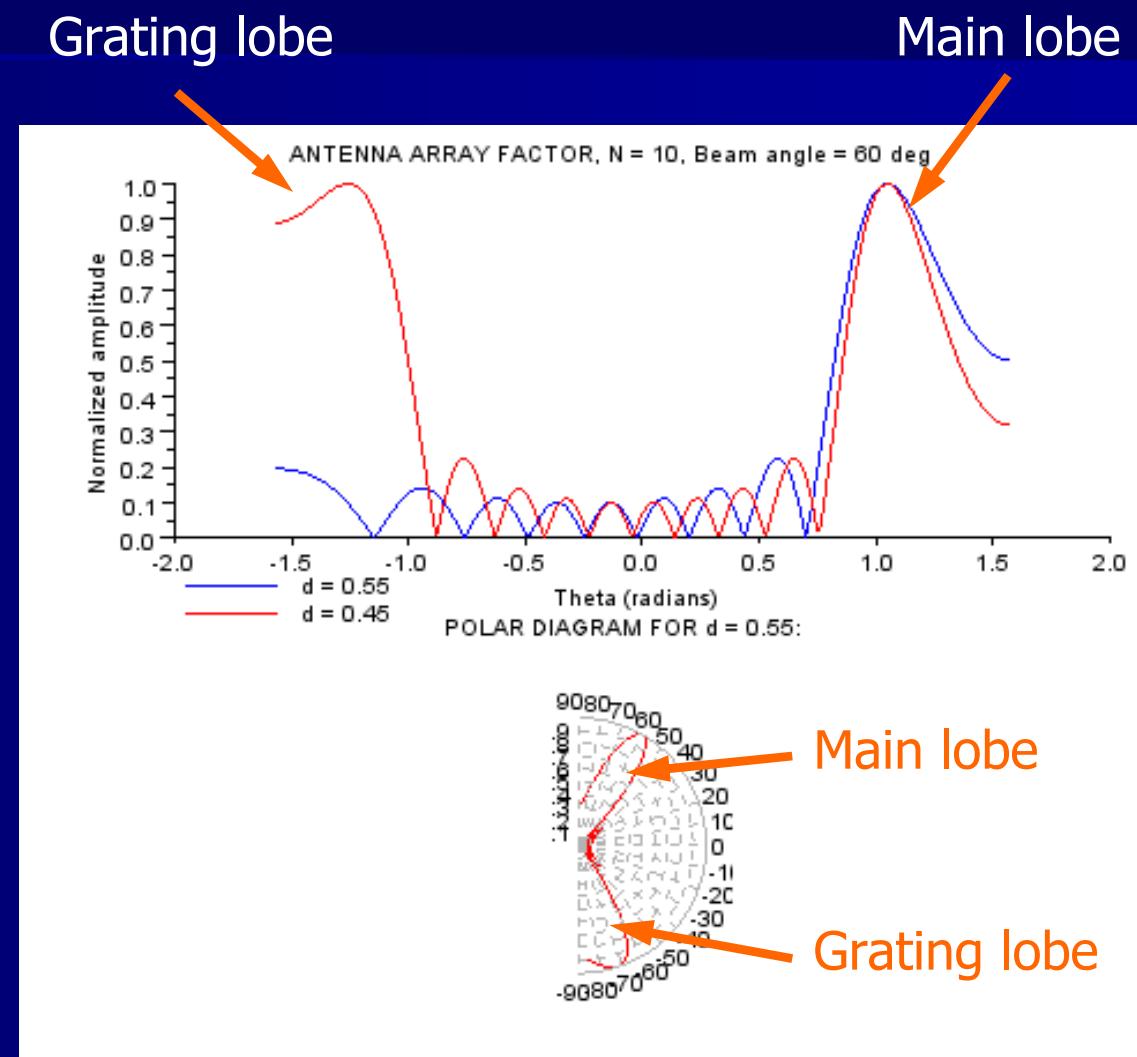
// Plot functions:
subplot(211)          // Linear plot (d=0.45,0.55)
plot2d(theta,[AF_norm1,AF_norm2], style=[2,5],...
leg="d = 0.55@d = 0.45")
xtitle("ANTENNA ARRAY FACTOR, N = 10, Beam angle = 60 deg',...
"Theta (radians)","Normalized amplitude")

subplot(212)          // Polar diagram (d=0.55)
polarplot(theta,AF_norm2, style=5)
xtitle('POLAR DIAGRAM FOR d = 0.55:')
```

Ex 3-4: plot

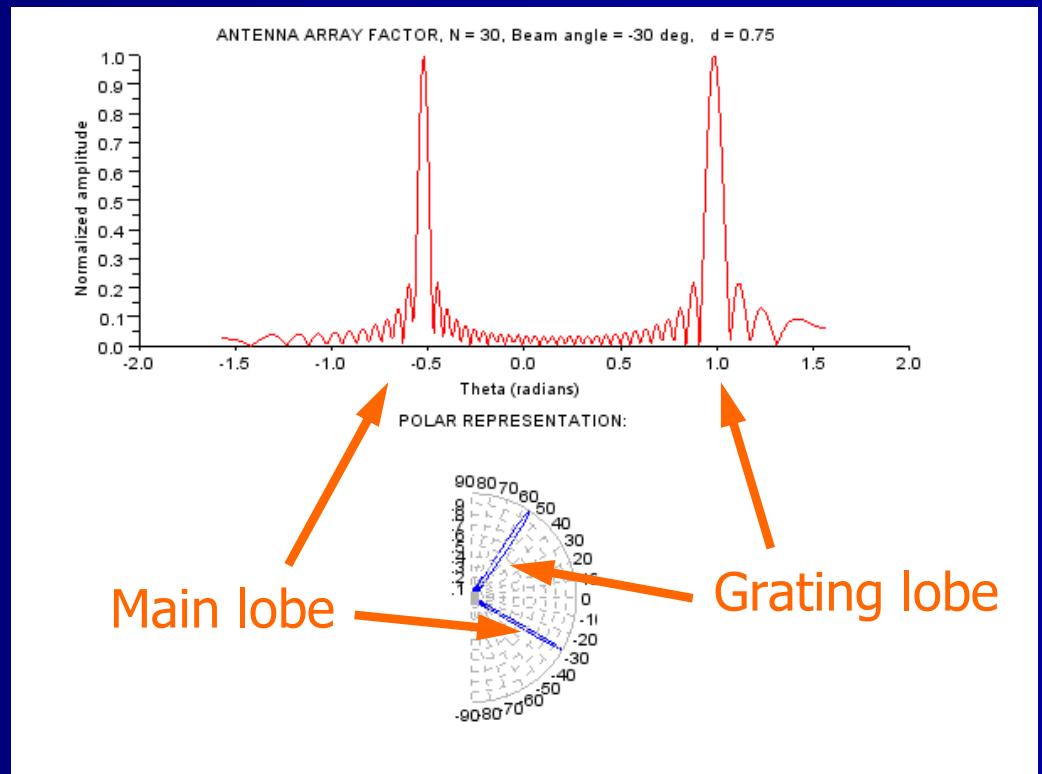
The plot verifies the common **rule of thumb**, according to which the array element spacing must satisfy the condition $d < \lambda/2$ or detrimental grating lobes will show up

Note that there is a **mirror image** in the other half space, only the $\pm 90^\circ$ case has been plotted



Ex 3-4: modified plot

- This case shows $d=0.45$ only. Other changes are:
 - Element spacing $d = 0.75$
 - Element number $N = 30$
 - Scan angle = -30°
- Scilab 5.1.1 tended under conditions like these to present only a part of the polar plot, but at the same time increase the size and let the polar plot overflowed the linear plot (which wasn't bad). This seems to have changed



Example 3-5: 3D sinc

This example is adapted from Kubitzki, *Einführung in Scilab*, pp. 41-42, and can be compared with the earlier one for plotting a 3D sinc function

The script introduces the use of colormaps to identify graphics colors. The argument of coppercolormaps() defines the number of colors in the map; 32 is typical

Note that color_map works on the Figure level ($f=gcf()$)

Here we use the pair drawlater() and drawnow() to control the plot process

```
//sinc_colormap.sce

// Define and plot 3D sinc function, graphic /
// adjust properties with handles /

clear,clc,clf;
x=linspace(-10,10,50); // Linear space of x
y=x; // Ditto for y

// **** SUBROUTINE sincf(): **** /
//-----
function [z]=sincf(x, y)
    r=sqrt(x.^2+y.^2)+%eps; // Auxiliary computation
    z=sin(r)./r; // Amplitude
endfunction

// **** MAIN, Compute sinc function: **** /
//-----
w=feval(x,y,sincf); // Evaluate with SUBROUTINE sincf()

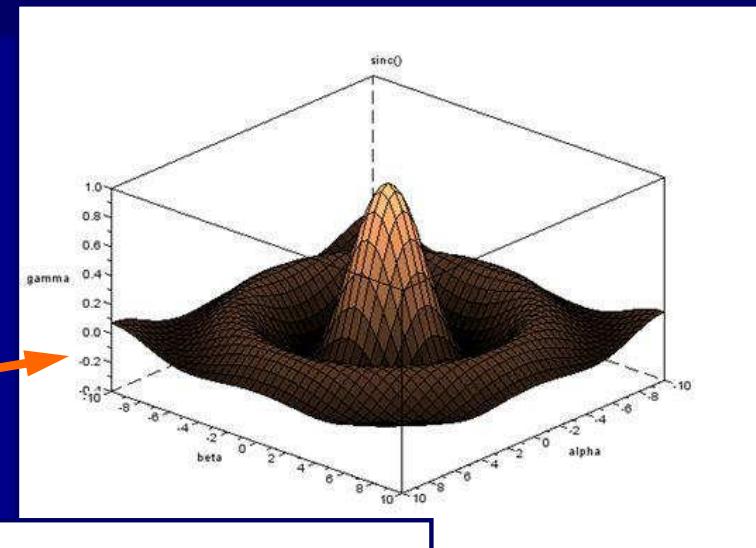
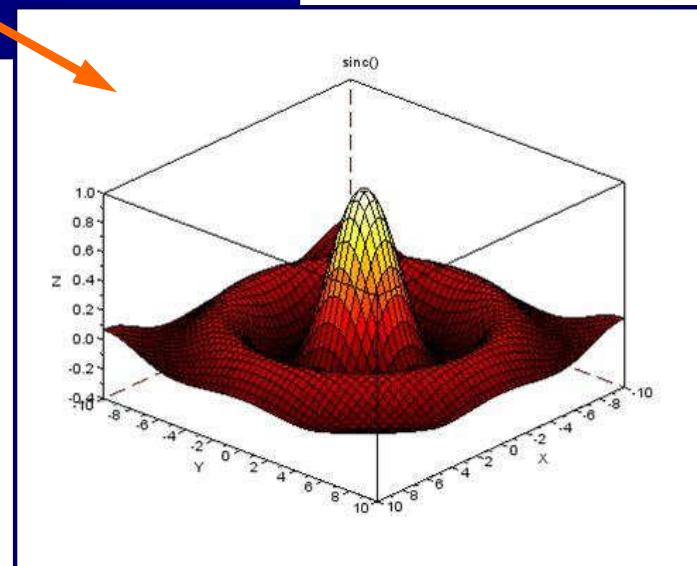
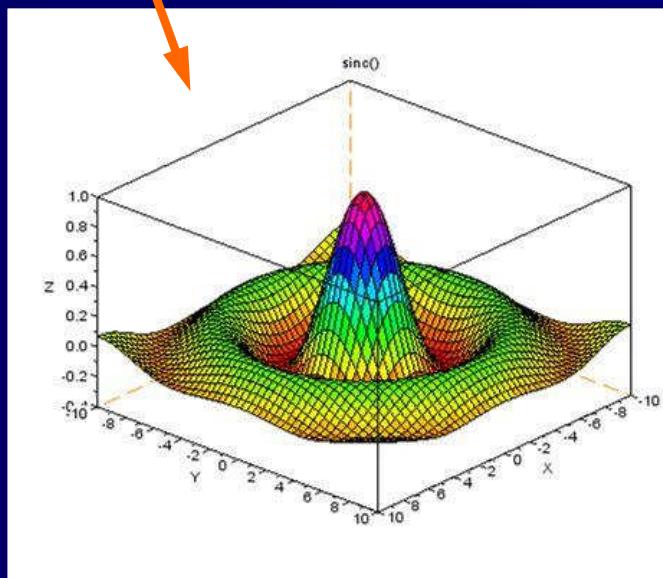
// Plotting & plot control:
//-----
drawlater(); // Suppress plotting until ready
plot3d(x,y,w); // (Suppressed) plot function
f=gcf(); // Get Figure handle
f.color_map = coppercolormap(32); // Set color table
h=gca(); // Get Axes handles
h.rotation_angles=[87,42]; // Set angle of observation
h.children.color_flag=1; // Use current color table
xtitle('sinc()','X','Y','Z'); // Title & legend
drawnow(); // Plot now
```

Ex 3-5: 3D sinc, plots & comments

Scilab has numerous colormap alternatives that allow the color of a 3D plot to be changed, e.g. the ones shown here. Check Help/Color management/colormap for more alternatives

hsvcolormap() coppercolormap()

hotcolormap()



Example 3-6: Lissajous figures, task

- The task is to write a script that generates Lissajous figures and to edit the figure with handles
- Lissajous figures are familiar to all who have worked with an oscilloscope in high school physics lab
- Mathematically Lissajous figures are the graph of a system of parametric equations of the type:

$$x = A \cdot (\sin(\omega t) + \varphi)$$

$$y = B \cdot \sin(\omega t)$$

- We shall plot two figures in one window, the combination of
 $\sin(x)$ & $\cos(3x)$ and
 $\sin(1.5x)$ & $0.5 * \cos(1.5x)$
- For comparison we first do a basic plot with `plot2d()` and then modify the figure with handles

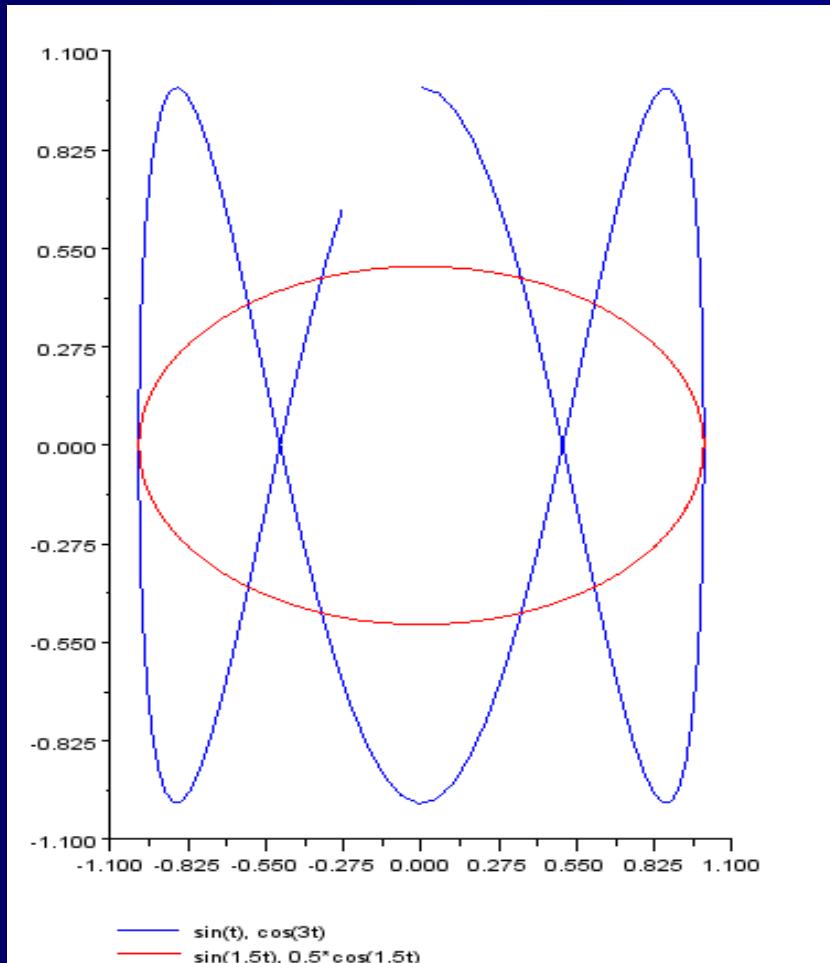
Ex 3-6: Lissajous figures, script 1

- Sine and cosine functions are grouped into matrices
- The plot2d() argument [2,5] defines graph colors
- The argument leg defines the legend
- The argument nax defines axes divisions
- The argument rect defines the extension of x and y axes

```
// handles_demo2-1.sce
// Two Lissajous figures, sin(t) & cos(3t) and      /
// sin(1.5t) & 0.5*cos(1.5t), with plot definitions / 
// given by arguments of the plot2d() function      /
clear,clc,clf;

// Plot Lissajous figures:
//-----
t=linspace(0,6,100)';
sines = [sin(t) sin(1.5*t)];
cosines = [cos(3*t) 0.5*cos(1.5*t)];
plot2d(sines, cosines, [2,5], ...
leg='sin(t), cos(3t)@sin(1.5t), 0.5*cos(1.5t)',...
nax=[1,9,1,9], rect=[-1.1,-1.1,1.1,1.1])
```

Ex 3-6: Lissajous figures, plot 1



The figure defined by $\sin(t)$, $\cos(3t)$ has **not quite finished** a full loop (its reach is defined by the argument 6 in `linspace()`)

The second figure, $\sin(1.5t)$, $0.5*\cos(1.5t)$, is already on its second loop. The **ellipse** becomes a **circle** if we change the cosine amplitude to 1

Pay attention to the fact that `plot2d()` combines sines and cosines arguments element-by-element

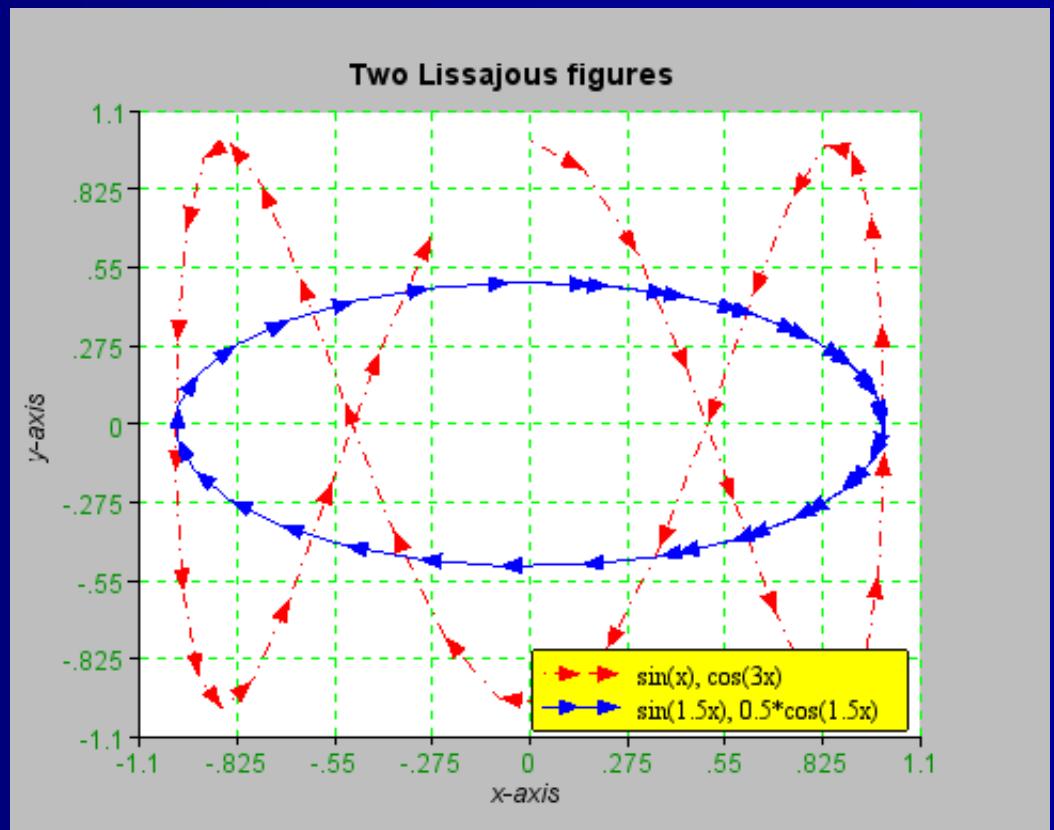
That was the basic thing, how do we improve it?

Ex 3-6: Lissajous figures, plot 2

This is the plot that has been modified using handles.* The script is presented on the next four slides

Major modifications are:

- Both Lissajous figures are arrow style, one line is dash-dotted
- Title and axes labels have been added & edited
- Background color has been added
- The legend box has been put in the lower right-hand corner, text edited and box color added
- A grid was added & edited



*) Ok, I have done some additions as well.

Ex 3-6: Lissajous figures, script 2 (1/4)

- # of linspace() steps is lowered to 40 to better show the arrows that are used below
- The body of plot2d() is retained, the remainder will be done with handles
- The figure handle is called by gcf(), after which the figure background color can be defined (addition to Script 1)

```
// handles_demo2-3.sce  
  
// Two Lissajous figures, sin(t) & cos(3t) and /  
// sin(1.5t) & 0.5*cos(1.5t), with plot edited /  
// using handles /  
  
clear,clc,clf;  
  
// Plot Lissajous figures:  
//-----  
x=linspace(0,6,40)'; // 40 steps to allow arrows  
sines = [sin(x) sin(1.5*x)]; // First figure  
cosines = [cos(3*x) 0.5*cos(1.5*x)]; // Second figure  
plot2d(sines,cosines,rect=[-1.1,-1.1,1.1,1.1])  
  
// Add background color:  
//-----  
f=gcf(); // Get Figure handle  
f.background=color('grey'); // Grey background color
```

Ex 3-6: Lissajous figures, script 2 (2/4)

- Call Axes handle with `gca()`, then edit the two Lissajous figures
- `p1` & `p2` are Compounds, children to Axes
- The graphs are Polylines and grandchildren to Axes
- Title and axes labels must first be added, after which they can be edited
- Recall that Title is a child to Axes
- Check with Help/`graphics_fonts` for details on fonts

```
// Edit Lissajous figures:  
//-----  
a=gca();           // Get Axes handle  
p1=a.children;    // sin(1.5x), 0.5*cos(1.5x)  
p1.children(1).polyline_style=4;      // Arrow mode  
p1.children(1).foreground=2;          // Change color to blue  
p1.children(1).arrow_size_factor=2;   // Line thickness  
p2=a.children;    // sin(x), cos(3x)  
p2.children(2).line_style=4;          // Dash-dot line  
p2.children(2).foreground=5;          // Change color to red  
p2.children(2).polyline_style=4;      // Arrow mode  
p2.children(2).arrow_size_factor=2;   // Line thickness  
  
// Add & edit title & labels:  
//-----  
xtitle('Two Lissajous figures', 'x-axis', 'y-axis');  
a.title.font_style=8;                // Font: Helvetica bold  
a.title.font_size=3;                // Increase title font size  
a.x_label.font_style=7;              // Font: Helvetica italic  
a.x_label.font_size=2;              // Increase x-label font  
a.y_label.font_style=7;              // Font: Helvetica italic  
a.y_label.font_size=2;              // Increase y-label font
```

Ex 3-6: Lissajous figures, script 2 (3/4)

- x- and y-axis ticks & marks (legends) are added
- Axes label font color & size are redefined
- Note that Ticks and Legends (marks) are children to Axes, similar to Labels
- A legend is added & edited

```
// Edit ticks & marks (labels):  
//-----  
a.x_ticks = tlist(['ticks','locations','labels'],...  
[-1.1,-.825,-.55,-.275,0,.275,.55,.827,1.1],...  
['-1.1','-.-825','-.55','-.275','0','.275','.55',...  
.825','1.1']);  
a.y_ticks = tlist(['ticks','locations','labels'],...  
[-1.1,-.825,-.55,-.275,0,.275,.55,.827,1.1],...  
['-1.1','-.-825','-.55','-.275','0','.275','.55',...  
.825','1.1']);  
a.labels_font_color=13; // Change label color  
a.labels_font_size=2; // Increase label size  
  
// Add & edit legend:  
//-----  
legend(['sin(x), cos(3x)'; 'sin(1.5x), 0.5*cos(1.5x)'], 4);  
leg=a.children(1); // Get legend handle  
leg.font_style=2; // Font: Times  
leg.font_size=2; // Increase legend font size  
leg.font_color=1; // Font color black  
leg.background=7; // Yellow legend box fill
```

Ex 3-6: Lissajous figures, script 2 (4/4)

- To finish, the grid is turned “on” and line colors edited
- Scilab does not have an equivalent for Matlab’s “grid on,” this is a way of circumventing the problem

```
// Add & edit grid:  
//-----  
set(gca(),'grid',[1 1]); // Matlab's "grid on"  
a.grid(1)=color('green'); // Vertical line color  
a.grid(2)=color('green'); // Horizontal line color
```

There were huge problems when I first tried to include the gce(), get current Entity, command in the script. The background color did not come up after Scilab was reloaded, I could not define ticks, etc.

Lessons learned: Be sure that you now what you do with gce()!

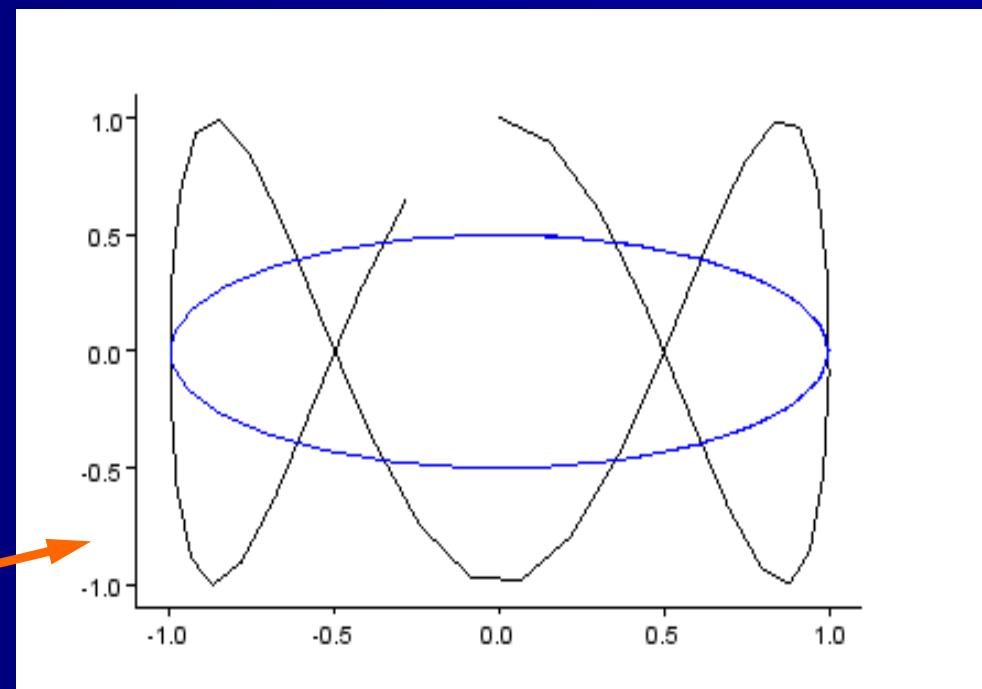
And a final check on the next slide...

Ex 3-6: Lissajous figures, check

After all these modifications, let's make sure that we can recall the basic plot by **adding the following lines** at the end of the script:

```
// Check default settings:  
//-----  
xdel();      // Delete Graphics Window  
sda();       // Reset default Axes  
plot2d(sines,cosines,rect=[-1.1,-1.1,1.1,1.1])
```

When we run the script, Scilab briefly flashes the modified plot,* deletes it, and puts up this window instead. The basic Lissajous figures seem to be ok



*) You can use the function pair drawlater() and drawnow() to avoid the flashing, as was done in Ex 3-5.

9. Converting Matlab files

The embedded Matlab-to-Scilab translator seems to work and manual conversion is an option

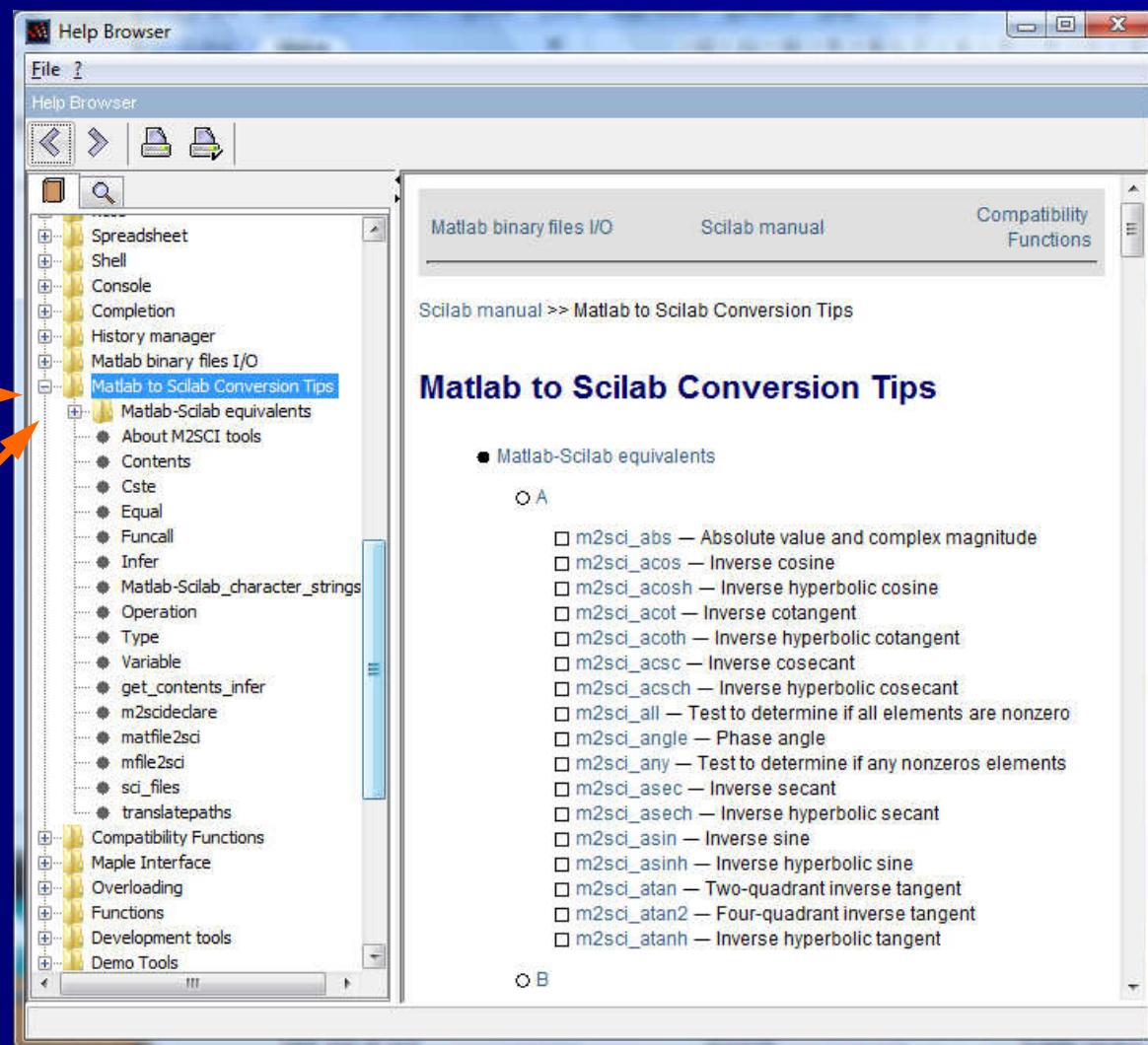


[Return to Contents](#)

Tips on Matlab to Scilab conversion

About halfway down the Help Browser (which is not in any logical order) you find Matlab to Scilab Conversion Tips. You will see a long list of the m2sci_... type of functions

Click on the first subheading, Matlab–Scilab equivalents, and you get a list of Matlab functions and their Scilab equivalents (missing ones are not included, like text() and xlim() mentioned below)

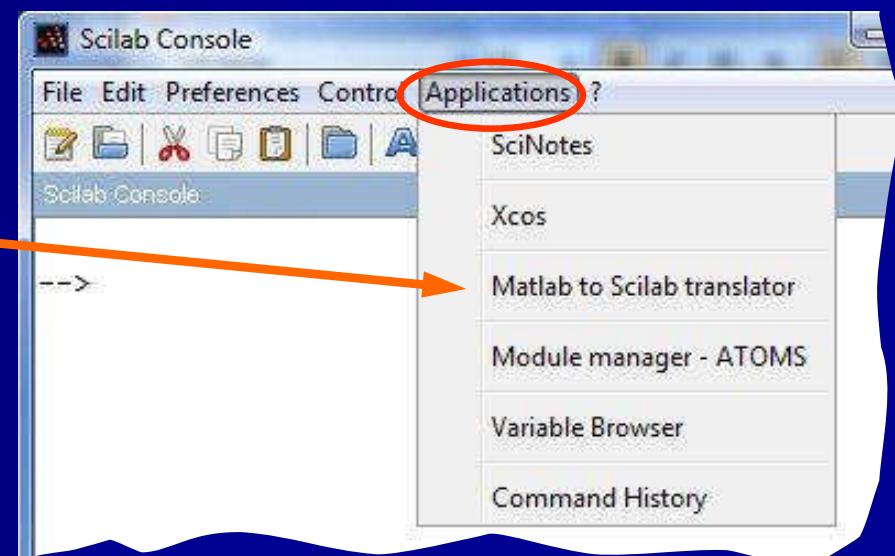


Using the integrated Matlab-to-Scilab translator

- Scilab can convert Matlab's .m files to .sci files, although we should not expect the conversion to be fully successful every time. The translated script may have to be modified manually
- We start by opening the translator: *

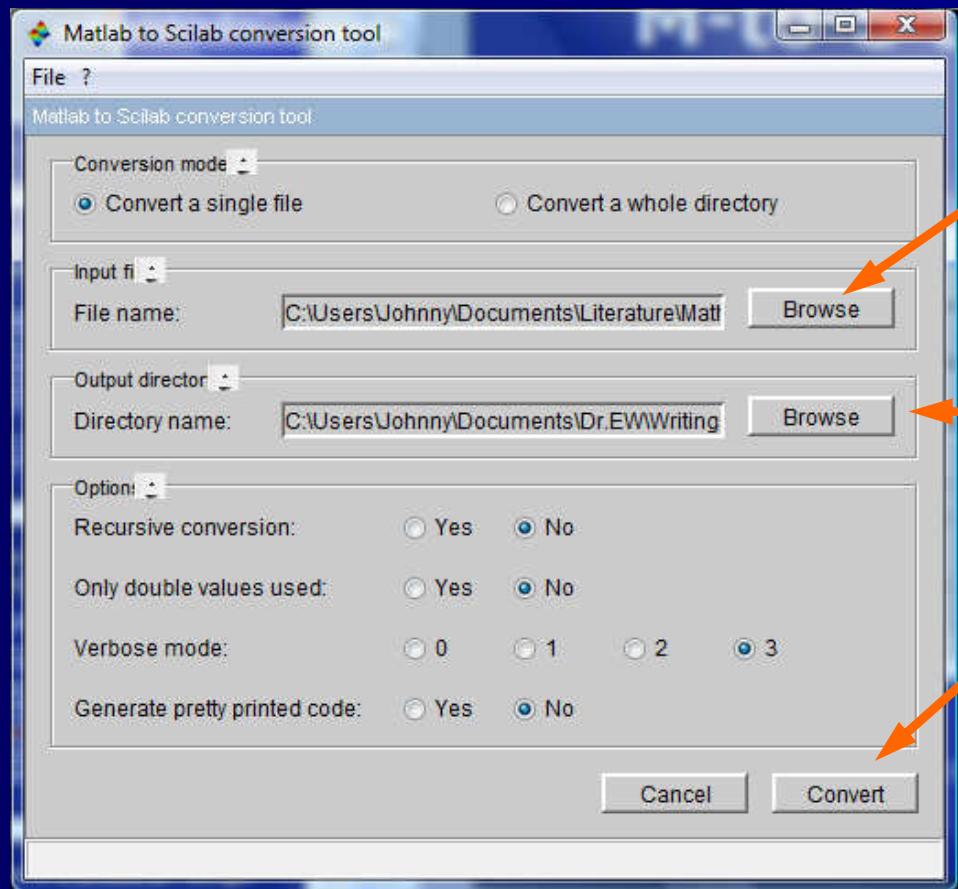
On the Console, Click:
Applications\Matlab to
Scilab translator

What happens next you can see
on the following slide



*) It used to be possible to import Matlab files directly, but this option does not exist any more.

M-to-S translator: the process



The conversion tool opens

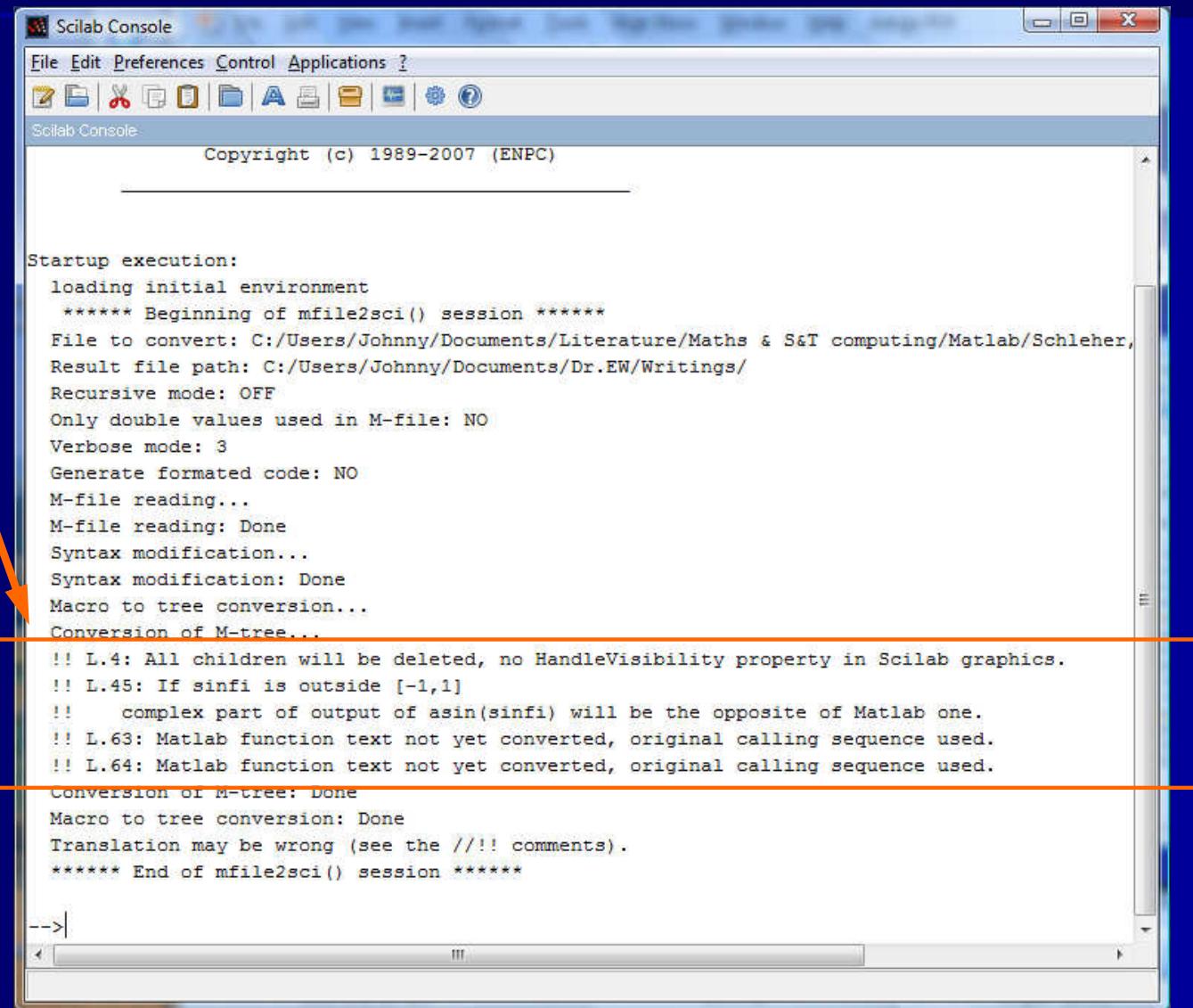
1. Click: (File name) Browse and identify the file that you want to translate
2. Click: (Directory name) Browse to point out where to put the translated script (and associated products*)
3. Click: Convert

*) The conversion produces two text documents and two .sci scripts

M-to-S translator: messages on the Console

Scilab presents a list of translation conditions and also warnings of possible errors on the Console. The warnings are repeated as comments in the script (and on one of the text documents)

Next, open the translated script in the Editor



```
Scilab Console
File Edit Preferences Control Applications ?
Scilab Console
Copyright (c) 1989-2007 (ENPC)

Startup execution:
loading initial environment
***** Beginning of mfile2sci() session *****
File to convert: C:/Users/Johnny/Documents/Literature/Maths & S&T computing/Matlab/Schleher,
Result file path: C:/Users/Johnny/Documents/Dr.EW/Writings/
Recursive mode: OFF
Only double values used in M-file: NO
Verbose mode: 3
Generate formated code: NO
M-file reading...
M-file reading: Done
Syntax modification...
Syntax modification: Done
Macro to tree conversion...
Conversion of M-tree...
!! L.4: All children will be deleted, no HandleVisibility property in Scilab graphics.
!! L.45: If sinfi is outside [-1,1]
!!      complex part of output of asin(sinf) will be the opposite of Matlab one.
!! L.63: Matlab function text not yet converted, original calling sequence used.
!! L.64: Matlab function text not yet converted, original calling sequence used.
Conversion of M-tree: Done
Macro to tree conversion: Done
Translation may be wrong (see the !!! comments).
***** End of mfile2sci() session *****

-->
```

M-to-S translator: script (1/4)

This is the script that the translator delivers. It contains comments that may or may not be of importance:

Statement & warning
added by Scilab

Here comes the
second warning. The
Matlab command was
clear,clc,clf;. May
be of importance if
the script is edited
with handles. In such
a case, try to create
a new script by copy-
pasting

```
// Display mode
mode(0);

// Display warning for floating point exception
ieee(1);

// Monopulse Antenna Pattern
// -----
clear,clc,!! L.4: All children will be deleted, no HandleVisibility property in Scilab graphics.
clf;

// Normalized Aperture Width
na = 4;

// Sampling Frequency=Number elements per norm aperture
fs = 8;
```

M-to-S translator: script (2/4)

Everything runs smoothly here

The code is expected to present the sum and difference patterns for a monopulse antenna (tracking radar, etc.)

```
// Norm aperture with N elements
N = fs*na;
xna = na*(-1/2:1/(N-1):1/2);

// Illumination Function

wxna(1,1:N/2) = ones(1,N/2);
wxna = mtlb_i(wxna,N/2+1:N,-ones(1,N/2));
wxnb(1,1:N/2) = ones(1,N/2);
wxnb = mtlb_i(wxnb,N/2+1:N,ones(1,N/2));

// Fill with M/2 zeros front and back

M = 1024;
xna = na*(-1/2:1/N+M-1:1/2);
wxna = [zeros(1,M/2),wxna,zeros(1,M/2)];
wxnb = [zeros(1,M/2),wxnb,zeros(1,M/2)];

// Beam Functions from -fs/2 to fs/2 in sine space

Nfft = max(size(wxna));
Esine = mtlb_fft(wxna,Nfft);
Esine = fftshift(Esine);
```

M-to-S translator: script (3/4)

Here comes more
warnings. May relate
to a rounding error



```
Esum = mtlb_fft(wxnb);
Esum = fftshift(Esum);

// Azimuth vector

sinf = ((fs/4)*(-Nfft/2:Nfft/2-1))/Nfft;

// Azimuth vector in radians

// !!! L.45: If sinfi is outside [-1,1]
// !!! complex part of output of asin(sinf) will be the
// opposite of Matlab one.
fi = asin(sinf);

// Beam gain functions

Gfi = (Esine .*conj(Esine))/Nfft;
Gfs = (Esum .*conj(Esum))/Nfft;

Gfi = mtlb_i(Gfi,1:Nfft/2,sqrt(Gfi(1:Nfft/2)));
Gfi = mtlb_i(Gfi,Nfft/2+1:Nfft,-sqrt(Gfi(Nfft/2+1:Nfft)));
Gfs = sqrt(Gfs);
```

M-to-S translator: script (4/4)

Note that title() is an alternative to xlabel()

Here come the last warnings. The next slide shows what they mean

Well, let's see how Scilab reacts by executing the script...

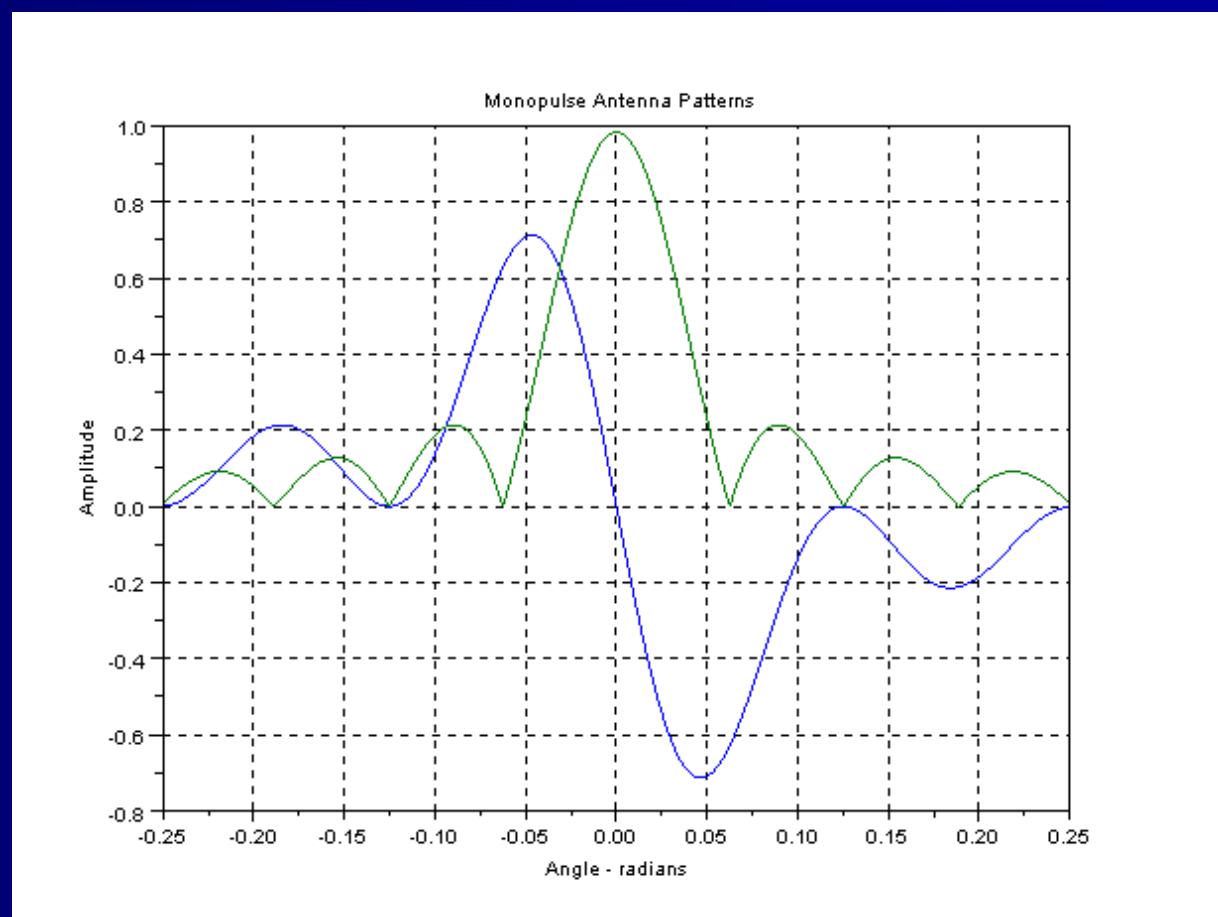
// Plot Monopulse Antenna Pattern

```
plot(fi,Gfi,fi,Gfs);mtlb_grid;
set(gca(),"data_bounds",matrix([-0.25,0.25,-0.8,1],2,-1));
ylabel("Amplitude");
xlabel("Angle - radians");
title("Monopulse Antenna Patterns");
// !!! L.63: Matlab function text not yet converted,
original calling sequence used.
text(0.04,0.8,"Sum Pattern");
// !!! L.64: Matlab function text not yet converted,
original calling sequence used.
text(-0.22,0.6,"Difference Pattern");
```

M-to-S translator: plot

Yees, it comes,
labels and all!

But the **legends** are
missing, which
means that Scilab
cannot cope with
Matlab's `text()`
function



M-to-S translator: comments

- Based on this example one could say that the embedded Matlab-to-Scilab translator is adequate
- A legend has to be added manually to compensate for the missing `text()` information*
- The example demonstrates partly good programming practice by **declaring each logical entity**. However, informative explanations could be added
- Another improvement is to use **expressive variable names**. Why not talk about `sampl_freq` instead of `fs`, and what does `wxna()` stand for?
- Help sheds no light over the meaning of the second .sci files that the conversion produces

*) A paper by Sharma & Gobbert (2010) reports that the translator cannot cope with Matlab's `xlim()` function. In their case the `plot()` function had to be manually changed to `plot2d()` to correct the problem.

Manual conversion (1/6): Case #1, script

- To the right is a Matlab code (top) and its Scilab equivalent (bottom). The way I did it:
- Checked visually for differences:
 - Comments: % → //
 - Built-in functions: pi → %pi
 - Plot commands are split up on multiple rows. Indents are not necessary, but a visual help
- Changed what I could, then run the script and let Scilab's debugger yell about the rest
- Checked frequently with Help (particularly Matlab–Scilab equivalents) to understand the error messages on the Console

```
% beating sinusoidal tones
%
t = linspace(-1e-2,1e-2,1001);
x = cos(2*pi*1500*t) + cos(2*pi*1300*t);
m = 2*cos(2*pi*100*t);
plot( t, m, 'b:', t, -m, 'b:', t, x, 'k',...
axis( [-0.01 0.01 -2.4 2.4] ),...
title( 'Beating between tones' ),...
xlabel( 'Time (s)' ),...
ylabel( 'Amplitude' )
```

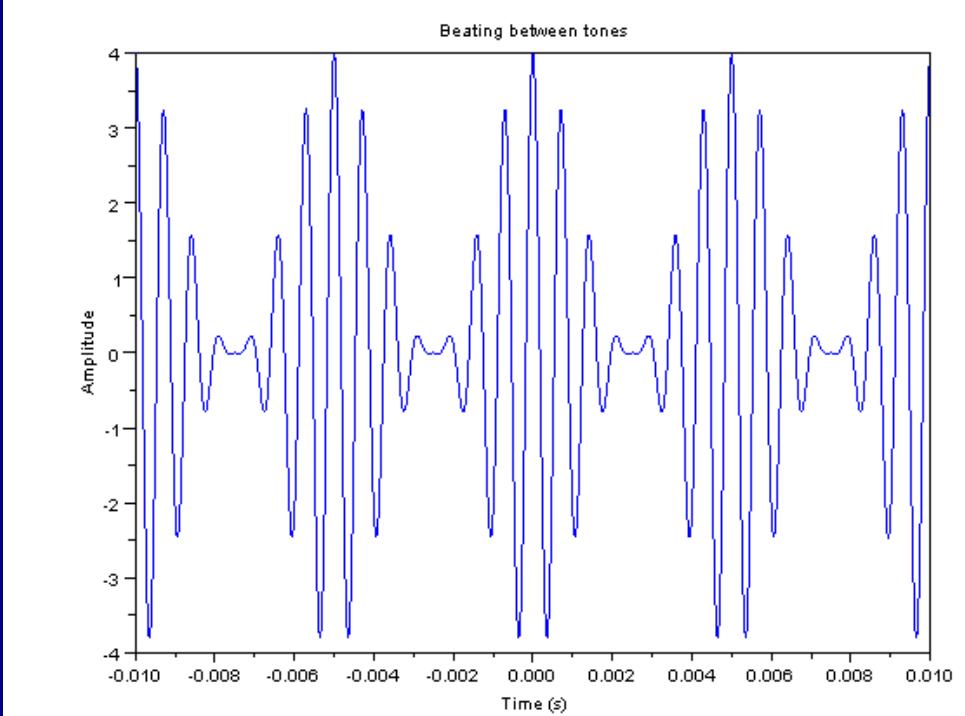
```
// M-to-S_1-modulation.sce

// beating sinusoidal tones  /

clear,clc,clf;
t = linspace(-1e-2,1e-2,1001);
x = cos(2*%pi*1500*t) + cos(2*%pi*1300*t);
m = 2*cos(2*%pi*100*t);
plot( t, x.*m, rect = [-0.01 0.01 -2.4 2.4] )
title( 'Beating between tones' )
xlabel( 'Time (s)' )
ylabel( 'Amplitude' )
```

Manual conversion (2/6): Case #1, plot

- There were some problems with this conversion:
 - I split up Matlab's long `plot()` command, but the abbreviated form did not work in Scilab
 - First I changed Matlab's `axis()` to `rect()`, then swapped the preceding argument part for `x.*m`. Now `plot()` worked
 - The label commands gave problems. The reason was—again—that I had copied the Matlab code and the **quotation marks** were pasted incorrectly



Manual conversion (3/6): Case #2, script & plot

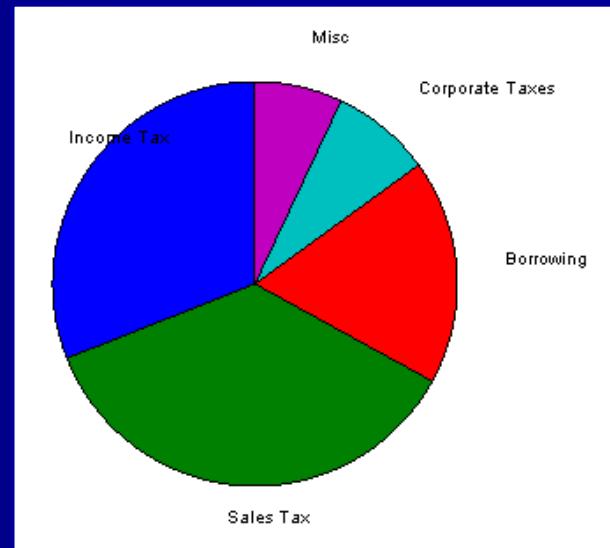
- The pie() function has not been discussed before, but below is a short Matlab script that draws a pie graph*
- The pie() function also exists in Scilab, the difference is that Scilab does not support Matlab's pielabel() function

```
revenues = [31 36 18 8 7];
h = pie(revenues);
pielabel(h,['Income Tax: ';'Sales Tax: ';'Borrowing:',...
    'Corporate Taxes: ';'Misc: ']);
```

```
// M-to-S_2-pie.sce

// Draw a pie graph with labels /

clear,clc,clf;
revenues = [31 36 18 8 7];
pie(revenues,['Income Tax','Sales Tax','Borrowing',...
    'Corporate Taxes','Misc']);
```



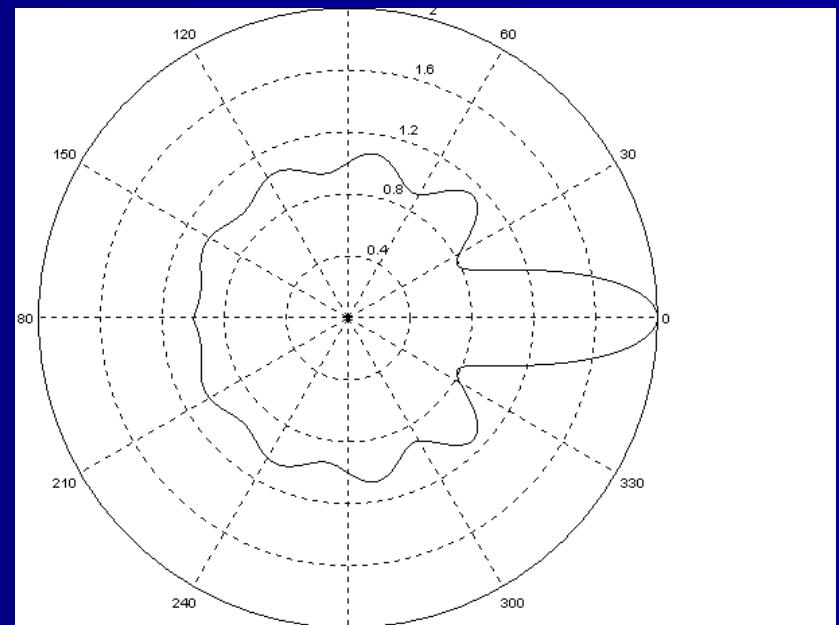
*) A more general sector chart will be presented in Example 6-4

Manual conversion (4/6): Case #3, script & plot

- As a last case, let's look at a shifted sinc function
- In this case the problem is that `polardb()` is an informal creation by Matlab users which Scilab does not support

```
x = -(5*2*pi):.1:(5*2*pi);
th = linspace(-pi,pi,length(x));
rho=((1+sin(x)./x));
polardb(th,rho)
```

```
// M-to-S_3polarplot.sce
// Polar plot of 1+sin(x)/x  /
clear,clc,clf;
x = -(5*2*%pi):.1:(5*2*%pi);
th = linspace(-%pi,%pi,length(x));
rho = 1+sin(x)./x;
polarplot(th,rho)
```

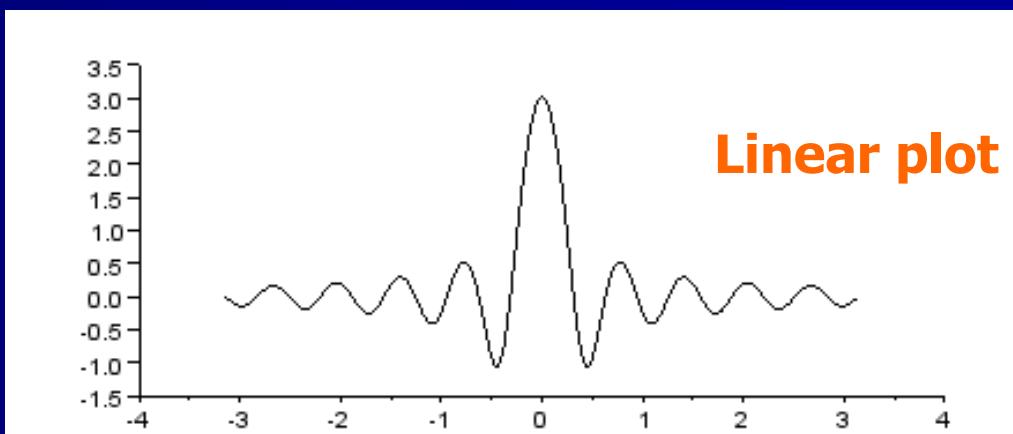


Similar, but not the same as the Matlab plot if radial units are in dB

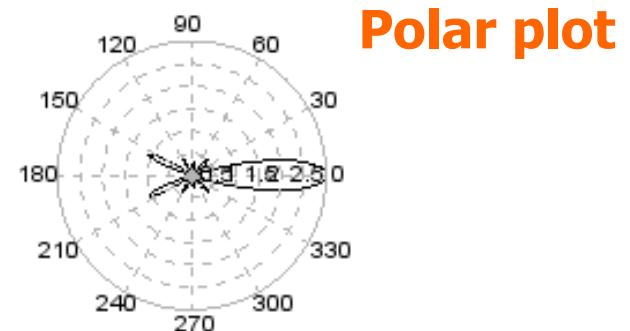
Manual conversion (5/6): Case #3, discussion

- The polar plot with radial units in dB looks rather “counterintuitive,” since its sidelobes appear to be pointing in the wrong direction

```
// M-to-S_3polarplot.sce  
  
// Polar plot of 1+sin(x)/x /  
  
clear,clc,clf;  
x = -(5*2*%pi):.1:(5*2*%pi);  
th = linspace(-%pi,%pi,length(x));  
rho = 10*log10((1+sin(x)./x));  
subplot(211);  
plot2d(th,rho)  
subplot(212);  
polarplot(th,rho)
```



Linear plot



Polar plot

Manual conversion (6/6): discussion & hints

- Manual conversion of Matlab codes to Scilab scripts is possible, there are users who claim to do it regularly
- *Scilab for Matlab Users* –tutorials and Scilab discussion forums can help in understanding the differences
- Some Matlab functions simply do not exist in Scilab (and vice versa). Examples are axis(), compass(), feather(), fill(), nargin(), polar(), quad(), quiver(), stem(), stairs(), and waterfall()
- Sometimes alternative Scilab commands exist (e.g., Scilab's plot2d2() can compensate for Matlab's stairs()), sometimes not. If not, the script may have to be rewritten
- Scilab's user-defined functions must be loaded with getf(), while Matlab has no separate load function
- Matlab's run data.m should be traded for exec('data.sci') in Scilab
- One more case of manual conversion will be presented in Example 6-5 (Chapter 19)

10. Subroutines

This discussion on subroutines is a prelude to flow control that will be discussed in Chapter 11



[Return to Contents](#)

Terminology

Recall from Chapter 1 that Scilab does not recognize the term “subroutine,” which belongs to the group of varied constructs that Scilab calls “function.” More exact, we are talking about user defined functions (UDFs), an expression that Scilab also does not know

Regardless of official Scilab terminology, I will—when possible—use the traditional term **subroutine** since it **is an elegant way of pointing to specific entities** in computer programs



An introductory demo

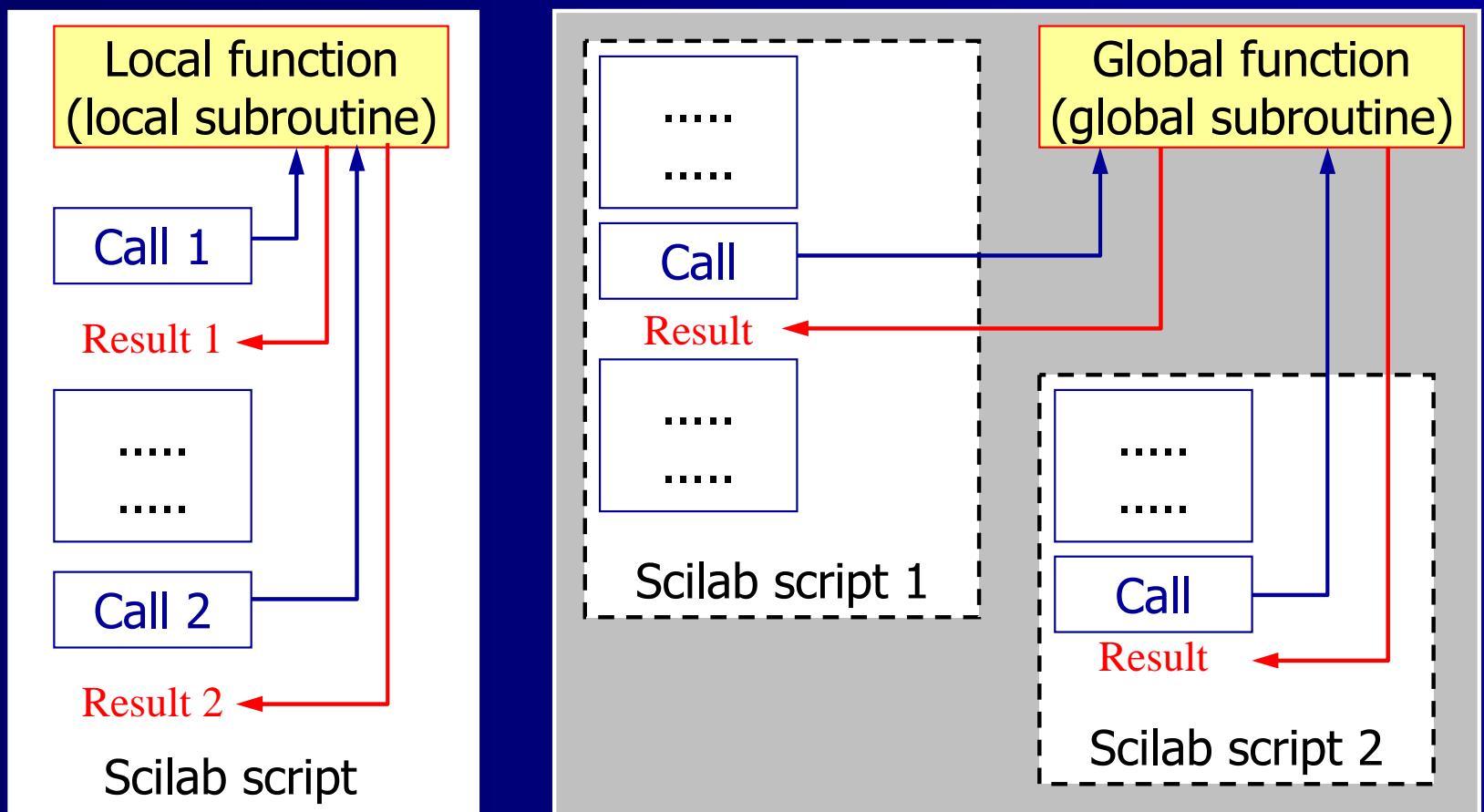
- Recall **Example 1-3** that introduced the concept of user defined functions (UDFs)
- Task: Write a function that calculates the area of a triangle with known side lengths
- The function is entered on Editor
- It is then loaded into the Console using the Editor command Execute/... file with echo
- The function is executed by entering the function name and input parameters (side lengths) on the Console

```
function A = triangle_area(a,b,c)
// The function 'triangle_area' calculates the /
// area of a triangle with side lengths a, b, c. /
funcprot(0)
p = (a+b+c)/2 // p = half perimeter
A = sqrt(p*(p-a)*(p-b)*(p-c))
endfunction
```

```
-->function A = triangle_area(a,b,c)
-->// The function 'triangle_area' calculates the /
-->// area of a triangle with side lengths a, b, c. /
-->funcprot(0)
-->p = (a+b+c)/2 // p = half perimeter
-->A = sqrt(p*(p-a)*(p-b)*(p-c))
-->endfunction
-->
-->triangle_area(4,5,6)
ans =
9.9215674
```

Local and global functions (subroutines)

- Local functions are embedded in a script and valid for it alone, global functions are saved separately and accessible to any script



Local and global variables

- You will run into the terms local and global variables and they need a short clarification:
 - As with functions, Scilab has two types of function **variables**, local and global:
 - **Local variables** are limited to a specific function
 - **Global variables** are available to, and can be altered by, all functions in which the variable has been declared global
 - The transfer of parameters using command window (Console) variables and global variables is not too obvious. Global variables, in particular, can lead to **errors that are difficult to detect**
 - For the reasons mentioned, **the use of global variables should be limited to a minimum**
- In conclusion, we consider only **local variables** that are the **default setting** in Scilab. This discussion is therefore trivial

Subroutines, more formally

- In the general case where a subroutine has **several** input arguments (in_arg1,in_arg2,...) and returns **several** output arguments (out_arg1,out_arg2,...), the structure is:

```
function [out_arg1, out_arg2,...] =...
    funktion_name(in_arg1, in_arg2, in_arg3,...)
    out_arg1 = expression for 1st output argument;
    out_arg2 = expression for 2nd output argument;
    ...
endfunction
```

- Structure borders are the *function endfunction* limiters
- Input arguments are grouped with **brackets** (parentheses), output arguments with **square brackets** (not needed for a single output parameter)
- In both cases the arguments are separated by commas

On output arguments

The example to the right highlights the basic way in which Scilab manages output arguments of subroutines

When you need to influence the management of **input and output** variables, Scilab offers the **functions** `argn()`, `varargin()`, and `varargout()`

```
-->function [y1,y2] = myfunc(x1,x2);
-->y1 = 3*x1;
-->y2 = 5*x2+2;
-->endfunction

-->myfunc(4,7)
ans =
12.  3x4=12

-->y2 = myfunc(4,7)
y2 =
12.  3x4=12

-->[y1,y2] = myfunc(4,7)
y2 =
37.  5x7+2=37

y1 =
12.  3x4=12
```

With no output argument defined, the first output argument is returned in the `ans` variable

The same answer is returned when only one output argument is defined

With both output arguments defined, the result of the computation is returned in full

Vector arguments

- This function* uses vectors as input and output arguments
- The function is first defined
- After that the output arguments—the operations to do—are defined
- Next the input arguments (row vectors) are entered
- At the end the function is executed
- The in-line function deff() is a specialized form of local functions

```
-->// Define local subroutine cross_product
-->function [x] = cross_product(a,b)
-->    x(1) = (a(2)*b(3) - a(3)*b(2))
-->    x(2) = (a(3)*b(1) - a(1)*b(3))
-->    x(3) = (a(1)*b(2) - a(2)*b(1))
-->endfunction

-->// Plug in numeric values
-->a = [-2 5 8];
-->b = [7 -13 -5];

-->// Executing the subroutine
-->c_prod = cross_product(a,b)
c_prod =
79.   x(1)=5*(-5)-8*(-13)
46.   x(2)=8*7-(-2)*(-5)
- 9.   x(3)=-2*(-13)-5*7
```

*) Here I use the term “function” since the code is independent and not called by a main program.

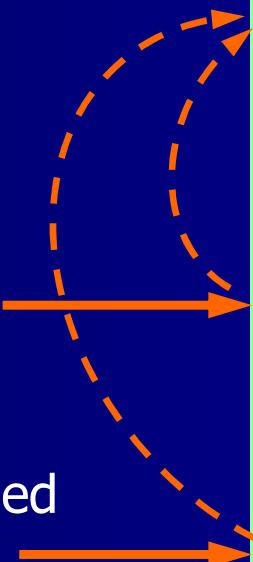
Demo (1/2): script

Task: Compute & plot a parabola, find its positive root

Here the subroutine is called for the first time using the input argument x

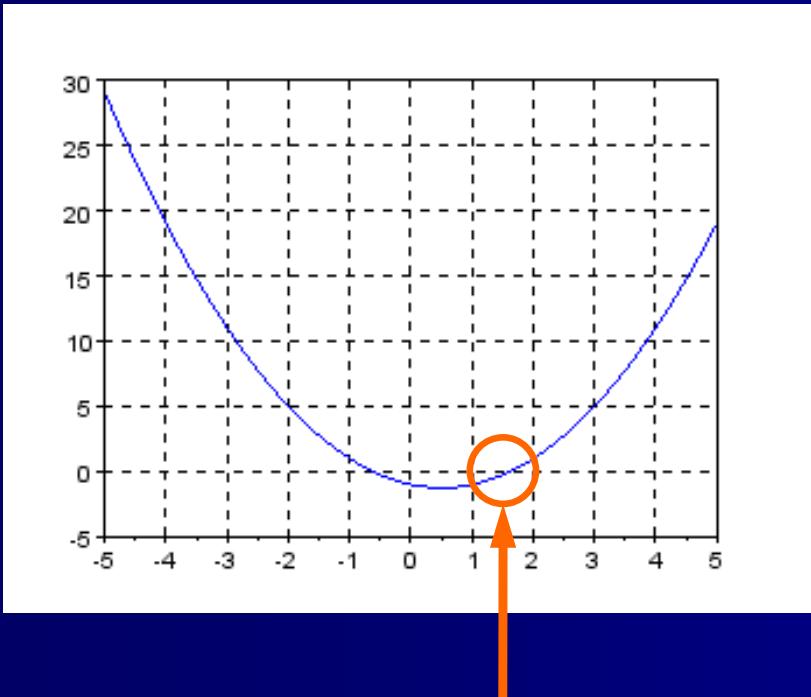
Here the subroutine is called twice more, first with the input argument a , then b

Interesting way of finding the root location. Later we'll do the same using `fsolv()`



```
// subroutine1.sce  
  
// Compute & plot the function y = x^2-x+1 in the /  
// range [-5,5] and determine its positive root. /  
// Assume we know the root lies in the range [1,2] /  
  
clear, clc, clf;  
  
// SUBROUTINE para():  
//-----  
function y=para(x); // Subroutine declaration  
y = x^2-x-1 // Equation (parabola)  
endfunction  
  
// MAIN script:  
//-----  
x = linspace(-5,5,100); // Range of interest  
plot(x,para) // Call subroutine and plot  
xgrid; // Add grid  
  
a=1; b=2; // Search limits  
while b-a > 10^(-4) // Accuracy for searching root  
    c = (a+b)/2; // Midpoint of limits  
    if para(a)*para(c)>0 then // IF (lower)*(midpoint)  
        // is positive  
        a=c; // THEN change lower limit  
    else  
        b=c; // ELSE change upper limit  
    end  
end  
disp("The root lies between "... // Output root limits  
     +string(a)+" and "+string(b))
```

Demo (2/2): plot, printout & comments



The root lies between 1.617981 and 1.618042

- This demo was borrowed from the pamphlet "Scilab pour les Lycees"
- Notice that the calling command was abbreviated to the extent possible. Instead of `plot(x,para)` we could write:
 $a = \text{para}(x);$
`plot(x,a)`
- Subroutines have to be declared before the calling main part of the script
- Later we'll see scripts with multiple subroutines, told apart by their names. Subroutines can also be nested (next slide) and can call other subroutines

Nested subroutines

- Subroutines can be nested—if it is worth the added complexity
- The example shown here computes the equation

$$y(x) = (\sin(x) + 2\%pi)^2 - \sqrt{(\sin(x) + 2\%pi)} + 3^2 \rightarrow$$

in three steps: First $\sin(x)+2\%pi$ is calculated. The result is then squared and subtracted with the square root of itself. Finally, $3^2=9$ is added

- Plug in the equation in a calculator and yes, it gives the same answer. I prefer this old fashioned method

```
-->function y = nested(x)
-->    a = sin(x) + 2*%pi;
-->    function y = inner(x);
-->        y = x^2 -sqrt(x);
-->    endfunction
-->    y = inner(a) + 3^2;
-->endfunction
```

```
-->value = nested(%pi/3)
value =
57.437413
```

```
--> (sin(%pi/3)+ 2*%pi)^2 - sqrt(sin(%pi/3) + 2*%pi) + 3^2
ans =
57.437413
```

The deff() primitive

- The deff() primitive can be used to define **simple functions** (it resembles Matlab's inline() function)
- deff() is therefore used in local subroutines
- The syntax is:

```
deff ('y = function_name(x1,x2,...)', 'y=function expression')
```

- Below is the same equation computed on the Console with function and deff() alternatives:

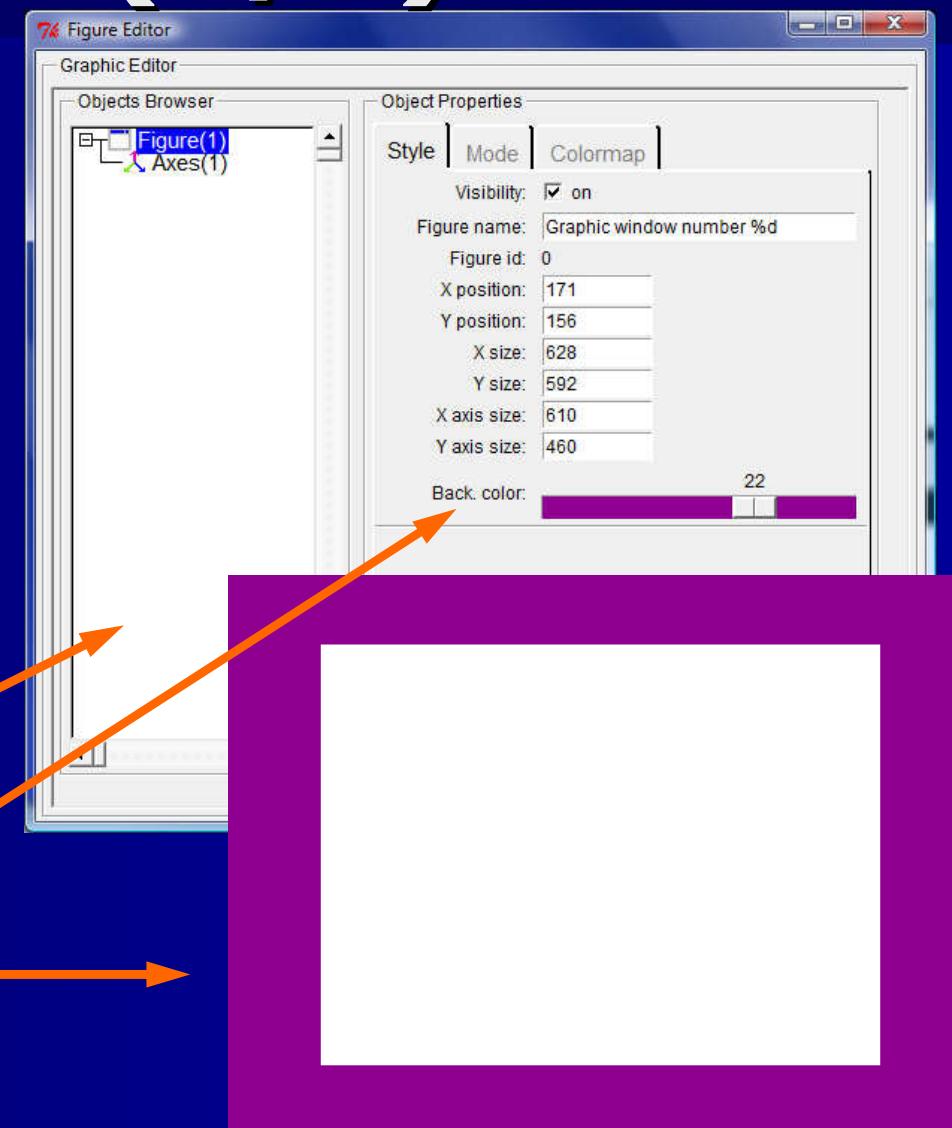
```
-->deff('y = f(x)','y = x^2+x-1')  
  
-->f(2), f(-5)  
ans =  
  
5.  
ans =  
  
19.
```

```
-->function y = g(x); y = x^2+x-1 endfunction  
  
-->g(2), g(-5)  
ans =  
  
5.  
ans =  
  
19.
```

Note the semicolon!
(Scilab understands if you skip it after the second y expression)

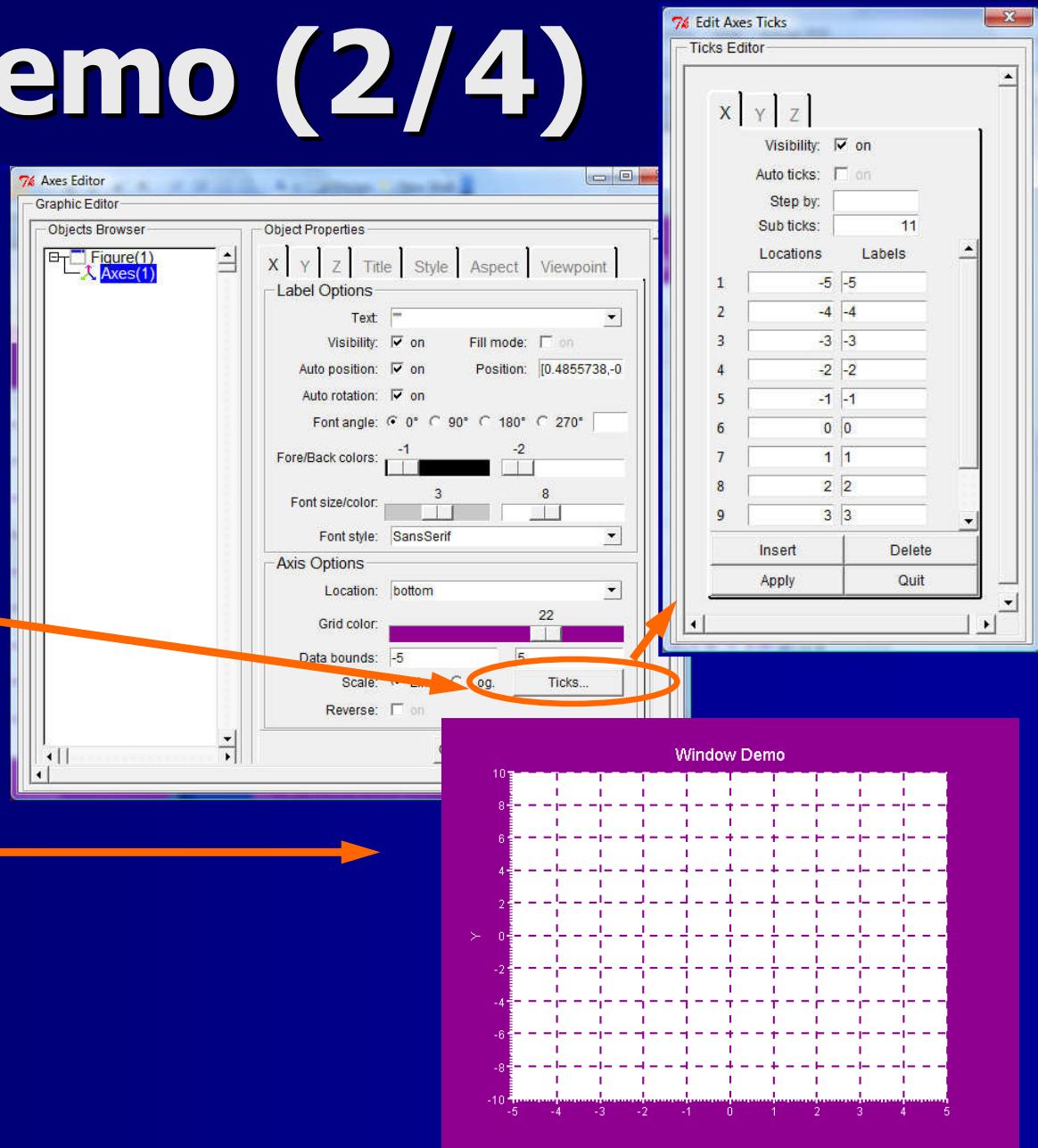
Global subroutines: window demo (1/4)

- This demo creates a global subroutine (function) for a reusable graphics window
- **First**, open Scilab's Graphics Window (one alternative is by the command `gcf()`; on the Console). The empty Graphics Window pops up
- **Next**, On the Graphics Window, Click: Edit/Figure properties to open the Figure editor (this has been explained before)
- **Third**, select a suitable Back color (e.g. 22 for ecclesiastic violet) and you can see the frame on the Graphics Window (the bar goes only to 32, gray is not an option)



Global subroutines: window demo (2/4)

- **Fourth**, you need to play with the Figure Editor for quite a while to fill in all details
- **Note** the Ticks... button on the Graphics Editor. It opens a separate window in which you can define & label grid lines
- **Later**, when all details match, you reach something like this
- **Finally**, save it with the help of the Graphics Window. Click: File/Save...



Global subroutines: window demo (3/4)

- Scilab answers on the Console:

Figure saved.

- So it is as a subroutine. I called it window_demo.scg . Note the ending .scg . Not .sce or .sci. It's g for graphic
- Then we need a main script that uses global subroutine
- Note that I call the Entity handle e=gce(). It simplifies compared to the path needed if calling the Axes handle, as was done in Chapter 7

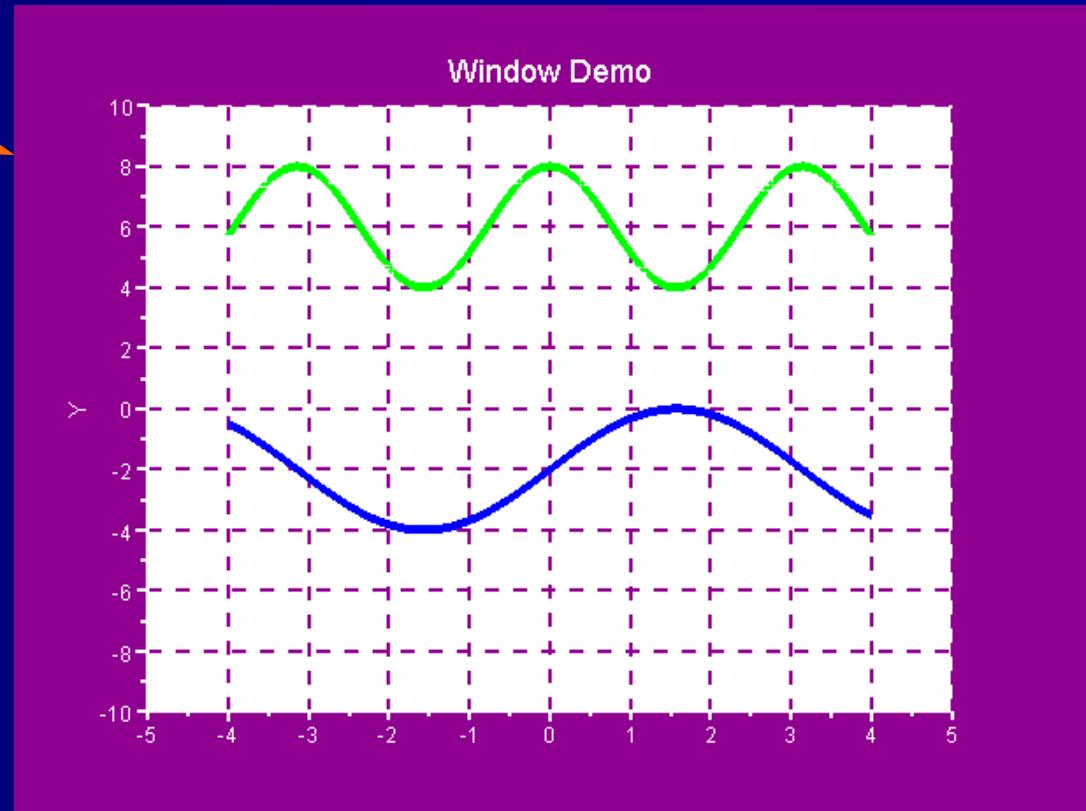
```
// reuse_function.sce
// Reusing graphics function,           /
// definitions with handle commands   /
clear,clc;

// SUBROUTINE, load function:
//-----
load('window_demo.scg');

// MAIN, define and plot:
//-----
x = [-4:0.01:4];          // Horizontal extension
y1 = 2*sin(x) - 2;        // First equation
plot2d(x,y1, style=2);    // First plot, blue
e=gce();                  // Get Entity handle
e.children.thickness=5;    // Polyline size
y2 = 2*cos(2*x) + 6;      // Second equation
plot2d(x,y2, style=3);    // Second plot, green
e=gce();                  // Get Entity handle
e.children.thickness=5;    // Polyline size
```

Global subroutines: window demo (4/4), plot

- Do you have problems with the plot?
- If yes, make sure that you defined the window correctly
- For instance, if you do not correctly fill both sides in the Edit Axes Ticks list there will be some funny grid locations
- Make also sure that the data bounds $([-5, 5], [-10, 10])$ are defined in the Axes Editor/Axes Options window



I typed the wrong handle call `gca()`; and
Scilab crashed definitely. Reboot ...

Comment: multiple plots with a single command

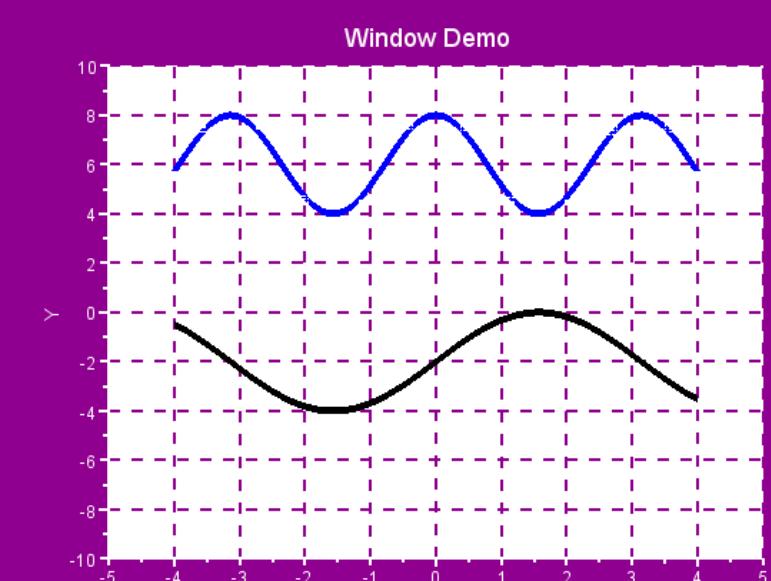
- It is possible to plot multiple graphs with a single command by defining the function arguments as column vectors

- Here is the modified plot command of the previous window demo, compressed to

```
plot2d(x,[y1',y2'])
```

- The result is shown here. Scilab automatically picks different colors for the graphs
- Scilab warns if the Graphics Window is docked

```
y1 = 2*sin(x) - 2;           // First equation  
y2 = 2*cos(2*x) + 6;         // Second equation  
plot2d(x,[y1',y2'])        // Plot both  
e=gce();                     // Get Entity handle  
e.children.thickness=5;       // Polyline size
```



11. Flow control

Flow control (conditional branching, programming) bring important new structures



[Return to Contents](#)

Introduction

- We have now come to the line that separates boys from men
- Some examples using flow control (also known as **conditional branching** and **programming**) have come up in the preceding chapters, but the nature of conditional branch commands like if ... then ... else has not been discussed*
- Flow control together with subroutines are needed for serious practical simulations
- We shall therefore take a good look at the most important aspects of flow control
- Note, however, that **loop operations are slow**. We should aim for **vectorized operations** if a task requires a lot of loop iterations (there is a brief discussion on the subject in Chapter 18)

*) Historical note: Konrad Zuse, the German who built the first real computer during WW II (using over 2000 relays because he did not trust vacuum tubes), got everything right except for conditional branches.

Flow control constructs

The following are main constructs that you should be familiar with:

Branch commands:*

```
for ... (if ... else) ... end  
while ... (if/then/else) ... end  
if ... (elseif ... else) ... end  
select ... case (... else) ... end  
break ... continue  
try ... catch ... end
```

Logical operators:

&	and
or /	or
~	not

Comparison operators:

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

Logical/Boolean constants:

%t or %T	true
%f or %F	false

*) Rumor has it that Scilab will see select ... case renamed switch ... case in line with Matlab.

for ... end

- The for ... end loop repeats a group of statements a **predetermined number of times**. The general expression of for ... end is:

```
for variable = initial_value:step:final_value  
    // foo  
end
```

No semicolon!

- As seen here: →

"square" cannot
be used as
variable name
since square()
is a Scilab function

```
// for-end_demo.sce  
  
// Compute the square root and square /  
// of odd integers from 1 to 8 /  
  
n = 8;  
for k = 1:2:n  
    root = sqrt(k);  
    quadrat = k^2;  
    disp([k, root, quadrat])  
end
```

1.	1.	1.
3.	1.7320508	9.
5.	2.236068	25.
7.	2.6457513	49.

for ... if ... else ... end

- for ... end can be nested with if/else conditions to allow for execution of alternate statements:

```
for variable = initial_value:step:final_value
    if condition
        // foo
    else
        // foo
    end
end
```

- The next few slides demonstrate a case where random Gaussian “noise” is generated, sorted, and reported both verbally and with a plot

for ... if ... else ... end: demo, script (1/3)

Only variables, nothing to comment

```
// for-if-else.sce

// -----
// The script generates Gaussian noise around a fixed signal.
// Each sample ("signal") is sorted according to whether it
// is within, above or below default variance limits (+/-1). The
// result is reported verbally with strings and is also plotted
// -----

clear,clc,clf;

// Define variables:
// -----
n = 500;      // # of for...end loops
above = 0;     // Signals above upper variance limit
below = 0;     // Signals below lower variance limit
within = 0;    // Signals within variance limits
ave = 3;       // Mean (average)
x = [] ;       // x axis vector
```

for ... if ... else ... end: demo, script (2/3)

Random generation
as discussed before



Note how the signal
array is read
element-by-element
as j goes from 1 to n



```
// Generate signal:  
// -----  
dt = getdate(); // Get date  
rand('seed',(531+n)*dt(9)+dt(10)); // Initialize random generator  
signal= ave + rand(1,n,'normal'); // Shifted Gaussian signal  
  
// Sort signal:  
// -----  
for j = 1:1:n  
    if signal(1,j) > ave+1 then // Default variance = +/-1  
        above = above + 1;  
    elseif signal(1,j) < ave-1 // Count signal > mean+var  
        below = below +1;  
    else // Default variance = +/-1  
        within = within + 1;  
    end // Count signal < mean-var  
end // If within variance limits  
// mean-var <= signal <= mean+var
```

for ... if ... else ... end: demo, script (3/3)

Display on the
Console: Total,
mean, and variance
limits

This particular form
of multiple plots was
discussed earlier and
is worth keeping in
mind

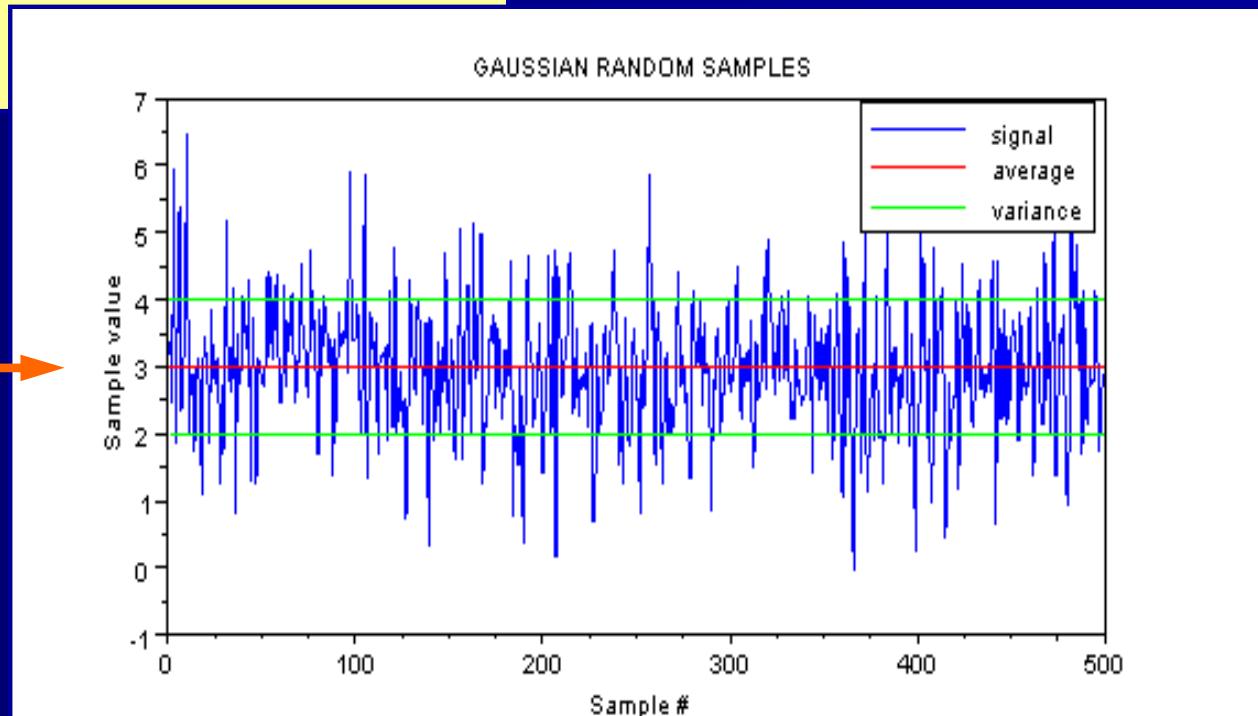
```
// Display result:  
// -----  
disp(['Result from generating', string(n), 'Gaussian distributed samples'])  
disp(['(signals) with mean =' string(ave) 'and variance = 1:'])  
disp([' - ' string(within) ' samples were inside variance limits,'])  
disp([' - ' string(above) ' above upper variance limit, and'])  
disp([' - ' string(below) ' below lower limit'])  
  
// Plot result:  
// -----  
x = [1:1:n]; // Array for x axis  
y1 = ave*ones(1,n); // Array for mean value  
y2 = (ave+1)*ones(1,n); // Array for upper variance limit  
y3 = (ave-1)*ones(1,n); // Array for lower variance limit  
rect = [0,ave-4,n+1,ave+4]; // Set plot window  
plot2d(x,signal,2,"011"," ",rect) // Plot samples  
plot2d(x,y1,5,"000") // Plot mean value  
plot2d(x,y2,3,"000") // Plot upper variance limit  
plot2d(x,y3,3,"000") // Plot lower variance limit  
legend('signal','average','variance');  
xtitle('GAUSSIAN RANDOM SAMPLES','Sample #','Sample value')
```

for ... if ... else ... end: demo, print & plot

```
!Result from generating 500 Gaussian distributed samples !
!(signals) with mean = 3 and variance = 1: !
! - 348 samples were inside variance limits, !
! - 75 above upper variance limit, and !
! - 77 below lower limit !
```

69.6% of the samples are within $\pm 1\sigma$ bounds. Quite ok for 500 samples

It can be seen that there are one or two samples outside the $\pm 3\sigma$ limit, as should be



for ... if ... else ... end: comments

- Note how the random data is collected by the `signal(:,n)` array. Compare this with the `x = [x,k]` construct that is used later in connection with the discussion on break & continue
- This example was a maddening experience: I just could not figure out how to make the plot work. After hours of attempts I found the problem: I had put the random function inside the `for ... end` loop
- What kept me on the wrong track for too long was that the data was collected correctly, but it was destroyed when it was brought outside the `for ... end` loop. However, that did not happen if there was a display command inside the loop. For instance, no semicolon after `signal(:,n)`. Speak of coincidences.....
- The problem went away when I finally understood to separate data generation from the data sorting loop

Lessons learned: Be careful with what you put inside loops

while ... end

- The while ... end loop repeats a group of statements an **indefinite number of times** under control of a logical condition
- The general form of while ... end is:

```
while condition  
    // foo  
    // loop counter, i.e. count = count + 1;  
end
```

- The code to the right determines from which value of k the expression $2^{-k} \leq \%eps$
- Be **careful** with condition, it can easily lock the simulation in an endless loop

```
-->k = 1;  
  
-->while 2^-k > %eps  
-->k = k+1;  
-->end
```

```
-->k  
k =
```

while ... if /then/else ... end

- The while ... end condition can be nested with an optional if ... then ... else instruction:

```
while condition_1
    if condition_2 then
        // foo
    else
        // foo
    end
    // foo
end
```

- The function on the next slide is for a game in which the user should guess a random number that the computer draws. The game finishes only with the correct guess

while ... if /then/else ... end: demo

```
// game.sci

// The function draws a random number in the      /
// range [1,M] that the user should guess.      /
// Game finishes when correct number is found   /

clear,clc;

M=30;                      // Upper limit of numbers
number=floor(1+M*rand());    // Draw a random number
disp('Guess a positive integer in the range ');
disp([1,M]);                // State range of random numbers
guess=input('You guess: ');  // User's guess

while (guess~=number)        // Start while condition
    if guess>number then
        disp('Number is too big');
    else
        disp('Number is too small');
    end
    guess=input('You guess: '); // User's next guess
end                          // End while condition

disp('Correct!');
```

Loop



Save the script, load it
into Scilab (on Editor),
type the function name
on the Console



-->guess_a_number

Guess an integer in the range

1. 30.

You guess: 15

Number is too small

You guess: 22

Number is too big

You guess: 17

Number is too small

You guess: 19

Correct!

Comments on interactivity

- The previous demo showed examples of interactive use of strings
 - To **instruct the user**:
`disp('Guess an integer')`
 - To **accept user inputs**:
`guess = input('You guess: ')`
- To the user the `input()` prompt is not very clear since the text string only pops up—it should at least blink. One must therefore try to find expressive text messages. Perhaps the following would be better in the previous case:

`guess = input('Now, Sir/Madame, type your guess: ')`
- Interactive text strings is a simple form of human-machine interfaces; Graphical User Interfaces (GUIs) more are advanced and will be discusses in Chapter 15 (there was a case already in Ex 1-3)

foo ... do ... end

- The do **keyword** can be used inside for and while **instructions** to separate the loop variable definition (condition) and the instructions. The keyword then can be used with if and while
- The following are examples of for ... do ... end and while ... do/then ... end:

```
-->n = 9;
```

```
-->for k = 1:1:3 do
```

```
-->n = n - 3
```

```
-->end
```

```
n =
```

```
6.
```

```
n =
```

```
3.
```

```
n =
```

```
0.
```

```
-->n = 9;
```

```
-->k = 1;
```

```
-->while k <= 3 do
```

```
-->n = n -3
```

```
-->k = k + 1;
```

```
-->end
```

```
n =
```

```
6.
```

```
n =
```

```
3.
```

```
n =
```

```
0.
```

```
-->n = 9;
```

```
-->k = 1;
```

```
-->while k <= 3 then
```

```
-->n = n-3
```

```
-->k = k + 1;
```

```
-->end
```

```
n =
```

```
6.
```

```
n =
```

```
3.
```

```
n =
```

```
0.
```

if ... (elseif/else) ... end

- The if statement evaluates a logical expression (condition) and executes a group of statements when the expression is *true*
- The optional elseif and else keywords provide for the execution of alternate groups of statements

```
if condition_1
    // foo
elseif condition_2
    // foo
.....
else
    // foo
end
```

if ... elseif/else ... end: demo

The following function computes the n:th term of the Fibonacci sequence when n is given:

```
// fibonacci.sci  
  
// Gives the n-th term of the Fibonacci /  
// sequence 0,1,1,2,3,5,8,13,... /  
  
funcprot(0)           // Suppress redefinition warning  
function [K] = fibonacci(n)  
    if n==1            // Begin if-elseif-else-end  
        K = 0;  
    elseif n==2          // Condition to proceed, n > 2  
        K = 1;  
    elseif n>2 & int(n)==n // Check if n is an integer >2  
        K = fibonacci(n-1) + fibonacci(n-2); // Compute Fibonacci #  
    else                // Previous conditions not met  
        disp('error! -- input is not a positive integer'); // Error message  
    end                  // End of if-elseif-else-end  
endfunction
```



Save the script, load it into Scilab (on Editor), type on the Console the function name with the n argument (Hint: do not use a large value!)



```
-->fibonacci(8)  
ans =  
  
13.
```

Check what happens for n < 1

select ... case ... else ... end

- The select ... case ... else ... end construct executes the first case that matches the stated condition
- If no match is found it executes the else statement
- The advantage of select ... case ... else ... end is that it allows us to avoid multiple if statements

```
select condition
    case 1
        // foo
    case 2
        // foo
    .....
    else
        // foo
end
```

Hint: Use select ... case when if ... elseif ... else threatens to become too complex

Note: select ... case is called switch ... case in Matlab (may be changed in Scilab)

select ... case ... end: demo, script

- Some textbooks on Matlab presents this as the drunk sailor problem. It demonstrates a random walk, one fixed step at a time
- The whole process is performed in a single function (`randwalk(steps)`) that has to be executed from the Console
- In this case there is no problem with having the random generator inside the `for ... end` loop
- The script plots two marks ($\circ -$) for each step, although they cannot be distinguished on the plot on the next slide

```
// randomwalk.sce
-----
// Creates a track of marks that proceed randomly
// in the x,y plane. The walk starts at the origin
// and proceeds for a predetermined number of steps
// either up, down, right, or left
-----

clear,clc,clf;
funcprot(0);

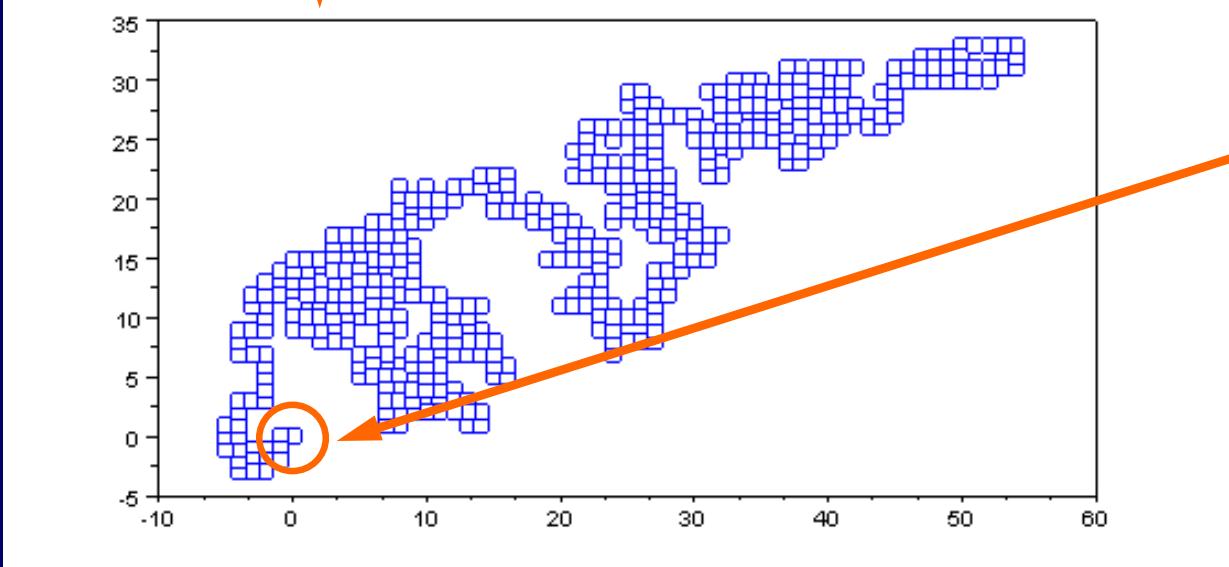
function randwalk(steps)
x=zeros(1,steps+1);           // Counter for x track
y=zeros(1,steps+1);           // Counter for y track
for k=1:steps
    direction=floor(4*rand()); // Draw random move
    select direction
    case 0 then
        x(k+1)=x(k)+1;       // Move right
        y(k+1)=y(k);
    case 1 then
        x(k+1)=x(k)-1;       // Move left
        y(k+1)=y(k);
    end
end
```

select ... case ... end: demo, script & plot

After loading the script into Scilab, the function has to be run from the Console

-->randwalk(1000)

```
case 2 then  
    x(k+1)=x(k);  
    y(k+1)=y(k)+1;      // Move up  
case 3 then  
    x(k+1)=x(k);  
    y(k+1)=y(k)-1;      // Move down  
end  
end  
clf  
plot(x,y,'o-');  
endfunction
```



The starting point is always the origin (I have run this simulation numerous times and Scilab seems to prefer to go in the northeastern direction)

break & continue

The break command:

- break lets you exit early from a for ... end or while ... end loop, or from within an if ... end statement
- Execution continues from the line following the end statement
- In nested loops, break exits only from the innermost loop

The continue command:

- continue is a forced return to the start of a for ... end or while ... end loop (not if ... end loops!)
- Statements between continue and the end of the loop will be neglected

```
-->k = 0;  
  
-->while 1 == 1,  
-->k = k + 1;  
-->disp(k);  
-->if k > 6 then  
-->    break  
-->end;  
-->end
```

```
1.  
2.  
3.  
4.  
5.  
6.  
7.  
  
-->for j = 1:2  
-->x = [];  
-->for k = 1:10  
-->if k>j+1 & k<=8 then  
-->    continue  
-->end  
-->x = [x,k];  
-->end  
-->x  
-->end  
x =  
  
1. 2. 9. 10.  
x =
```

1. 2. 3. 9. 10.

break: demo

```
// break.sce

// Input m positive integers that are summed /
// up, but the program breaks if the input      /
// is not a positive integer                  /

clear,clc;

n = input('Give amount of numbers to sum_');
summa = 0;          // Reset summa counter
for i = 1:n
    number = input('Give next number_');
    if number < 0           // Condition: number ~< 0
        disp('wrong----negative value!');
        break;
    end
    if number ~= int(number) // Check if integer
        disp('wrong----not an integer!');
        break;
    end
    summa = summa + number; // Sum up
end
disp(['Accumulated error-free sum is:' string(summa)]);
```

Give amount of numbers to sum_3

Give next number_13

Give next number_17

Give next number_7

!Accumulated error-free sum is: 37 !

Give amount of numbers to sum_3

Give next number_17

Give next number_2

Give next number_-1

wrong----negative value!

!Accumulated error-free sum is: 19 !

Give amount of numbers to sum_4

Give next number_ 18

Give next number_ 3.3

wrong----not an integer!

!Accumulated error-free sum is: 18 !

try ... catch ... end

- With no errors, the code between try and catch is executed
- If an error occurs, execution immediately shifts to the code between catch and end:

```
try
    // foo
    // If an error occurs in this part....
catch
    // .... execution continues here
    // foo
end
```

- Typically the code between catch and end informs of an expected error, e.g. disp('----warning: cannot access the function----')

12. Examples, Set 4

The first three examples relate
to Chapter 10, the rest to
Chapter 11

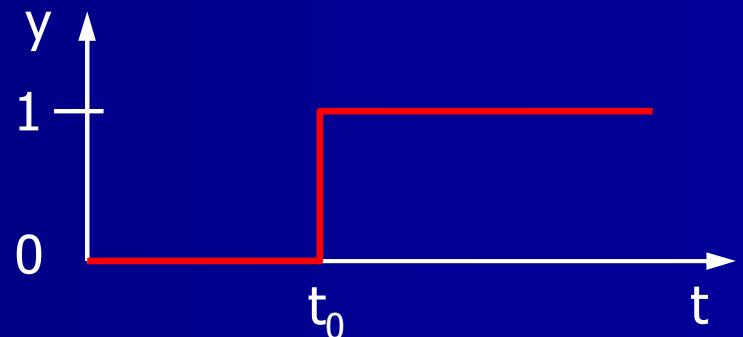


[Return to Contents](#)

Example 4-1: step function, unit step (1/2)

- Step functions are useful in many practical applications
- As mentioned in Ex 2-3, Scilab lacks a separate function for creating (unit) steps, but we can form them indirectly (in Ex 2-3 it was done with a unit vector)
- Here we shall look at two cases where a step is needed
- In the first demo a step is created with a user defined function that includes the `sign()` function (Help is of no help here, you don't understand what it says about `sign()`)

Unit step:



$$y(t) = \begin{cases} 0, & t < t_0 \\ 1, & t > t_0 \end{cases}$$

Ex 4-1: step function, unit step (2/2)

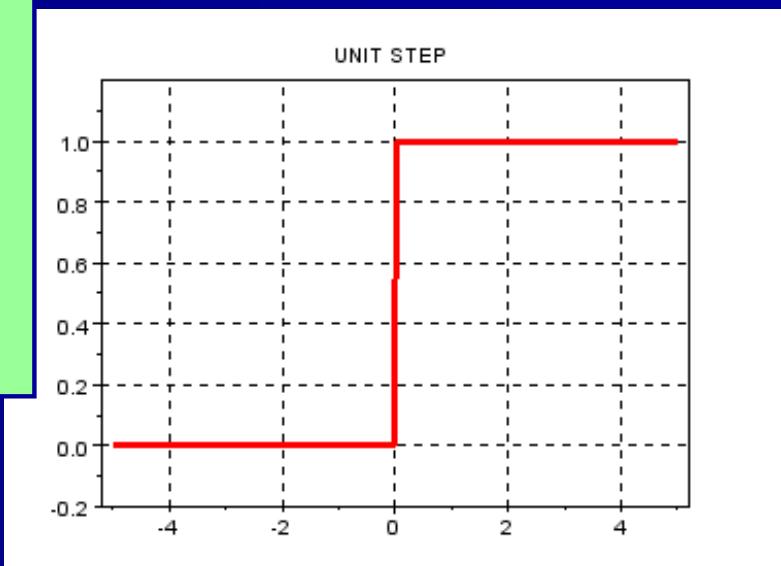
```
// step_sign.sce
// Plot a sign() function that is shifted by 1 /
// & compressed by 0.5 to give a unit step /
clear,clc,clf;
funcprot(0);

x = linspace(-5,5,400);
deff('y=u(x)','y=0.5*(1+sign(x))') // Define sign() function,
// shift & compress as needed

rect = [-5.2,-0.2,5.2,1.2];           // Define plot frame
plot2d(x,u(x),5,'011',' ',rect)    // Plot inside frame
xgrid()                                // Add grid to plot

a=gca();
a.title.text="UNITY STEP";
a.children.children.thickness=3;         // Get axes handle
// Add title
// Increase line thickness
```

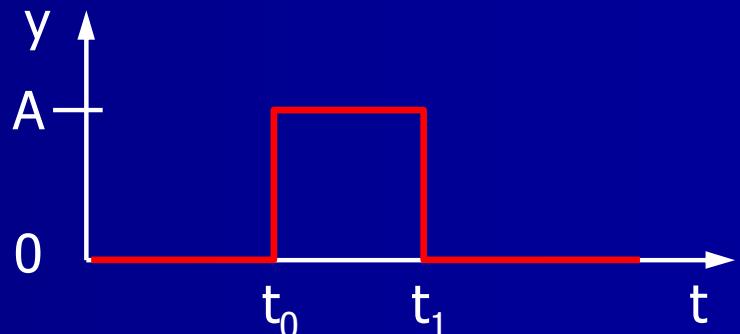
Note how the sign() function is shifted (addition by 1) and compressed (multiplied by 0.5) to get the required unit step



Ex 4-1: step function, rectangular pulse (1/2)

- The second case is a rectangular pulse with amplitude A as shown in the figure
- In this case we do it without a user defined function, since it leads to a simpler script
- The plot command can also be simplified somewhat

Rectangular pulse:



$$y(t) = \begin{cases} A, & t_0 \leq t < t_1 \\ 0, & \text{otherwise} \end{cases}$$

Ex 4-1: step function, rectangular pulse (2/2)

```
// rect_pulse.sce  
  
// Plot a rectangular pulse with /  
// width 3 < t < 5 and amplitude 2 /  
  
clear,clc,clf;  
  
t = 0:0.01:10;  
deff('y=u(t)','y=1*(t>=0)');  
  
y = 2*(u(t-3) - u(t-5));  
plot2d(t,y,5,rect=[0,0,8,2.5]) // Plot  
xgrid()  
  
f=gcf();  
f.children.thickness=2;  
a=gca();  
c=a.children;  
c.children.thickness=3;
```

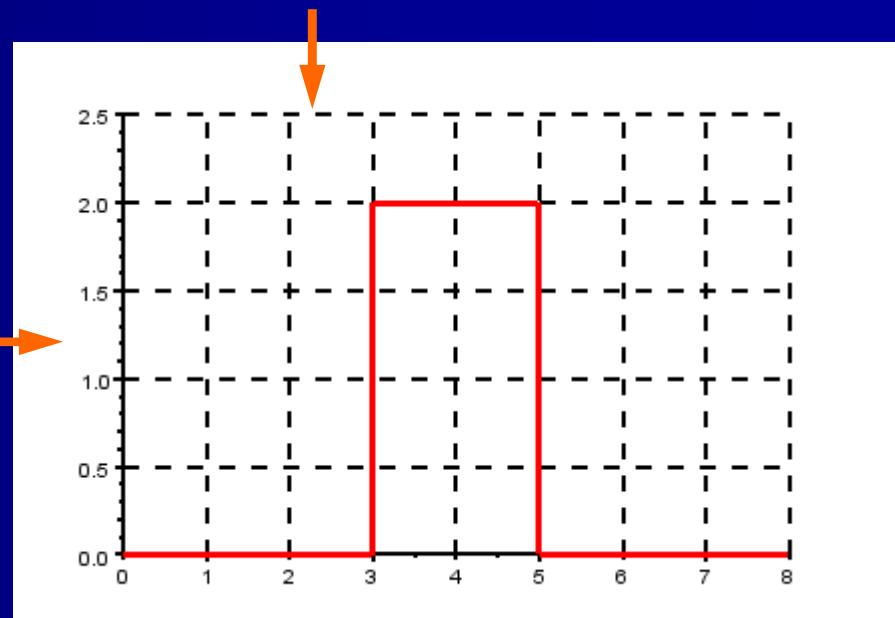
// Define u(t)
// Define pulse
// Plot
// Add grid

// Figure handle
// Figure lines

// Compound handle
// Line thickness

Note that the argument `u()` does not have to be defined separately
rect can be defined even if style, strf, and leg are absent

Thick figure lines & graph with increased line thickness



Ex 4-1: step function, comments

- In the first case (unit step) the handle command for line thickness is
a.children.children.thickness=3;

In the second case (rectangular pulse) Scilab did not accept this form and it had to be rewritten as

```
c=a.children;  
c.children.thickness=3;
```

I have no idea why this is the case and Help certainly is of no help

- In the latter case I happened to write the script without the deff() function, and for a while everything came out all right. But when I added handle commands Scilab decided that the variable u is undefined. The KISS principle (Keep It Simple, Stupid) did not apply in this case

Example 4-2: cones in a 3D space

- This example is adapted from Chancelier et al., pp. 163-166
- The script is **quite complex** with four subroutines and three separate plotting structures, which makes it difficult to follow the transfer of parameters. Changing an argument can have unexpected consequences
- Some aspects of the script have not been covered before and will be left without detailed discussion here as well (see however Ex 4-3)
- The object, a cone (the book mentions and shows a vase), is plotted in three separate positions using **lateral shifts, rotation, and non-rotational dilation (homothety)** of the objects
- The cones are shaded using **handles** that are called through the `gce()` command
- Scilab functions used for the first time: `diag()`, `eval3dp()`, `graycolormap()`, `isoview()`,* `size()`

*) The function `isoview()` is obsolete. The Help Browser recommends using `frameflag=4` instead.

Ex 4-2: script (1/4)

- vertical[] tells how to move along the z axis in later calculations. Note the increasing and decreasing values that will cause **problems** for shading
- The function cone() generates the cone in case. Example #12 discusses how it is done

```
// cone_manipulation.sce

// ****
// The script generates and plots a cone with its
// tip at the origin. It plots two copies of the
// cone, one shifted and one shifted & rotated
//
// ****

clear,clc,clf;

// Vertical reach of 3D object:
vertical=[0,1.0,1.6,2.5,2.2,2,1.6,0.9,0.5,0.3,0.3,0.4,0.6,1,1.4,
1.7,0,0,0.1,0.4,0.8,1.1,1.4,1.7,1.9,2.2,2.4,2.7,3,3.3,3.7,3.9]/2;

// SUBROUTINE 1: Generation of 3D object:
//-----
function [x,y,z]=cone(reach,Z) // Generation of a 3D object
    x=vertical(1,Z).*cos(reach) // Extension along x axis
    y=vertical(1,Z).*sin(reach) // Extension along y axis
    z=vertical(1,Z).*ones(reach) // Vertical (z) axis
endfunction
```

Ex 4-2: script (2/4)

- Lateral shifts of objects are handled by the → function `translation()`
- Non-rotational dilatation of objects is the task of `homothety()` →
- `rotation()` creates a matrix for rotating objects around the three axes →

Those are the four user defined functions

```
// SUBROUTINE 2, Lateral shifts:  
//-----  
function XYZ=translation(vect,xyz)  
    XYZ=(vect(:)*ones(1,size(xyz,2))) + xyz // Translation vector  
endfunction  
  
// SUBROUTINE 3, Non-rotational dilation: (center =  
// center of dilation, f = dilation factor)  
//-----  
function XYZ=homothety(center,f,xyz)  
    XYZ=translation(center,diag(f)*translation(-center,xyz))  
endfunction  
  
// SUBROUTINE 4, Rotation:  
//-----  
function XYZ=rotation(angle,xyz)  
    angle=angle/180*%pi; // Angle of rotation around axes  
    c=cos(angle);  
    s=sin(angle);  
    Rx=[1 0 0;0 c(1) s(1);0 -s(1) c(1)] // Rotation along x axis  
    Ry=[c(2) 0 s(2);0 1 0;-s(2) 0 c(2)] // Rotation along y axis  
    Rz=[c(3) s(3) 0;-s(3) c(3) 0;0 0 1] // Rotation along z axis  
    XYZ=Rx*Ry*Rz*xyz  
endfunction
```

Ex 4-2: script (3/4)

- eval3dp() transforms the smooth surface that cone() creates into a composition of quadrangular facets
- Here we plot the basic cone, which has its tip in the origin. The exterior and interior of the cone should have different shades
- Objects are manipulated by vectors created by the earlier user defined functions

```
// ----- MAIN ----- //
// --- STEP 1: CREATE & PLOT BASIC CONE --- //

// Superimpose rectangular facets:
//-----
[xv,yv,zv]=eval3dp(cone,linspace(-%pi,%pi,20),1:10);
f=gcf();                                // Get Current Figure, create figure
f.color_map=graycolormap(32); // Select color

// Plot basic cone with tip at the origin:
//-----
plot3d(xv,yv,zv)
e1=gce();                                // Get Current Entity handle
e1.color_mode = 24;                      // Object exterior: light grey
e1.hiddencolor = 30;                     // Object interior: dark grey

// --- STEP 2: MANIPULATE & PLOT OTHER CONES --- //

// Object manipulations parameters:
//-----
XYZ=[xv(:)';yv(:)';zv(:)'];           // XYZ = 3 x N matrix
XYZT=translation([1 3 -3],XYZ);        // Lateral shifts
XYZH=homothety([5 7 -3],1.5*[1 1 1],XYZT);
                                         // Non-dilational rotation
XYZR=rotation([-15 15 30],XYZT);      // Rotation
```

Ex 4-2: script (4/4)

- Plot another cone, this one is zoomed up and sifted laterally. Same shading as before



```
// Plot second cone (enlarged):  
//-----  
plot3d(matrix(XYZH(1,:),4,-1),matrix(XYZH(2,:),  
    4,-1),matrix(XYZH(3,:),4,-1))  
e2=gce(); // Get Current Entity handle  
e2.color_mode = 24; // Object exterior: light grey  
e2.hiddencolor = 30; // Object interior: dark grey
```

- And the third plot, with the cone shifted laterally and rotated. Shading as before



```
// Plot third cone (rotated):  
//-----  
plot3d(matrix(XYZR(1,:),4,-1),matrix(XYZR(2,:),  
    4,-1),matrix(XYZR(3,:),4,-1))  
e2=gce(); // Get Current Entity handle  
e2.color_mode = 24; // Object exterior: light grey  
e2.hiddencolor = 30; // Object interior: dark grey
```

- Properties of the box around the cones is adjusted. Isometric scaling is "on" (check with Help for an explanation)

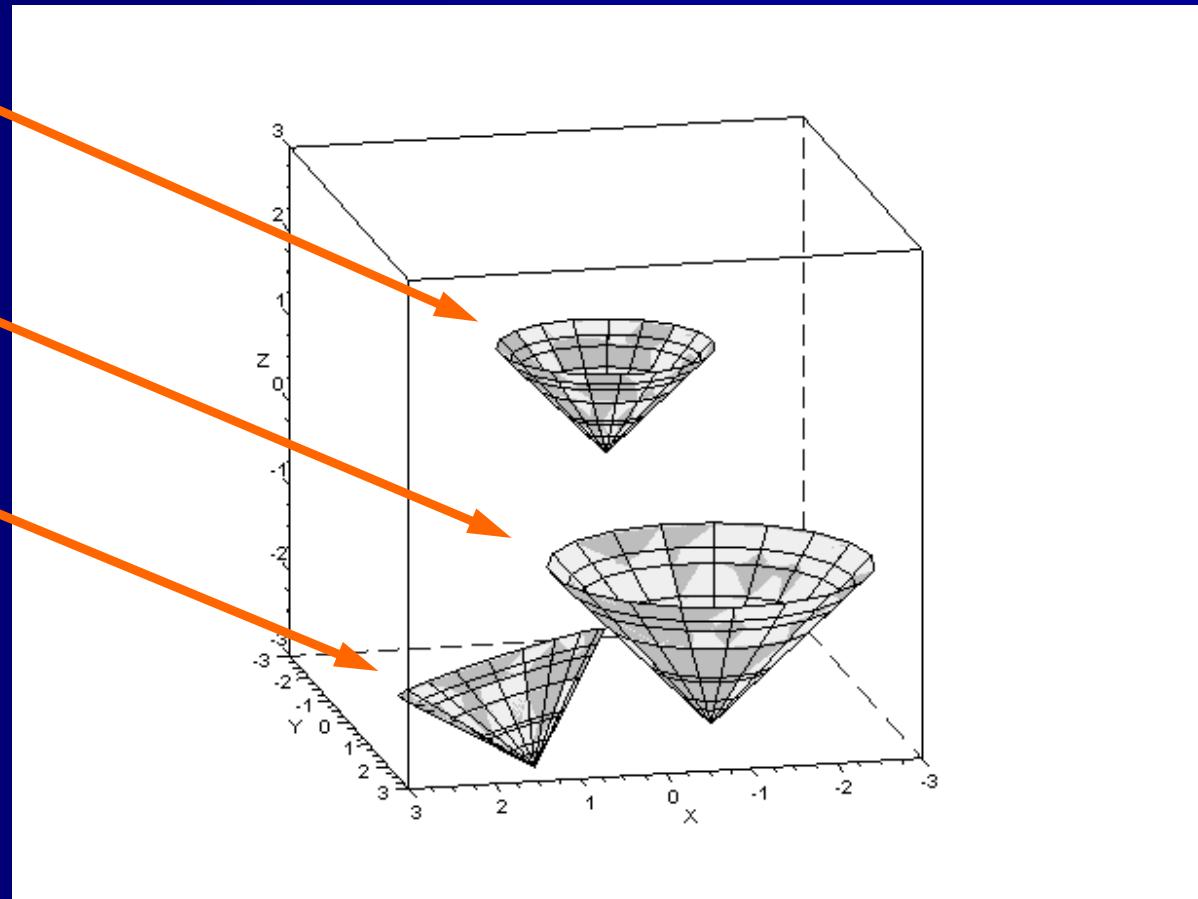


```
// ---- STEP 3: ADJUST THE BOX ---- //  
  
// Adjust Axes properties:  
//-----  
a=gca(); // Get Current Axes  
a.data_bounds=[-3 -3 -3;3 3 3]; // Box dimensions  
a.rotation_angles=[75 77]; // Rotation of the box  
a.isoview='on'; // Isometric scaling  
  
// ---- END OF MAIN ---- //
```

Ex 4-2: plot

- Original cone with tip at the origin
- Second cone, laterally shifted and enlarged
- Third cone, laterally shifted and rotated

And the shading of them is all **wrong**. See [Example 4-3](#) for an explanation



Ex 4-2: comments

- Chancelier et al. have not documented their examples too well, which in this case—together with errors in their solution—caused major problems when I tried to understand the script. **DO NOT UNDERESTIMATE THE NEED TO DOCUMENT PROGRAMS!** You may be the one that suffers when your code has to be changed, years after it was written
- The first requirement of documentation is liberal use of **comments** in the code
- Among the handle commands are some that have not been discussed before: `f.color_map=graycolormap`, `e1.color_mode`, `e1.hidden_color`, `a.rotation_angles`, and `a.isoview='on'` (recall however the `colormap` command that was used in Ex 3-5)

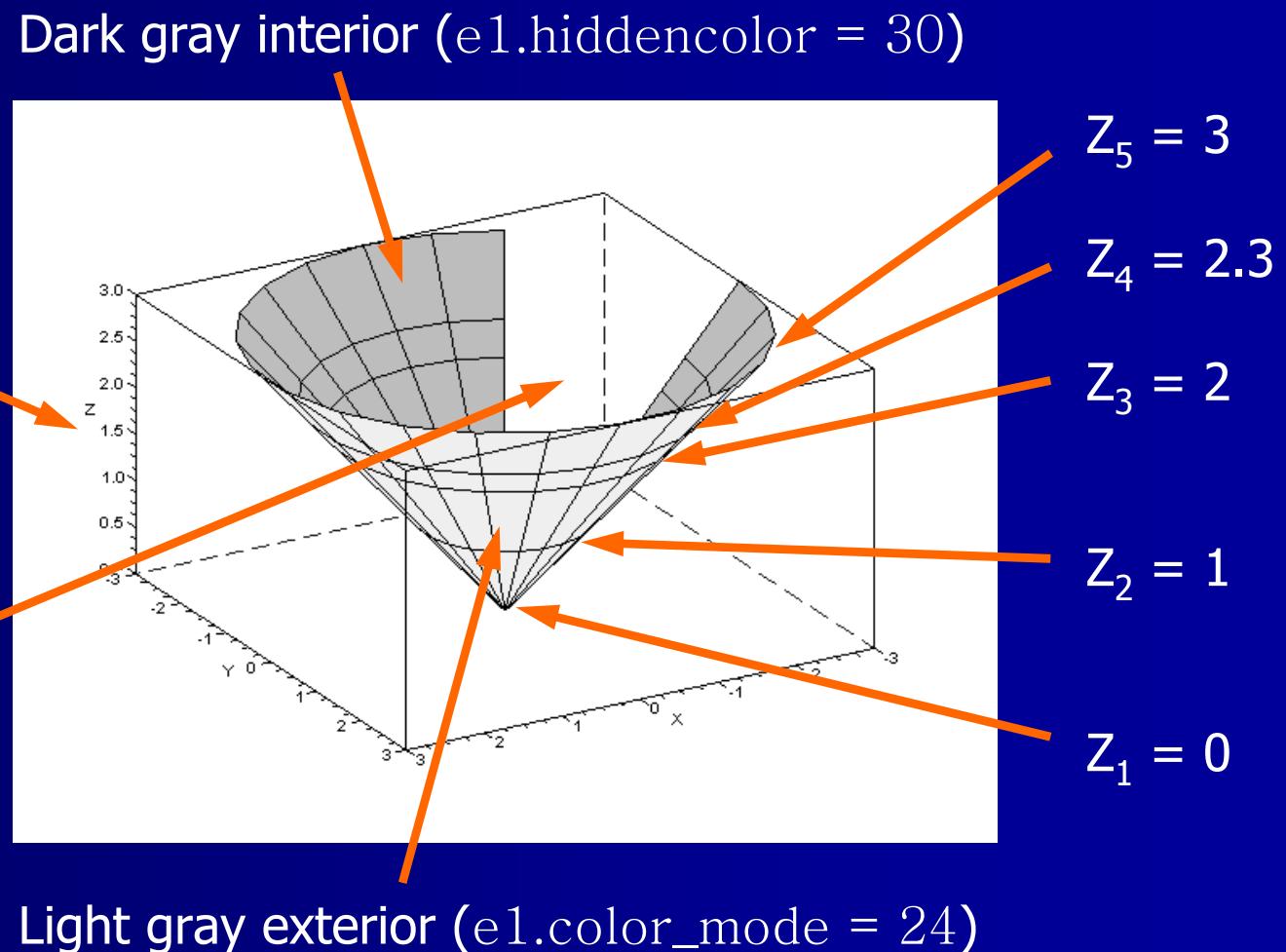
Example 4-3: how to generate a cone

- How was the cone in the previous example generated? The interplay between the matrix `vertical[]`, user defined function `cone()`, and facet generation function `eval3dp()` are not too obvious
- Let's simplify the case to a bare minimum →
- And look at the result on the next slide

```
// cone_creation.sce  
  
// A bare-bone eval3dp() script for plotting a 3D cone /  
  
clear,clc,clf;  
vertical=[0,1,2,2.3,3,4];           // Vertical reach of 3D object  
  
function [x,y,z]=cone(reach,Z)    // Generation of a 3D object  
    x=vertical(1,Z).*cos(reach)    // Extension along x axis  
    y=vertical(1,Z).*sin(reach)    // Extension along y axis  
    z=vertical(1,Z).*ones(reach)   // Vertical (z) extension  
endfunction  
  
[xv,yv,zv]=eval3dp(cone,linspace(-%pi/1.5,%pi,20),1:5);  
  
plot3d(xv,yv,zv,theta=60,alpha=70) // Plot object  
e1=gce();                         // Get current Entity handle  
e1.color_mode = 24;                 // Object exterior: light grey  
e1.hiddencolor = 30;                // Object interior: dark grey
```

Ex 4-3: plot

Box alignment defined by theta and alpha in plot3d()
Gap in the surface due to the argument
`linspace(-%pi/1.5, %pi, 20)`



Ex 4-3: discussion

- The cone is created by the linearly increasing radius R_z of x and y:

$$x = R_z \cdot \cos(Z_n)$$
$$y = R_z \cdot \sin(Z_n)$$

If you change the first element in `vertical[]` from 0 to 0.5, you'll see that the tip of the cone is cut off

- There are six elements in the vector `vertical[]`. The last one (4) is never used since the third argument in `eval3dp()` is 1:5, meaning that only the first five vector elements are needed. Hence the z axis of the plot is [0,3]
- I left a gap in the perimeter of the cone to demonstrate the role of the second argument in `eval3dp()`
- This example has **correct shading** of the object. The surface pattern in Ex 4-2 is no artistic creation but messed up due to overlapping Z_n values

Ex 4-3: how to transform the cone to a vase

- How do we create the vase that Chancilier et al. talk about?
- Quite obviously, we have to alter R_z in
$$x=R_z.*\cos(Z_n)$$
$$y=R_z.*\sin(Z_n)$$
- Here is one way to do it: by introducing a vector R_factor that compensates for the linear increase in R_z
- And the result is shown on the next slide

```
// vase_creation.sce

// A bare-bone eval3dp() script for plotting a 3D vase /

clear,clc,clf;

vertical=[0,1,2,2.3,3,4]; // Vertical reach of 3D object
R_factor=[1,1,0,-1.5,-1,0]; // Correction matrix

function [x,y,z]=cone(reach,Z) // Generation of a 3D object
R=vertical+R_factor; // Radius of vase, R=f(Z)
x=R(1,Z).*cos(reach) // Extension along x axis
y=R(1,Z).*sin(reach) // Extension along y axis
z=vertical(1,Z).*ones(reach) // Vertical (z) extension
endfunction

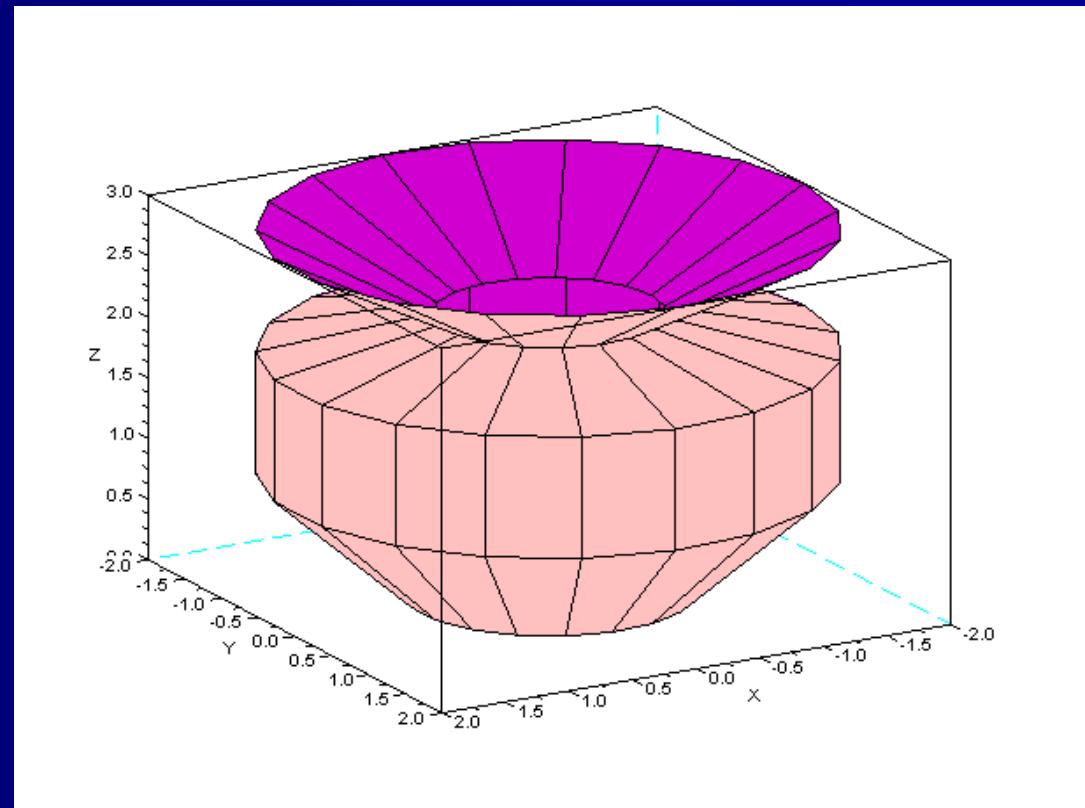
[xv,yv,zv]=eval3dp(cone,linspace(-%pi,%pi,20),1:5);

plot3d(xv,yv,zv,theta=60,alpha=70) // Plot object
e1=gce(); // Get Current Entity handle
e1.color_mode = 24; // Object exterior: light grey
e1.hiddencolor = 30; // Object interior: dark grey
```

Ex 4-3: vase plot

Not bad, eh?

But I have no idea where the pink & aniline colors came from, they bumped up when I executed the script after **Scilab had crashed**. The gray scale returned after I reloaded Scilab for a second time



Example 4-4: ballot engine for politicians

- The function on the next two slides is a ballot machine that help politicians decide on how to cast their vote
- The number of issues to vote on is entered and the code checks that the number is a positive integer
- Scilab then draws random numbers and transforms them to verbal votes (yes/no/abstain)
- The votes are finally put into groups of three
- The function demonstrates the use of select ... case ... end with a finishing modulo() statement
- It also shows the use of repeated if ... end statements (necessary or not?)

-->voting
Give number of issues to vote on_5

Now this is how you should vote:

yes

no

yes

abstain

no

-->voting
Give number of issues to vote on_-2.2

warning----must be > 0

Ex 4-4: script (1/2)

- A good part of the function commands are related to checking the validity of data
- The first check makes sure that the number entered by the user is > 0
- The next check is to make sure that n is an integer
- Pay attention to the abort commands!

```
// voting.sci  
// Ballot machine for politicians. The number /  
// of issues to be voted on is entered and /  
// Scilab tells how to vote on them. The /  
// answers are presented in groups of three /  
  
clear,clc;  
funcprot(0)  
  
function voting  
  
    // Give input and check entered number:  
    //-----  
    n = input('Give number of issues to vote on_ ');  
    if n <= 0 do          // # of votings must be > 0  
        disp('warning----must be > 0');  
        abort;  
    end  
    if n ~= int(n) do      // n should be an integer  
        disp('warning----not an integer!');  
        abort;  
    end
```

Ex 4-4: script (2/2)

- Generation of random numbers in the similar manner to Ex 1-3
- Then a select ... case ... end construct that transforms the random numbers to text strings
- Finally the string outputs are grouped into threes. Pay attention to how handy the modulo() function is!

```
// Create n random numbers 0,1 or 2:  
//-----  
dt=getdate(); // Get initial seed  
rand('seed',1000*dt(9)+dt(10)); // Seed random generator  
votes = floor(3*rand(n,1)); // Generate votes (0,1, or 2)  
  
// Transform random numbers to verbal votes:  
//-----  
disp('Now this is how you should vote:');  
for k = 1:n  
    select votes(k)  
        case 0 then  
            disp('yes'); // 0 = yes  
        case 1 then  
            disp('no'); // 1 = no  
        case 2 then  
            disp('abstain'); // 2 = abstain  
    end  
    if modulo(k,3)==0 // 3 votes given?  
        disp(' ') // Leave space after 3 rows  
    end  
end  
endfunction
```

Ex 4-4: comments

- Scilab has several commands related to forced termination of an ongoing process: `abort`, `break`, `exit`, `quit`, `return`, `resume`. Check with Help for details
- In this example I had some problems with jumping out of the program in the right manner:
 - According to Help Browser the `exit` command should end the current Scilab session—whatever it means. It turned out that `exit` performs more or less like the `break` command by only ending the present loop
 - `quit` is a brute that closes down Scilab
 - Trial and error showed that `abort` had the expected effect of jumping to the end of the function

Good old `GO TO` statement, where have you been all these years—and why do they give you such fancy names?

Example 4-5: nested structures, script

This script contains an if ... elseif ... else ... end structure nested within an while ... end structure
(read the title for an explanation of what the script does)

Note how min() and max() ensure that scale limits are not exceeded

while :: end {
 if ... }

```
// conditional.sce  
  
// Climb up or down the scale depending on /  
// input data ("u" or "d") without exceeding /  
// the limits. The process ends when "e" is /  
// pressed /  
  
scale = [1 2 3 4 5 6 7 8 9]'; // Define scale to climb  
i = 1; // Preset counter  
strg = ''; // strg = empty string  
while strg ~= 'e' // Until the "e" key is hit  
    disp(scale(i,:)); // Display location on scale  
    strg = input('Exit(e), Up(u), Down(d)?','string')  
    if strg == 'u' then // If "u" is hit  
        i = min(i+1, size(scale,1)); // One step up, until highest  
    elseif strg == 'd' then // But if "d" is hit  
        i = max(i-1, 1); // One step down, until lowest  
    elseif strg == 'e' then // If "e" is hit  
        break; // Jump out of the loop  
    else // Whatever else is hit  
        disp('---incorrect input---') // Present error message  
    end // End of if statement  
end // End of while statement  
disp('you hit e=Exit') // Exit message
```

Example 4-5: execution

The scale counter *i* is preset to 1 and increases/decreases depending on the entered data

Any input parameter except *u*, *d*, or *e* give an error message

The break command works well in this case

Homework: Modify the script by using the select ... case ... else ... end **structure** instead of if ... elseif ... else ... end.
Which solution is simpler?

```
1. Exit(e), Up(u), Down(d)?u  
strg =  
u  
  
2. Exit(e), Up(u), Down(d)?u  
strg =  
u  
  
3. Exit(e), Up(u), Down(d)?d  
strg =  
d  
  
2. Exit(e), Up(u), Down(d)?6  
strg =  
6  
---incorrect input---  
  
2. Exit(e), Up(u), Down(d)?u  
strg =  
u  
  
3. Exit(e), Up(u), Down(d)?e  
strg =  
e  
you hit e=Exit
```

13. Doing math on Scilab

Scilab contains functions for sophisticated mathematics. We'll stay with the simpler cases



[Return to Contents](#)

Math in earlier chapters

- Chapter 3: Complex numbers, vectorized functions, polynomials
- Chapter 4: Trigonometric functions, random functions
- Chapter 5: Matrices, matrix operations, matrix operators, symbolic computing, random generators
- Chapter 6: Linear equation systems with real coefficients
- Chapter 7: 2D and 3D functions, vector fields, histograms, rotation surfaces, logarithms, polar coordinates
- Chapter 8: Polynomial expressions
- Chapter 9: Application of matrices & trigonometric functions
- Chapter 10: Arithmetic and algebra
- Chapter 11: Logical expressions
- Chapter 12: Step functions, application of 3D vector spaces

"Do not worry about your problems with mathematics, I assure you mine are far greater." Albert Einstein

optim() & fsolve(): demo (1/4), the task

- The functions optim() and fsolv() give us tools by which to investigate nonlinear equations and/or equation systems:
 - optim() to find minima (and indirectly maxima)
 - fsolv() to find solutions (roots) to equations/equation systems
- optim() is a quite complex function, which is evident in the Help Browser's confusing description. Here we shall stick to a basic case by applying optim() and fsolv() to the equation

$$y = \sin(x)/((x - 0.1)^2 + 0.1)$$

- We solve the problem in two steps:
 - First by plotting the graph to get better understanding of the function, and simultaneously computing min and max values for y using optim()
 - Then we apply fsolve() to compute exact root locations with the aid of visual estimates from the plotted graph

optim() & fsolve(): demo (2/4), script

optim() requires a Scilab subroutine of the type [f,g,ind]=cost(x,ind). The numeric value of grad is irrelevant

Plotting is done with fplot2d(), which is quite similar to plot2d()

I do not know why there has to be a third numeric argument in list(), Scilab just requires something (I tried and cried...)

The second argument of optim(list(),0) defines the gradient that we are interested in

```
// optim_list.sce

// Investigation of minima and maxima of the function //
// sin(x)/((x-0.1)^2+0.1) //

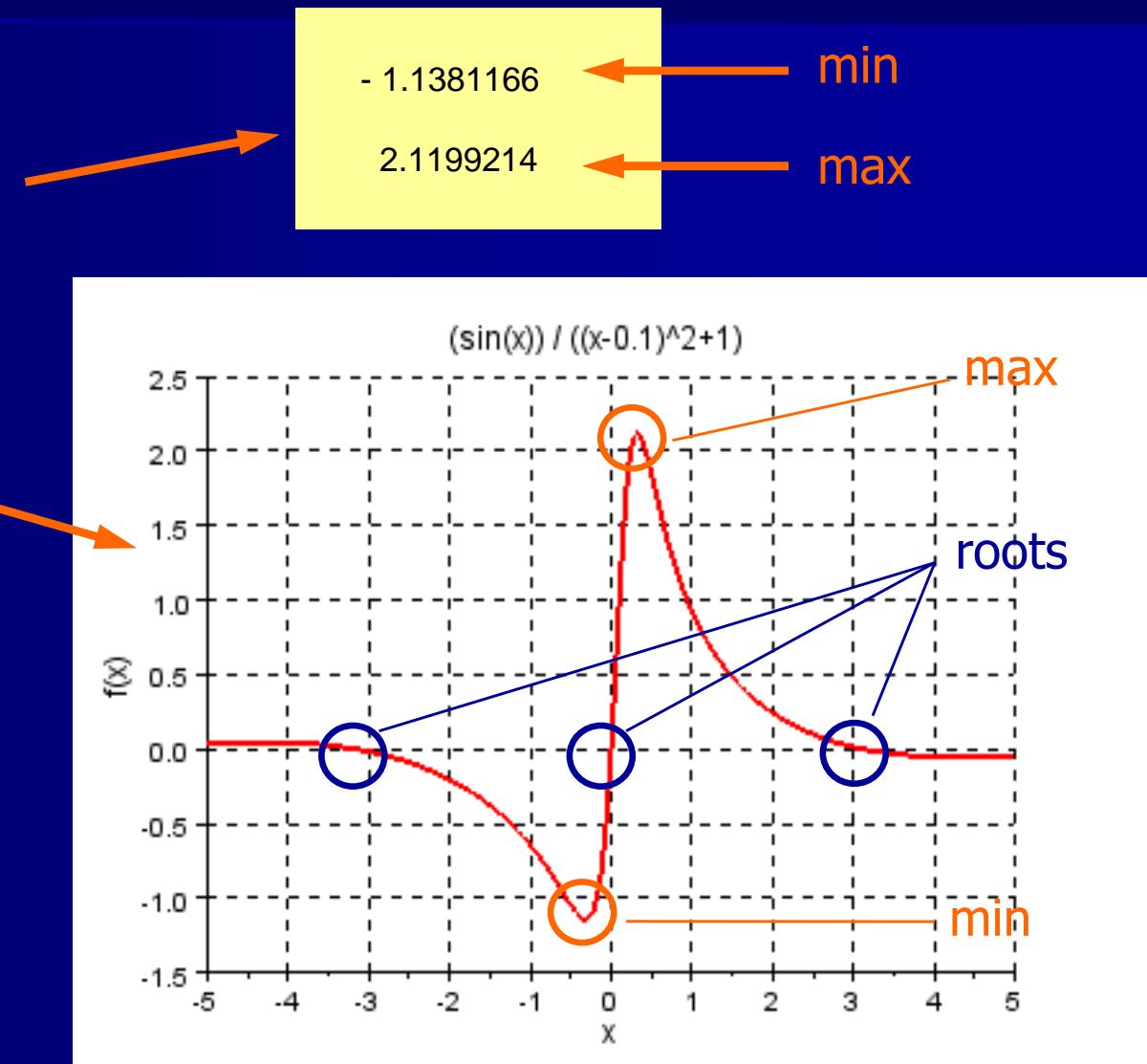
clear,clc,clf;
// SUBROUTINES
//-----
deff(['fun1,grad,ind']=cost1(x,ind)',...           // Function
      'fun1=sin(x)/((x-0.1)^2+0.1),grad=0');;
deff(['fun2,grad,ind']=cost2(x,ind)',...           // Inverse function,
      'fun2=-sin(x)/((x-0.1)^2+0.1),grad=0');; // note minus sign

// ---- MAIN ---- //
// Plot function:
//-----
x=-5:0.01:5;
fplot2d(x,cost1,5)                                // Plot function
xgrid

// Display min & max by calling subroutines:
//-----
disp(optim(list(NDcost,cost1,0),0))          // Display y min
disp(-optim(list(NDcost,cost2,0),0))          // Display y max
// ---- END OF MAIN ---- //
```

fsolve() and optim(): demo (3/4)

- Here are the minimum and maximum y values produced by optim()
- And here is the plot. It is clear that it has three roots
- The next task is to locate the roots. For that we must provide approximate solutions (e.g. -3,0,3 in this case), based on which Scilab computes an exact solution for the given neighborhood



fsolve() and optim(): demo (4/4), solving the roots

- As said on the previous slide, approximate values for the roots are:
 $x_1 \approx -3$, $x_2 \approx 0$, $x_3 \approx 3$
- With the script is loaded into Scilab, we find the solutions on the Console using the command $x = \text{fsolve}(x_0, f)$:

```
-->x1 = fsolve(-3,cost1)
x1 =
- 3.1415927

-->x2 = fsolve(0,cost1)
x2 =
0.

-->x3 = fsolve(3,cost1)
x3 =
3.1415927
```

x1

x2

x3

Equation systems require a different approach. See e.g. Zogg, pp. 66-69

I said above that the Help Browser is confusing when one tries to find out something about optim(). A better source is Section 4.2 in Campbell et al.

fsolve(): limitation

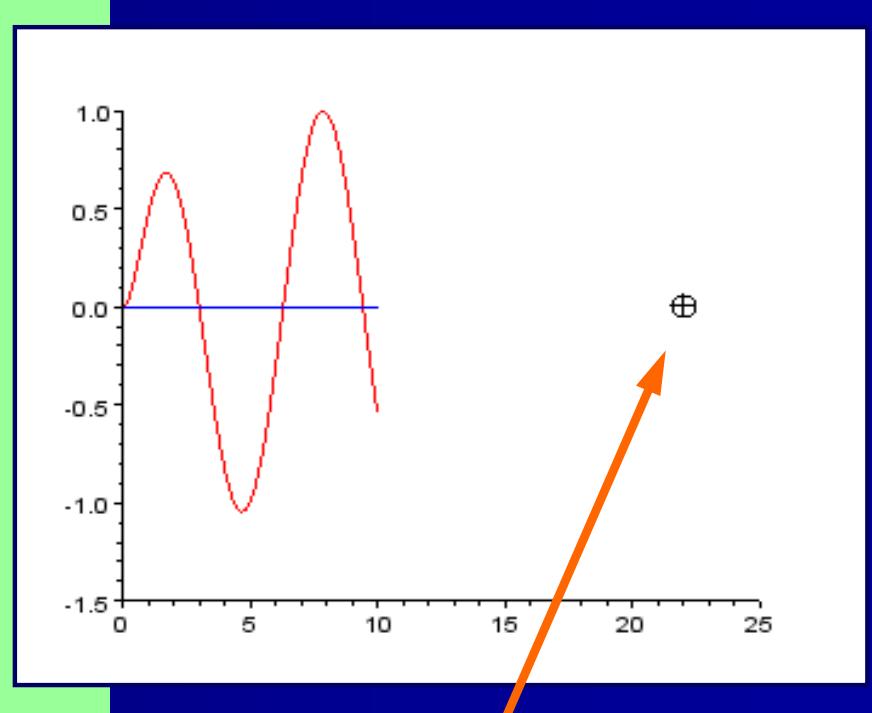
The script below demonstrates that for values of point close to peak of the sin curve, e.g. 4.6 or 8, Scilab cannot solve the root correctly

```
// fsolve.sce

// Solves, for the equation sin(a*x)-x*exp(-x),    /
// the root closest to a defined point.                /
// Note: The selected point must not be too           /
// close to the midpoint between two roots            /

clear,clc,clf;
function y=myfunc(x)
  a=1;
  y=sin(a*x)-x.*exp(-x);
endfunction

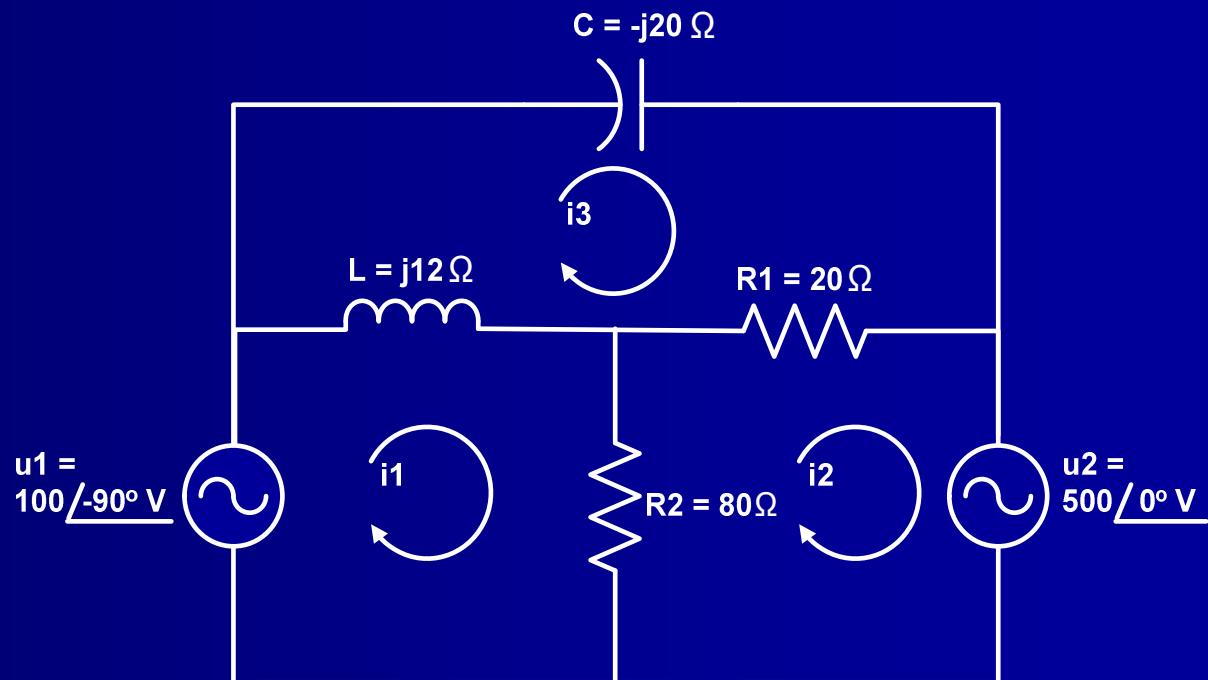
x1=linspace(0,10,300);
plot2d(x1,myfunc(x1),5)      // Plot function
plot2d(x1,zeros(x1),2)       // Add y=0 graph
point = 8;                    // Point of interest
[x,y]=fsolve(point,myfunc)    // Def root closest to point
plot2d(x,y,-3)               // Add mark for root location
```



Root mark in wrong place

Complex numbers: demo, task

- Complex numbers have not been discussed at any length before, so let's look at a practical problem
- The task is to solve the steady-state currents i_1 , i_2 , and i_3 in the shown circuit
- Recall Example 2-2 and write down the **impedance matrix Z** by inspection



$$\rightarrow [Z] = \begin{bmatrix} R_2 + jL & -R_2 & -jL \\ -R_2 & R_1 + R_2 & -R_1 \\ -jL & -R_1 & R_1 + jL - jC \end{bmatrix}$$

Complex numbers: demo, equations

- By plugging in numeric values we get the following state-space equation $[I]=[Z]^{-1}[u]$. Scilab does not have a function for shifting between polar and rectangular coordinates, so we recalculate the voltages manually (a rectangular-to-polar conversion routine is included in the script), which is simple in this case:

$$\begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 80+j12 & -80 & -j12 \\ -80 & 100 & -20 \\ -j12 & -20 & 20-j8 \end{bmatrix}^{-1} \begin{bmatrix} 0-j100 \\ -500-j0 \\ 0+j0 \end{bmatrix}$$

- Note that u_2 was selected opposite to u_1 , hence the minus sign
- Scilab has no problems with doing inverse matrices but, as mentioned before, left hand division (\) typically gives better accuracy

Complex numbers: demo, script (1/3)

- The initial step is as Ex #5; the residual check is at the end of the script
- Now we have to transform rectangular data to polar data
- The for ⋯ end loop is run through three times, once for each current ($i_1 \dots i_3$)
- Computing the magnitude is straightforward

```
// circuit3.sce  
  
// Complex mesh-current solution. The complex results are /  
// converted from rectangular to polar values by computing /  
// their magnitude and phase. The clean() function is used /  
// to eliminate computing errors around zero. /  
  
clear,clc;  
  
// Compute complex currents:  
//-----  
Z = [80+12*%i, -80, -12*%i;  
      -80, 100, -20;  
      -12*%i, -20, 20-8*%i]; // Impedance matrix  
u = [-100*%i; -500; 0]; // Voltage matrix  
i_n = Z\u; // Compute  $i = Z \backslash u$   
  
// Calculate magnitude and phase:  
//-----  
magn_i = []; // Define empty current matrix  
phase_i = []; // Define empty phase matrix  
for j = 1:1:3 // Compute for three currents  
    magn_i(j) = sqrt(real(i_n(j))^2 + imag(i_n(j))^2);  
    // Computes magnitude
```

Complex numbers: demo, script (2/3)

- This is where one has to be careful and consider all alternatives
- Note that the zero (0) condition gets a margin for computing errors through the clean() function
- Each time the for ... end loop is run through, the matrix result() collects the data

```
// Calculate phase:  
//-----  
if clean(real(i_n(j))) > 0 then      // In 1st or 4th quadrant  
    phase_i(j) = atan(imag(i_n(j))/real(i_n(j)))*(180/%pi);  
elseif clean(real(i_n(j))) < 0        // In 2nd or 3rd quadrant  
    if clean(imag(i_n(j))) > 0 then    // In 2nd quadrant  
        phase_i(j) = atan(imag(i_n(j))/real(i_n(j)))*(180/%pi) + 180;  
    elseif clean(imag(i_n(j))) < 0 then // In 3rd quadrant  
        phase_i(j) = atan(imag(i_n(j))/real(i_n(j)))*(180/%pi) - 180;  
    else                                // On negative Re-axis  
        phase_i(j) = 180;  
    end  
elseif clean(imag(i_n(j))) > 0        // On positive Im-axis  
    phase_i(j) = 90;  
elseif clean(imag(i_n(j))) < 0        // On negative Im-axis  
    phase_i(j) = -90;  
else                                    // Origin: imag(i_n(j)) = real(i_n(j)) = 0  
    phase_i(j) = 0;  
end  
result(j,:) = [i_n(j), magn_i(j), phase_i(j)];  
// Matrix collects computed data  
j = j+1;  
end
```

Complex numbers: demo, script (3/3) & print

- The result is displayed with the `disp()` command with everything included in the argument vector
- Finally, the preliminary result is checked as before
- And the answer on the Console:

In plain English:

$$\begin{aligned}i_1 &= 22.0 \cos(\omega t - 129.5^\circ) \text{ A} \\i_2 &= 24.0 \cos(\omega t - 129.3^\circ) \text{ A} \\i_3 &= 25.5 \cos(\omega t - 78.7^\circ) \text{ A}\end{aligned}$$

// Display summary:

```
//-----
currents = ['i1 = ', 'i2 = ', 'i3 = '];           // String matrix
statement = [' equals: '' equals: '' equals: '']; // String matrix
disp(['CURRENTS IN COMPLEX AND POLAR FORM:'])    // Headline
disp([currents, string(result(:,1)), statement,... // Display result
      string(result(1:3,2)), string(result(1:3,3))])
```

// Check residual:

```
//-----
residual = clean(u - Z*i_n)'                  // Check initial results
```

CURRENTS IN COMPLEX AND POLAR FORM:

```
! i1 = -14-%i*17     equals: 22.022716 -129.47246 !
! i2 = -15.2-%i*18.6 equals: 24.020824 -129.25584 !
! i3 = 5-%i*25       equals: 25.495098 -78.690068 !
```

Numeric derivation (1/3): derivative()

- The derivative of the function $f(x)$ is defined as the limit

$$f'(x) = \lim_{d \rightarrow 0} \frac{f(x + d) - f(x)}{d}$$

- We can compute the numeric value of $f'(x)$ at a point x using the function

`derivative(f(x),x,opt(d))`

where `opt(d)` is an optional step size. However, Scilab's Help Browser recommends using the **default value**

- To the right the derivative for the earlier investigated function has been computed at five different points
- `derivative()` outputs a 5×5 matrix, in which the diagonal is of interest

```
// derivative_1.sce
// Derivative of sin(x)/((x-0.1)^2+0.1) /
// calculated at selected points /
clear,clc;
funcprot(0);

deff('y=f(x)', 'y=sin(x)./((x-0.1)^2 + 0.1)');

x = [-2 -1 0 1 2]; // Points of interest
disp(["Point", "Derivative"])
disp([x, diag(derivative(f,x))])
```

!Point Derivative !

- 2.	- 0.2800316
- 1.	- 0.6663016
0.	9.0909091
1.	- 1.2353251
2.	- 0.3632083

Numeric derivation (2/3): script

- This script that plots the previous function together with its derivative
- The equation and its derivative are defined with separate deff() functions
- fplot2d() accepts the same multiple plot structure as was used earlier with plot2d()
- children(2) and children(3) are used because children(1) is reserved for legend

```
// derivative_3.sce

// Plotting f(x) = sin(x)/((x-0.1)^2+0.1)  /
// and its derivative                         /

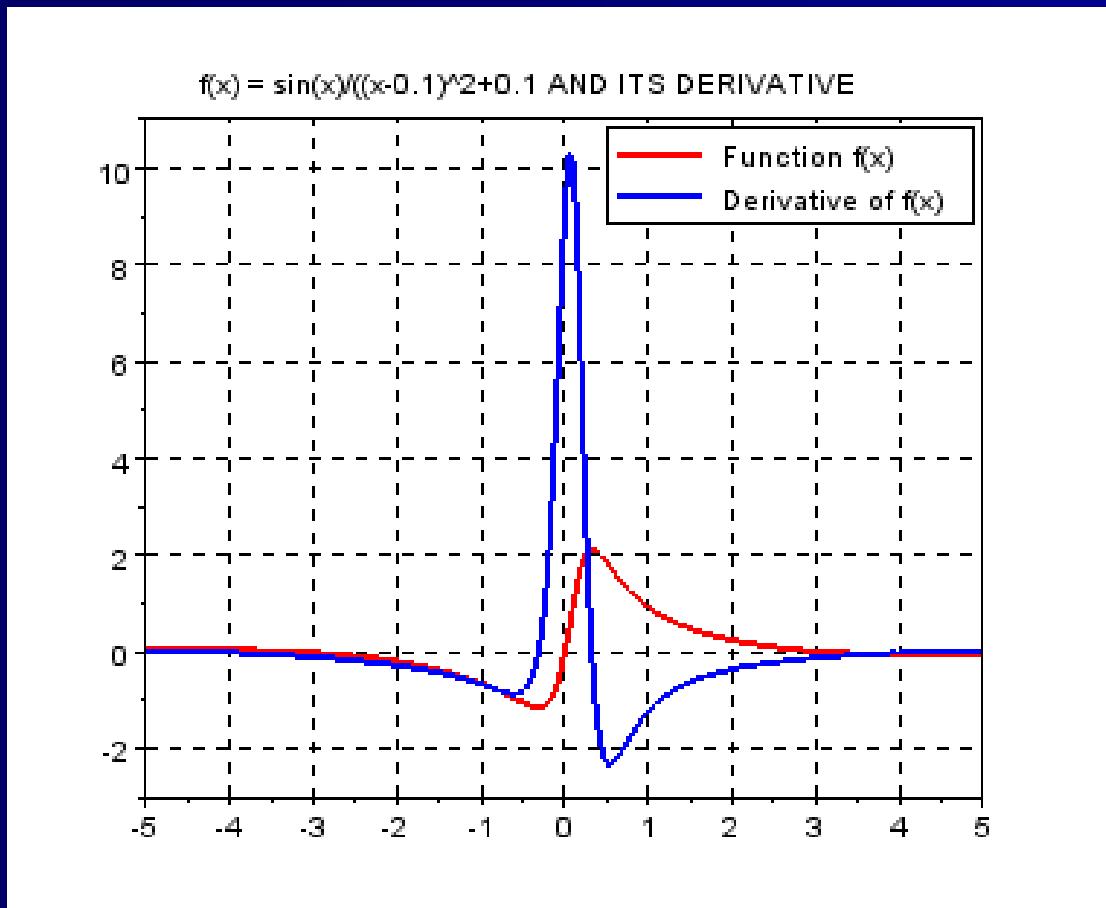
clear,clc,clf;
funcprot(0)

x = -5:0.01:5;                                // Area of interest
d = 0.001;                                     // Step size

// Define function & derivative:
// -----
deff('y1=f(x)','y1=sin(x)./((x-0.1)^2 + 0.1)'); // f(x)
deff('y2=g(x)','y2=((sin(x+d)./(((x+d)-0.1)^2 + 0.1))... // f'(x)
-(sin(x)./((x-0.1)^2 + 0.1))/d')';

// Plot function & derivative:
// -----
rect = [-5,-3,5,11];
fplot2d(x,f,5,"011"," ",rect)                // Plot function
fplot2d(x,g,2,"000")                          // Plot derivative
xgrid                                         // Add grid to plot
xtitle('f(x) = sin(x)/((x-0.1)^2+0.1 AND ITS DERIVATIVE')
legend('Function f(x)', 'Derivative of f(x)')
a=gca();
a.children.children(2).thickness=2             // f'(x) thickness
a.children.children(3).thickness=2             // f(x) thickness
```

Numeric derivation (3/3): plot



A lesson from doing this exercise is that two deff() functions in tandem, i.e. one for $f(x)$ followed by one for $f'(x)$ that utilizes $f(x)$, does not work. On the contrary, the attempt may cause Scilab to crash

Pay attention to the legend command in the script. It comes before the related handle statements, but Scilab does not complain. Beats me...

Numeric integration (1/6): definite integral

- Consider the definite integral

$$A = \int_a^b f(x) dx$$

- To solve the integral, first define the function $y = f(x)$, e.g. using the deff() function
- The integral can then be evaluated using Scilab's intg() function,* i.e.:

$$A = \text{intg}(a,b,f)$$

*) The function integrate() can be more useful in some cases.
Check with Help

```
-->deff('y=f(x)', 'y=6*x^2');
```

```
-->A = intg(-2,1,f)  
A =
```

18.

```
-->deff('y=f(x)', 'y=sin(x)');  
-->A = intg(%pi/4, 3*%pi/4, f)  
A =
```

1.4142136

Change the integration limits to 0 and $2*\pi$, and this is what you get

```
-->A=intg(0, 2*%pi, f)  
!--error 24  
Convergence problem...
```

Numeric integration (2/6): length of an arc

- The length of an arc $f(x)$, between points a and b , is given by the definite integral

$$L = \int_a^b \{1 + [f'(x)]^2\}^{1/2} dx$$

- Let's compute the length of $f(x) = x^3/24 + 2x^{-1}$ from $x=2$ to $x=3$
- The task requires manual derivation, which yields

$$f'(x) = x^2/8 - 2x^{-2}$$

```
-->deff('y=g(x)', 'y=sqrt(1+(x^2/8-2*x^(-2))^2)');
```

```
-->L=intg(2,3,g)  
L =
```

1.125

```
-->L=intg(3,4,g)  
L =
```

1.7083333

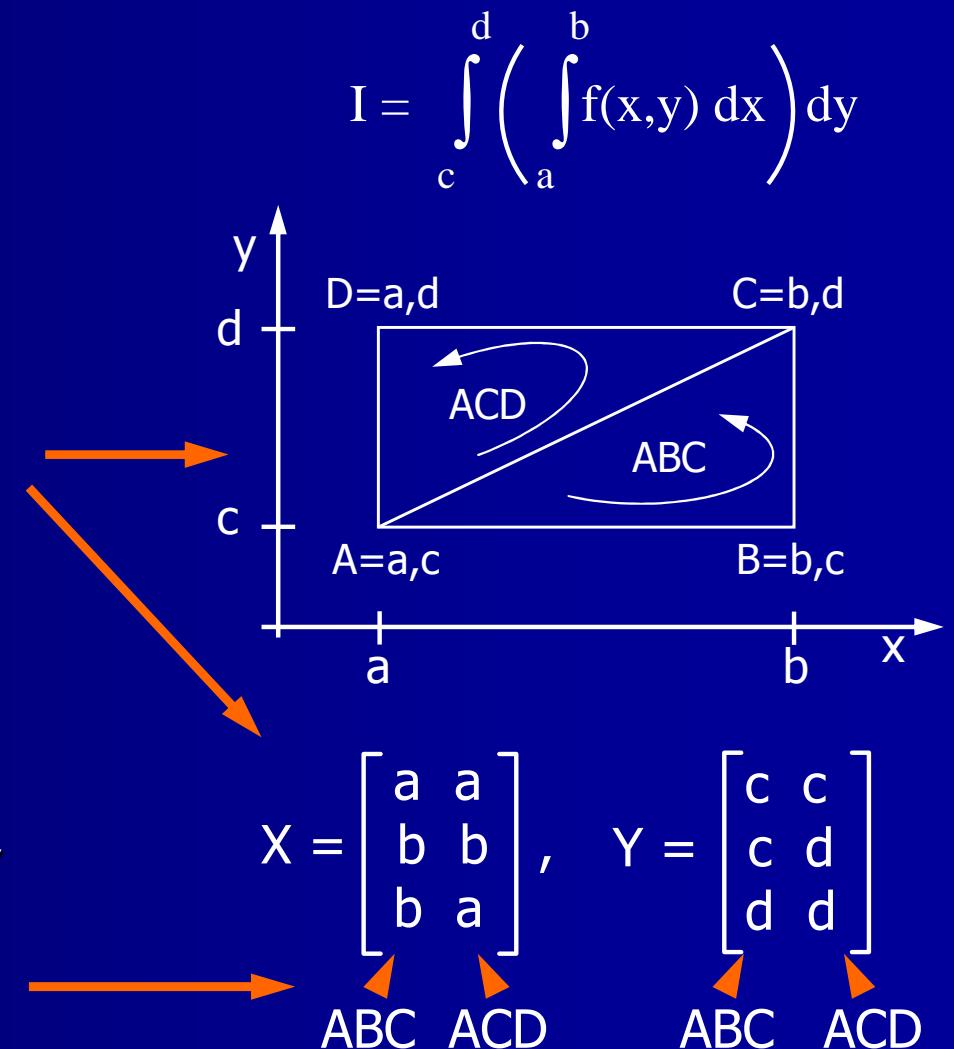
Numeric integration (3/6): double integral, principle

- The function `int2d()` computes the 2D area integral of a function $f(x,y)$ over a region consisting of N triangles
- x and y must therefore be defined through triangulation matrices X and Y , after which the command is

`[I,err] = int2d(X,Y,f),`

and Scilab returns the integration variable I and an estimation of the error, err (not mandatory)

- The triangles are ABC and ACD , as shown in the picture. Triangle elements are inserted column-wise in the matrices



Numeric integration (4/6): double integral, demo

- Let's compute the double integral

$$I = \int_{\pi/2}^{2\pi} \left(\int_0^{\pi} (y \cos(x) + x \sin(y)) dx \right) dy$$

- By looking at the integration limits of the function we find the triangulation matrices X and Y:

$$X = \begin{bmatrix} 0 & 0 \\ \pi & \pi \\ \pi & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} \pi/2 & \pi/2 \\ \pi/2 & 2\pi \\ 2\pi & 2\pi \end{bmatrix}$$

```
-->def('z=f(x,y)', 'z=y*cos(x)+x*sin(y)');

-->X = [0 %pi %pi; 0 %pi 0]'
```

X =

0.	0.
3.1415927	3.1415927
3.1415927	0.

```
-->Y = [%pi/2 %pi/2 2*%pi; %pi/2 2*%pi 2*%pi]'
```

Y =

1.5707963	1.5707963
1.5707963	6.2831853
6.2831853	6.2831853

```
-->[I,err] = int2d(X,Y,f)
```

err =

9.805D-11

I =

- 4.9348022

Numeric integration (5/6): double integral, plot

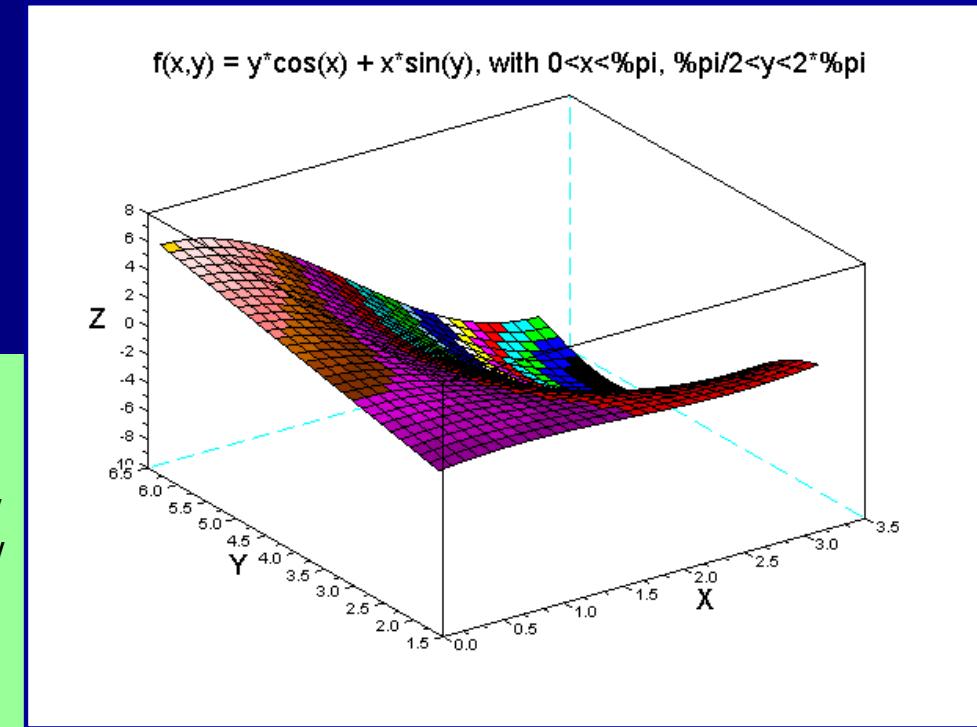
The plot of $f(x,y) = y\cos(x) + x\sin(y)$ is here done with a separate script:

```
// double_integral_plot.sce

// Plot the function z = y*sin(x) + x*sin(y)      /
// over the rectangle 0<x<%pi, %pi/2<y<2*%pi   /

clear,clc,clf;

x=linspace(0,%pi,30);          // Linear x axis
y=linspace(%pi/2,2*%pi,30);    // Ditto y axis
[X,Y]=meshgrid(x,y);          // Surface mesh
Z=(Y.*cos(X)+X.*sin(Y));      // 3D surface equation
surf(X,Y,Z)                   // Plot 3D surface
xtitle('f(x,y) = y*cos(x) + x*sin(y),...      // Add title
       with 0<x<%pi, %pi/2<y<2*%pi')
```

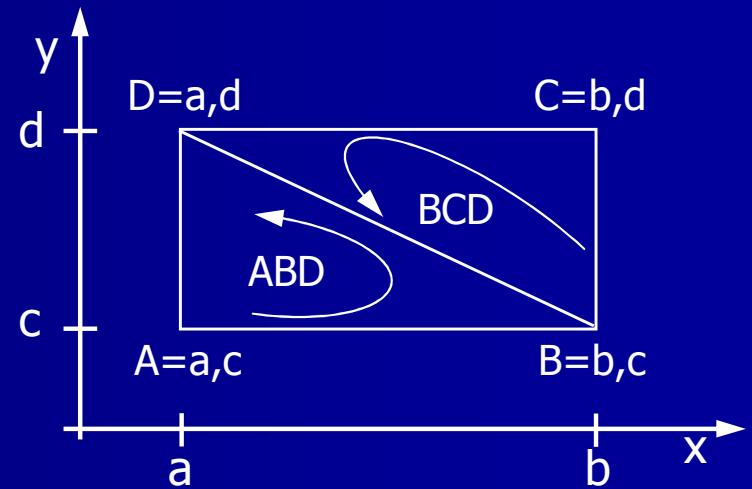


The figure has been edited
with the Figure Editor

Numeric integration (6/6): double integral, check

- We can check the computed result using **the other possible triangulation**
- Now we get the triangulation matrices shown here
- Plugging these matrices into the Console gives the following result:

```
-->X=[0 %pi 0; %pi %pi 0]';  
  
-->Y=[%pi/2 %pi/2 2*%pi; %pi/2 2*%pi 2*%pi]';  
  
-->[I,err]=int2d(X,Y,f)  
err =  
  
9.887D-11  
I =  
  
- 4.9348022
```



$$X = \begin{bmatrix} a & b \\ b & b \\ a & a \end{bmatrix}, \quad Y = \begin{bmatrix} c & c \\ c & d \\ d & d \end{bmatrix}$$

Same result, but a small difference in the estimated error

Ordinary differential equations (ODEs): `ode()`*

- This simplest call for solving ODEs is `ode()` that has the general form:

$$y = \text{ode}(y0, t0, t, f(t, y))$$

where

- $y0$ = initial condition (normally a column vector)
- $t0$ = initial time (normally 0)
- t = vector of instances for which the solution has to be computed, e.g. $t = [0:0.01:10]$
- $f(t, y)$ = function for which the solution has to be found, often stated as $[ydot] = f(t, y)$. Here t is a scalar, y a column vector, and $[ydot]$ a column vector with values of the derivative
- `ode()` can also have optional arguments. See Help for details

*) Sallet, G.: *Ordinary Differential Equations with Scilab*, <http://www.math.univ-metz.fr/~sallet/ODE_Scilab.pdf> is an “old” but good text.

First-order ODEs: demo

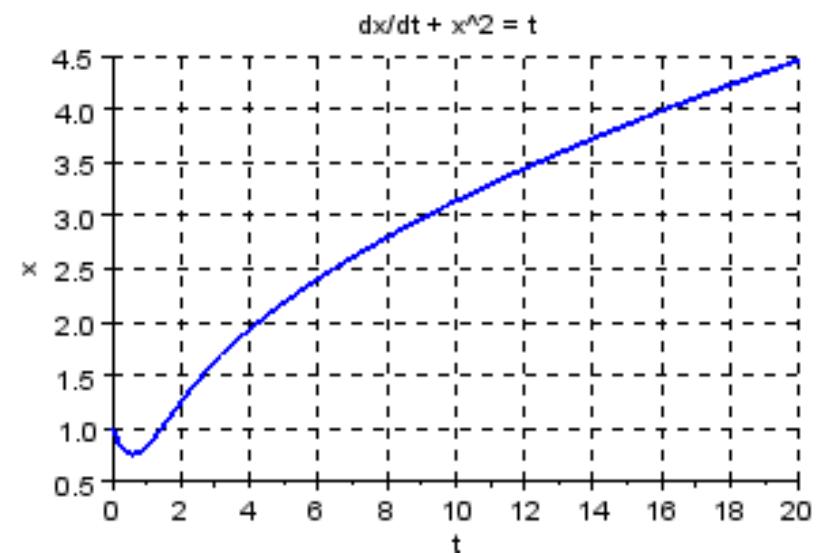
- Let's find the solution for the first-order homogenous ODE

$$x' + x^2 = t,$$

with the initial condition $x(0) = 1$. Plot the solution for $t \in [0,20]$

- Start by rewriting the function as $x' = -x^2 + t$
- Note how x' is designated y in the `deff()` argument

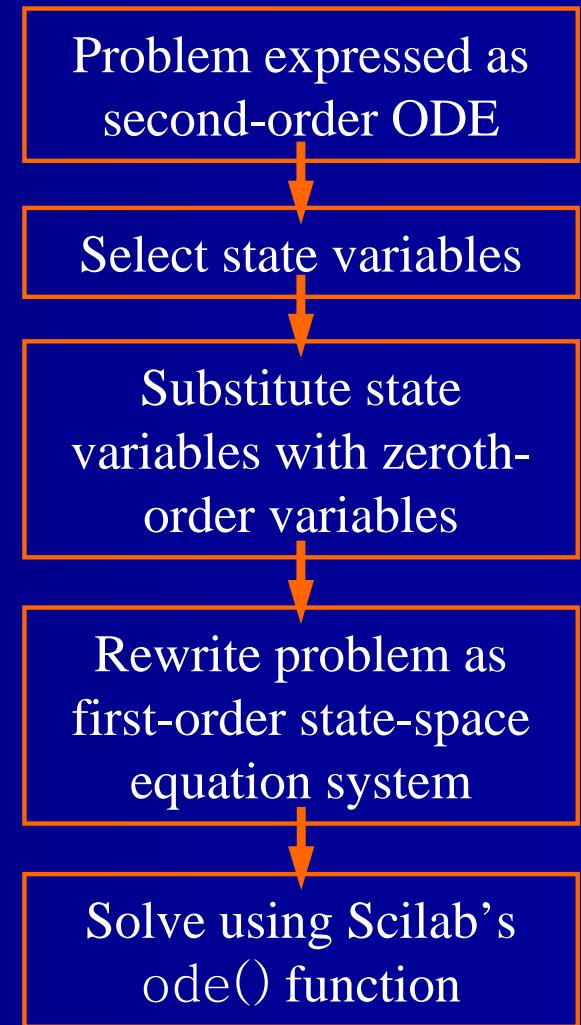
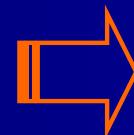
```
// first-order_ODE.sce
// Solve the equation x'+x^2 = t /
// for x(0) = 0
clear,clc,clf;
funcprot(0)
deff('y=f(t,x)','y=-x^2+t') // Define function
t=linspace(0,20); // Abscissa
x=ode(1,0,t,f); // Compute equation
plot2d(t,x,style=5) // Plot
xtitle('dx/dt + x^2 = t','t','x')
xgrid a=gca();
a.children.children.thickness=2
```



In this case Scilab does not accept numeric arguments of children

Second-order ODEs: introduction

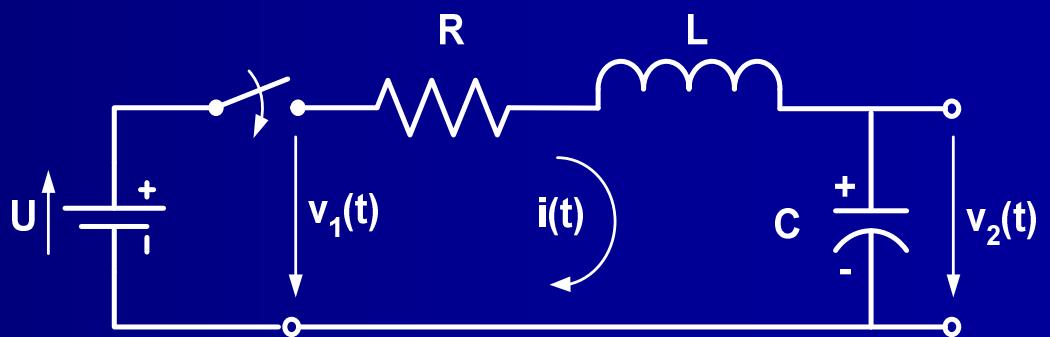
- Scilab only supports first-order differential equations—as do other programs for numeric computing
- Higher order problems must be reduced to first-order systems, i.e. by shifting to state-space representation
- The methodology runs according to the algorithm shown to the right
- A good treatment of state-space methods is e.g. Chapter 8 in Burns, R.S.: *Advanced Control Engineering*, Butterworth-Heinemann, 2001



Second-order ODEs: RLC circuit (1/5), the task

- The task is to plot the output voltage v_2 for the shown RLC circuit, when

- $U = 5V$
- switch closes at $t = 1$
- $R = 0.3 \Omega$
- $L = 0.5 H$
- $C = 0.8 F$



- We can derive the following second-order ODE for the circuit:

$$LC \frac{d^2v_2(t)}{dt^2} + RC \frac{dv_2(t)}{dt} + v_2(t) = v_1(t)$$

Second-order ODEs: RLC circuit (2/5), reduce

- Simplify the equation for clarity:

$$LCv_2'' + RCv_2' + v_2 = v_1$$

- Select v_2 and its derivative v_2' as state variables, and substitute:

$$x_1 = v_2 \text{ and } x_2 = v_2' (= x_1')$$

- With v_1 substituted by u , the first-order ODE system becomes:

$$\begin{cases} x_1' = + 0 \cdot x_1 + 1 \cdot x_2 + 0 \cdot u & (\text{simpler: } x_1' = x_2) \\ x_2' = - \frac{1}{RC} x_1 - \frac{R}{L} x_2 + \frac{1}{LC} u \end{cases}$$

- Which gives the state-space expression that we are looking for:

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{RC} & -\frac{R}{L} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{LC} \end{bmatrix} u$$

Second-order ODEs: RLC circuit (3/5), script

Recall the discussion
in connection with Ex
2-3: We are working
with a matrix
expression of the type

$$\mathbf{x}' = \mathbf{A}\mathbf{x} + \mathbf{b}\mathbf{u}$$

All of these factors
can be seen here,
with \mathbf{x}' being denoted
as \mathbf{ss} and \mathbf{x} substituted
by \mathbf{y}



```
// RLC_ODE.sce
// Simulation of a series RCL circuit with    /
// 5V step input voltage at t = 1s           /
clear,clc,clf;
//-----
// Define circuit components:
//-----
R = 0.3;      // Resistance (Ohm, V/A)
L = 0.5;      // Inductance (Henry, Vs/A)
C = 0.8;      // Capacitance (Farad, As)
//-----
// Define space-state equations & input signal:
//-----
A = [0 1; -1/(L*C) -R/L];          // System matrix
B = [0; 1/(L*C)];                  // Input matrix
deff(['ut']=u(t),'ut=2.5*(1+sign(t-1))'); // Step input signal
deff(['ss']=RLC(t,y),'ss=A*y+B*u(t)'); // Space-state expression
```

Second-order ODEs: RLC circuit (4/5), script

The `ode()` function computes our differential equation by using the RLC state-space expression of the second `deff()` function. Calling parameters are `y0` and `t0`

Note the plot command (new way of doing `plot2d()`)

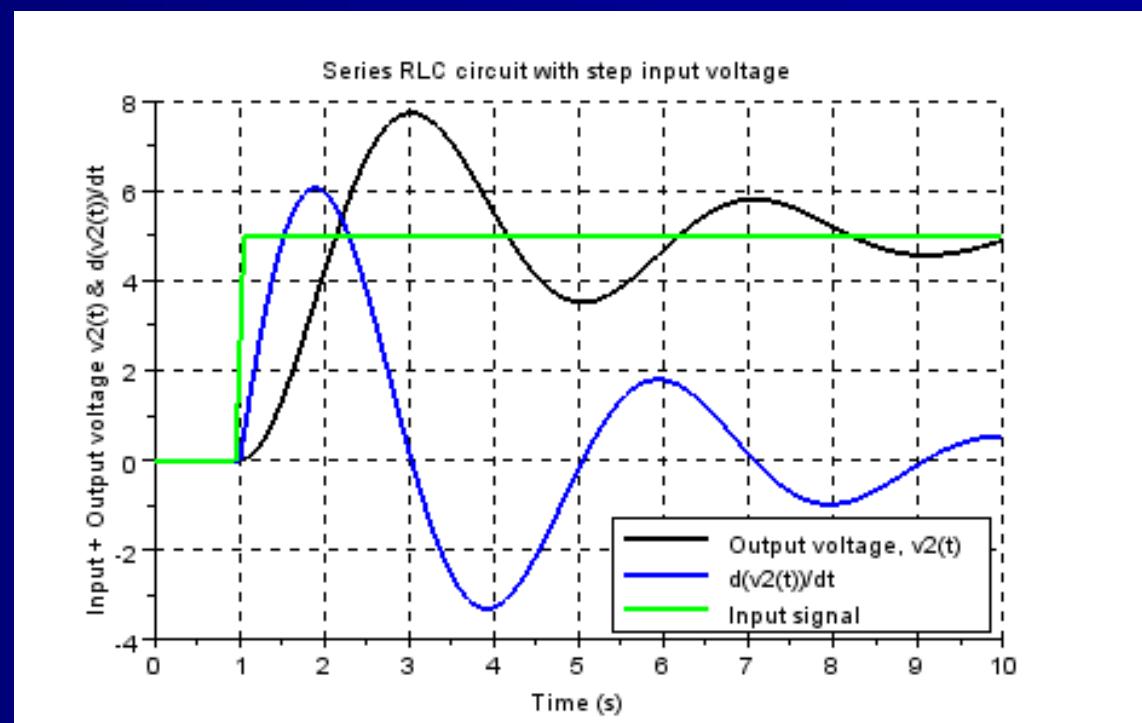
Handle commands come before the legend (in this case Scilab gives an error message if you try it the other way)

```
// Compute using ode(), which calls previous deff() function:  
//-----  
out0 = [0;0];           // Initial output voltage &  $d(v2)/dt = 0$   
t0 = 0;                // Initial time = 0  
Time = [0:0.05:10];    // Time as abscissa  
State = ode(out0,t0,Time,RLC); // State variable vector ( $v2'$ , $v2$ )  
  
// Plot and add title & grid:  
//-----  
plot2d(Time,[State',u(Time')]); // Note transposed arguments!  
xtitle('Series RLC circuit with step input voltage',...  
      'Time (s)', 'Input + Output voltage  $v2(t)$  &  $d(v2(t))/dt$ ')  
xgrid  
  
// Edit plot:  
//-----  
a=gca();  
a.children.children.thickness=2 // Make all graphs thicker  
  
// Add legend (must come after handle commands):  
//-----  
legend('Output voltage,  $v2(t)$ ', ' $d(v2(t))/dt$ ', 'Input signal', 4)
```

Second-order ODEs: RLC circuit (5/5), plot

The plot shows that the circuit is **undercritically damped**. Change the resistor value to 1.5Ω , and it becomes **critical**. It is **overcritical** for still higher values of R

Handle commands could be used to edit the figure further. I did not do it because the main point with this demo is to solve a second-order ODE



odeoptions()

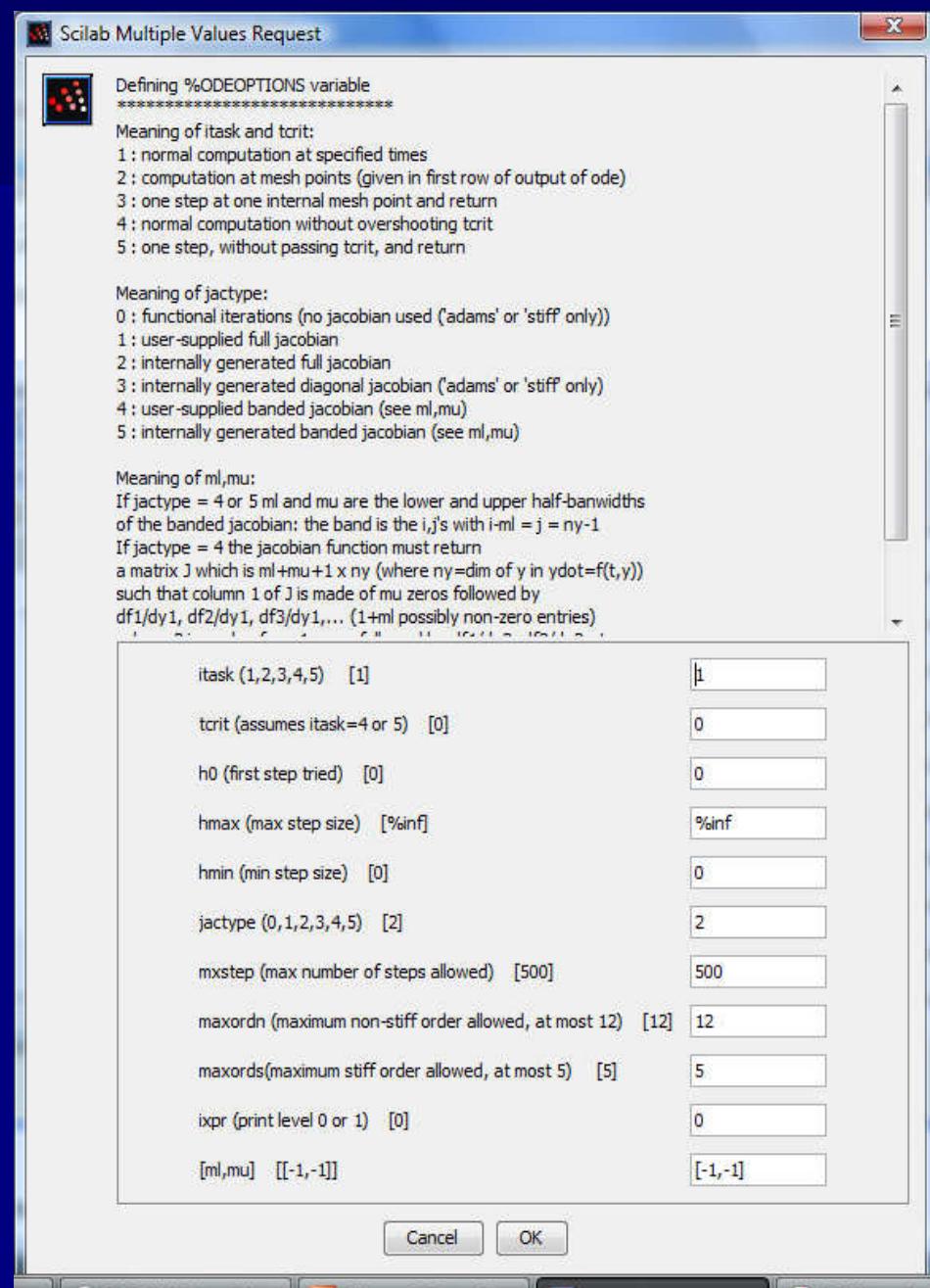
The command

`%ODEOPTIONS = odeoptions()`

opens the GUI shown right. With the help of it you can change parameters for solving differential equations. Examples:

- **h0** = size of first step
- **hmax** = maximum step size
- **hmin** = minimum step size
- **mxstep** = minimum # of steps

Check with Help for details



14. Examples, Set 5

The examples give additional insight into working with math on Scilab



[Return to Contents](#)

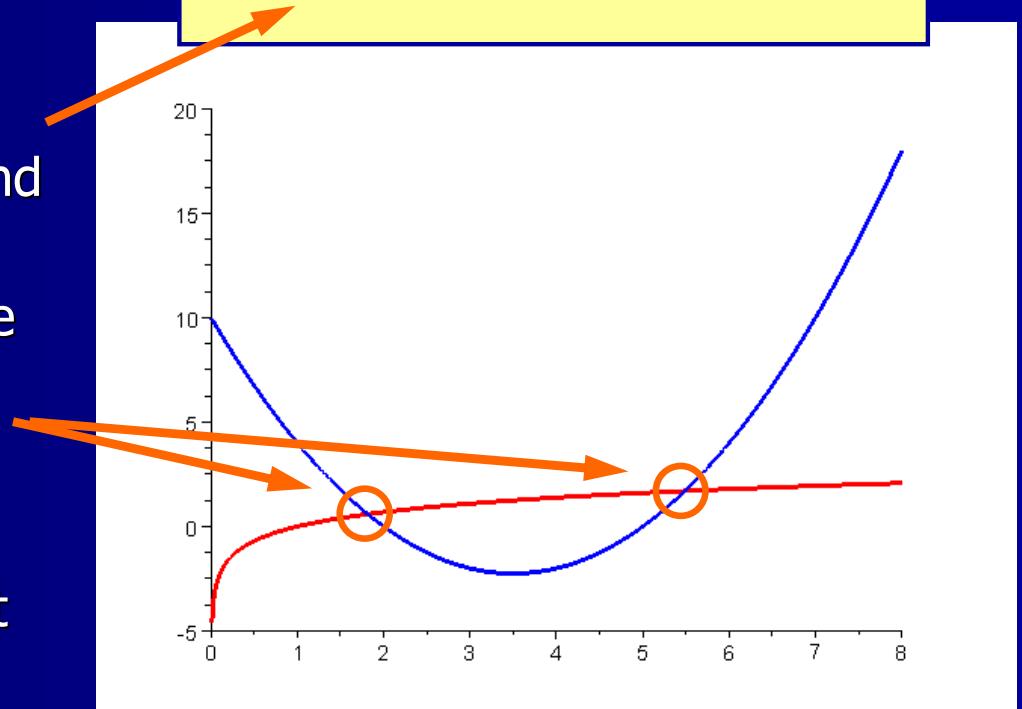
Example 5-1: solving an equation (1/3)

- This demo is based on Mäkelä
- Let's solve the equation

$$\ln(x) = x^2 - 7x + 10$$

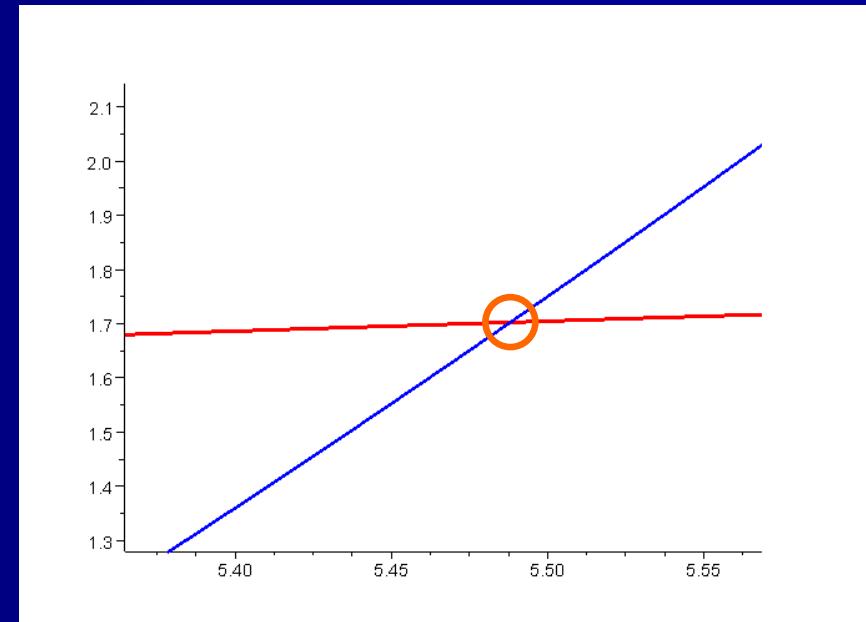
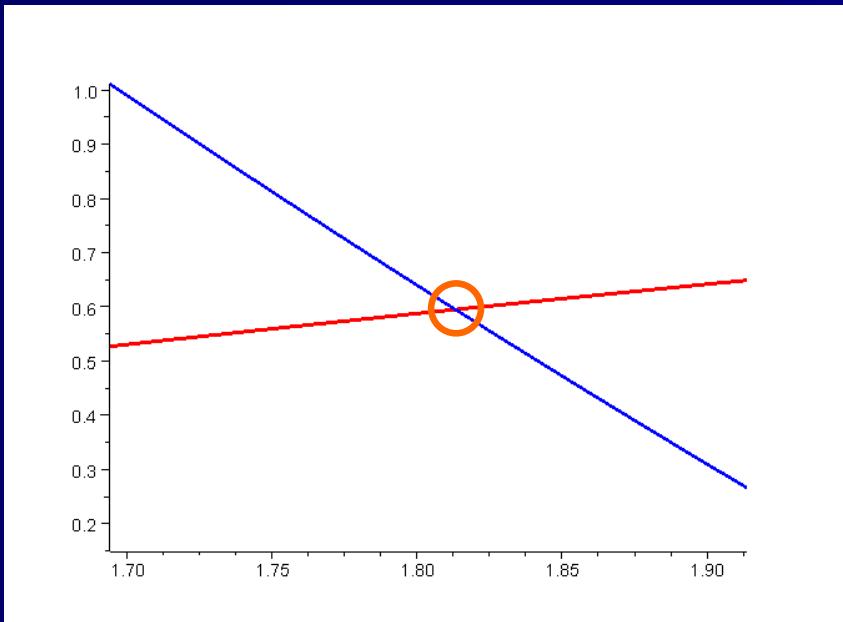
- We begin by plotting it (note how the multiple plot command is constructed)
- The plot reveals that there are two solutions, at $x_1 \approx 2$ and $x_2 \approx 5.5$
- You can see the roots more exactly by using the Graphics Window's **zoom function** (next slide)

```
-->x = (0.01:0.01:8)';  
  
-->plot2d(x,[log(x), x.^2-7*x+10])
```



Ex 5-1: solving an equation (2/3)

- The zoom function gives more precise values for the roots:
 $x_1 = 1.81$ and $x_2 = 5.49$
- To improve the accuracy even more we can calculate the roots with the `fsolve()` function (next slide)



Ex 5-1: solving an equation (3/3)

- `fsolve()` delivers the ultimate answer
- We can also check the error of the result. As shown, it is close to zero
- **Lessons learned:** Precise zoom in the Graphics Window produces satisfactory accuracy for most practical engineering purposes (two decimals), considering that an old engineering adage says that factors that influences the result by less than 10% can be forgotten

```
-->deff('y=f(x)', 'y=log(x)-(x^2-7*x+10)');

-->x1=fsolve(1.8,f)
x1 =
1.8132512

-->x2=fsolve(5.5,f)
x2 =
5.4881107

-->f(x1),f(x2)
ans =
- 7.772D-16
ans =
- 4.441D-16
```

Check

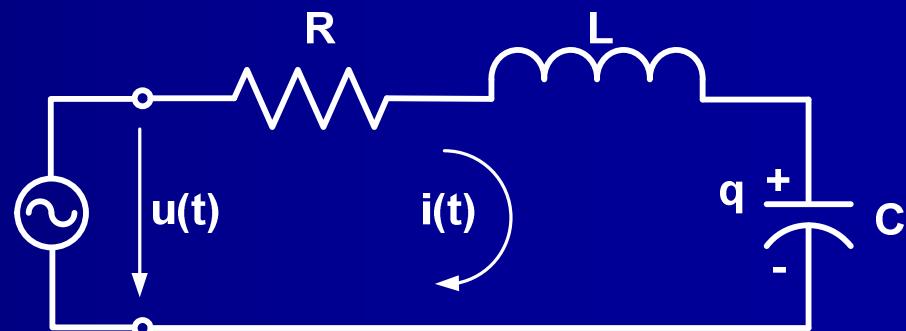
Exxample 5-2: ODE, series RLC circuit (1/5)

- This example is a modification of the earlier RLC circuit and its second-order ODE
- However, we now want to define the current $i(t)$ and charge $q(t)$ for a sinusoidal input signal and initial conditions $i(0) = 0$ and $q(0) = 0$
- Kirchoff's second law gives:

$$L \frac{di(t)}{dt} + Ri(t) + \frac{1}{C} q(t) = u(t)$$

where

$$q = \int_0^t i(t) dt \quad \text{or:} \quad \frac{dq}{dt} = i$$



$$R = 0.3 \Omega$$

$$L = 0.5 \text{ H}$$

$$C = 0.8 \text{ F}$$

$$u(t) = \sin(5t)$$

Ex 5-2: ODE, series RLC circuit (2/5)

- No substitutions are required in this case since q and its derivative i are state variables. The first-order equation system is therefore:

$$\begin{cases} q' = i \\ i' = -\frac{1}{LC}q - \frac{R}{L}i + \frac{1}{L}u \end{cases}$$

- Which gives the following state-space expression:

$$\begin{bmatrix} q' \\ i' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{LC} & -\frac{R}{L} \end{bmatrix} \begin{bmatrix} q \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} u$$

Remember: $x' = Ax + Bu$

Ex 5-2: ODE, series RLC circuit (3/5), script

There is nothing new here compared to the previous RLC/second-order ODE

```
// series_RLC_ODE.sce

// Simulation of the current i(t) and charge q(t) in      /
// a series RCL circuit with sinusoidal input voltage    /
// and initial conditions i(0)=0, q(0)=0.                  /
// Legend: ss = state-space                                /

clear;clc,clf;

// Define circuit components:
//-----
R = 0.3;          // Resistance (Ohm)
L = 0.5;          // Inductance (Henry)
C = 0.8;          // Capacitance (Farad)

// Define state-space equations & input signal:
//-----
A = [0 1; -1/(L*C) -R/L];           // SS system matrix
B = [0; 1/L];                       // SS input matrix
deff(['ut']=u(t),'ut=sin(5*t)');     // Sinusoidal input
deff(['ss']=RLC(t,y),'ss=A*y+B*u(t)'); // SS expression
```

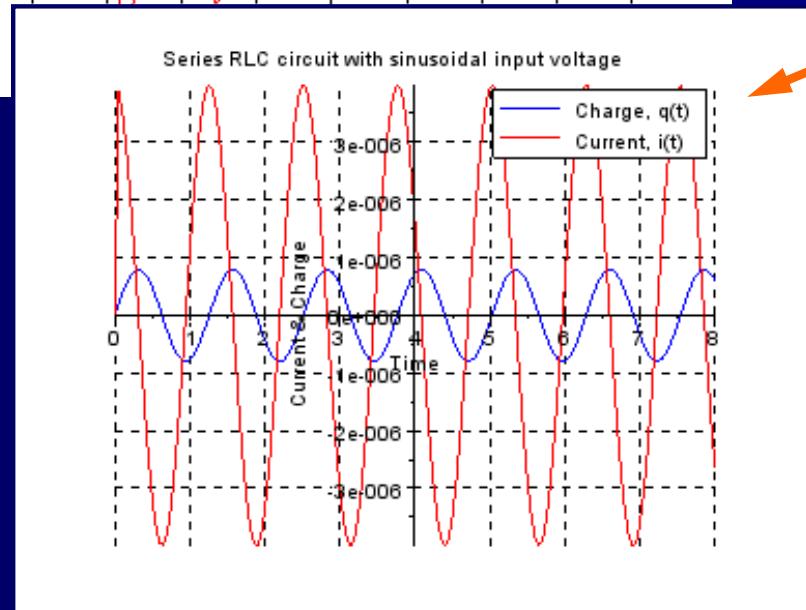
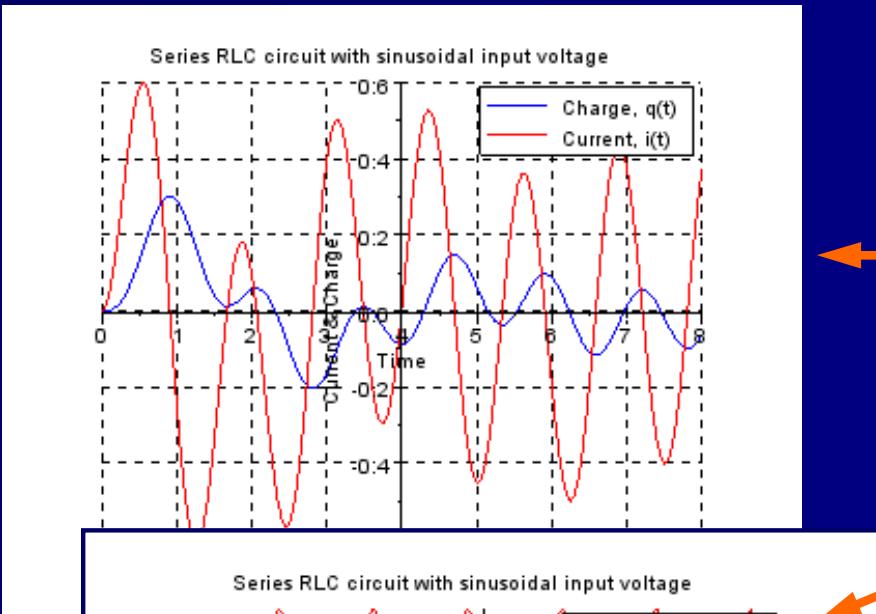
Ex 5-2: ODE, series RLC circuit (4/5), script

The `ode()` is the same as in the previous RLC case

Check the plot argument '024' and its effect on the plot (next slide)

```
// Compute using ode(), which calls the previous deff() function:  
//-----  
y0 = [0;0]; // Initial current & charge = 0  
t0 = 0; // Initial time = 0  
Time = [0:0.05:8]; // Time as abscissa  
Y = ode(y0,t0,Time,RLC); // Y = state variable vector (i,q)  
  
// Plot current & charge:  
//-----  
plot2d(Time,Y',[2 5],'024'); // Plot state vectors, note transposed Y  
xtitle('Series RLC circuit with sinusoidal input voltage',...  
'Time','Current & Charge')  
xgrid  
legend('Charge, q(t)','Current, i(t)')
```

Ex 5-2: ODE, series RLC circuit (5/5), plot



- This is the plot for the shown component values. There are initial fluctuations before the situation begins to stabilize
- This is the plot for more realistic component values of $R = 3 \text{ k}\Omega$, $L = 0.5 \mu\text{H}$, and $C = 0.8 \mu\text{F}$
- There used to be problems with the latter case (Scilab 5.1.1), but these have obviously been solved

Example 5-3: System of first-order ODEs

- This example is modified from Povy (pp. 66-67, Povy also has an animation version on pp. 67-68, but it causes Scilab to **crash**). The example finishes with an interesting `plot2d()` command
- The task is to plot the **slope (vector) field** for the following system of first-order ODEs :

$$\begin{cases} x' = y \\ y' = -x - y \end{cases} \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

together with a single **phase portrait** with the initial trajectory $x(0) = 1$ and $y(0) = 1$

- The script can utilize either the ODE system (as Povy has done) or the state-space representation. We'll select the latter, in line with earlier examples

Ex 5-3: script

The state-space function
is named `firstorder()`

The vector field is drawn
with `fchamp()`

`ode()` has only one
argument (and accepts
only one name) for the
initial condition → `x` and
`y` are renamed `x(1)` and
`x(2)` respectively, as
shown in the arguments
for `plot2d()`

```
// ode_phase_plane_m.sce
// The scripts plot the phase plane of the      /
// equation system x'=y, y'=-x-y together with  /
// a single phase portrait that satisfies the    /
// initial condition x(0)=1, y(0)=1              /
/
clear,clc,clf;
funcprot(0);

// First order transformation:
//-----
A = [0 1;-1 -1];           // State vector
deff(['ss']=firstorder(t,x)',ss=A*x');

// Create & draw slope (vector) field:
//-----
z = linspace(-1.5,1.5,10);
fchamp(firstorder,0,z,z)   // Draw vector field

// Create phase portrait:
//-----
x0 = [1;1];                 // Initial condition
t = linspace(0,30,300);
[x] = ode(x0,0,t,firstorder); // [x]=state variable vector
                                // with x=x(1), y=x(2)

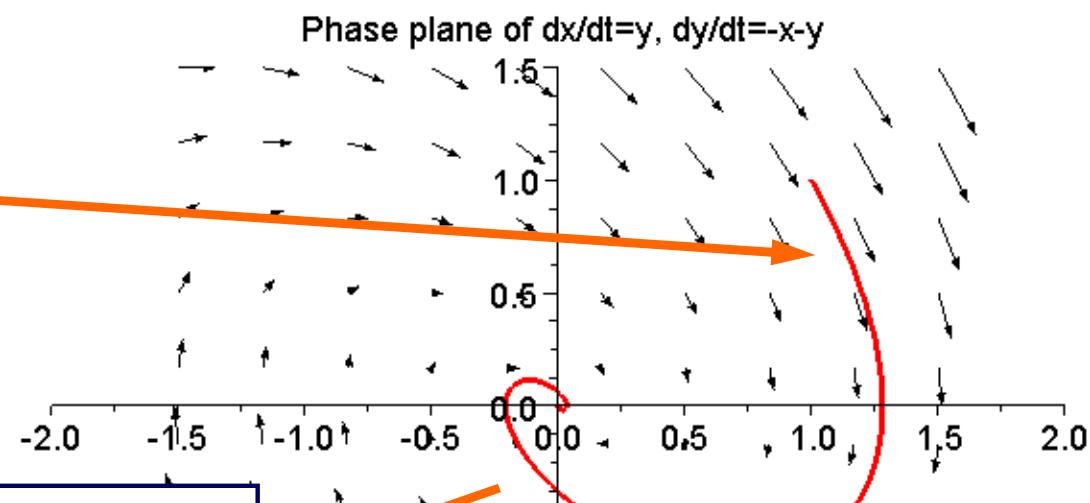
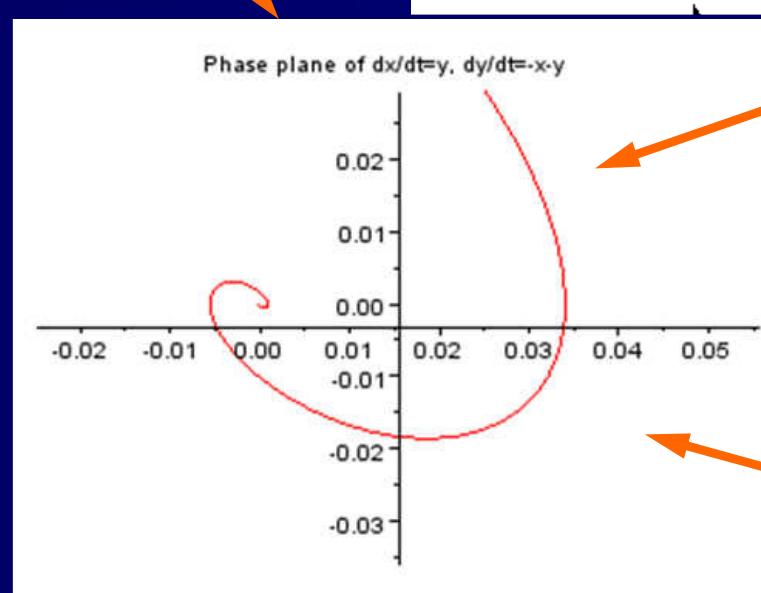
// Plot phase portrait on slope field:
//-----
plot2d(x(1,:),x(2,:),5,'004')
xtitle('Phase plane of dx/dt=y, dy/dt=-x-y')
```

Ex 5-3: plot

Full plot

Phase portrait
with initial
condition [1,1]

Zoomed center
area



Scilab does not put the “haircross” at the origin, which is just as well

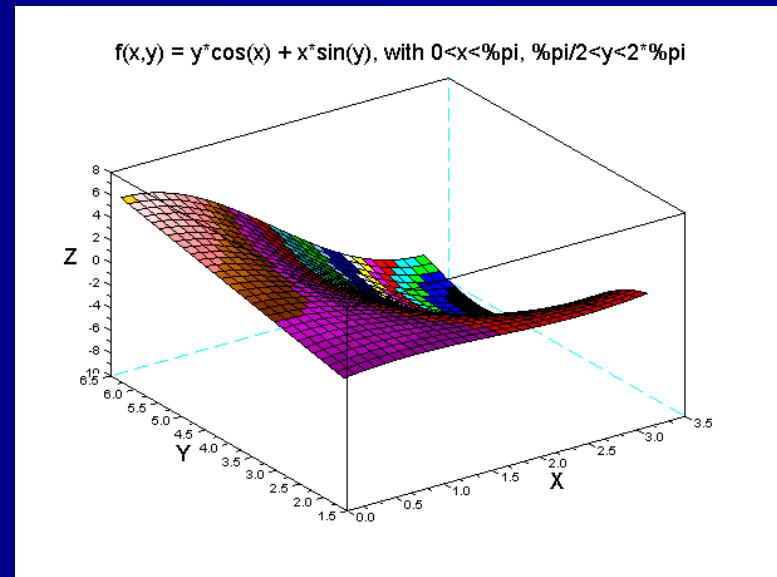
Example 5-4: Simpson's rule, the task

- This example demonstrates integration of double integrals using **Simpson's rule for calculating double integrals**
- Let's first define a subroutine for Simpson's rule and then add code for the function, the area integral of which should be calculated
- In this case we'll repeat the earlier function

$$I = \int_{\pi/2}^{2\pi} \int_0^{\pi} (y \cos(x) + x \sin(y)) dx dy ,$$

but the script can easily be modified for other algebraic expressions

- There are numerous variants of Simpson's rule for double integrals (for an accurate algorithm, see Faires, Burden: *Numerical Methods*, 3rd ed., Brooks Cole 2002). The one given on the next slide is based on Urroz and known as Simpson's 1/9 rule



Ex 5-4: Simpson's rule, algorithm

$$I = \frac{\Delta x \Delta y}{9} \sum_{\substack{i=2 \\ i=i+2}}^n \sum_{\substack{j=2 \\ j=j+2}}^m S_{ij}$$

where we calculate our function $f(x,y)$ in a rectangular domain $R = \{a < x < b, c < y < d\}$

Here x is divided into n and y into m even parts, so that:

$$\Delta x = \frac{b - a}{n}, \quad \Delta y = \frac{d - c}{m}$$

Furthermore:

$$S_{ij} = f_{i-1,j-1} + f_{i-1,j+1} + f_{i+1,j-1} + f_{i+1,j+1} + 4(f_{i-1,j} + f_{i,j-1} + f_{i+1,j} + f_{i,j+1}) + 16 f_{ij}$$

Ex 5-4: Simpson's rule, script

The script is built in four steps:

1. Overall headline comments for the program



2. UDF declaration followed by clarifying comments



3. Body of UDF (next slide)

4. The code for $f(x,y)$ that calls the UDF (two slides down)

```
// double_integration_simpson.sce
//-----
// The program calculates the double integral of the
// function f(x,y) = y*cos(x)+x*sin(y); by calling the
// subroutine simpson_double(x0,xn,n,y0,ym,m,f)
//-----
clear,clc;

function [integral] = simpson_double(x0,xn,n,y0,ym,m,f)

// The function calculates the double integral of
// the function f(x,y) in the region x0<=x<=xn,
// y0<=y<=ym using Simpson's 1/9 rule. The x- and
// y- ranges are divided into n and m subintervals,
// respectively, where both m and n must be even.
// The function modifies m and n if they are odd
```

Ex 5-4: Simpson's rule, script

This is the body of the UDF

It starts by checking and (if necessary) correcting the input parameters n and m

Here we again meet the function feval(). It returns a matrix $z(i,j) = f(x(i),y(j))$

Heart of UDF: The double summation that produces S_{ij} before forming the final answer (output argument)

```
// Check that n and m are even, correct as needed:  
//-----  
if modulo(n,2) <> 0 then // Check that n is even;  
    n = n + 1 // if not, add one  
end  
if modulo(m,2) <> 0 then // Check that m is even;  
    m = m + 1 // if not, add one  
end  
  
// Define x and y increments and region:  
//-----  
Dx = (xn-x0)/n // Define delta x  
Dy = (ym-y0)/m // Define delta y  
x=[x0:Dx:xn] // Region and increments of x  
y=[y0:Dy:ym] // Region and increments of y  
  
// Calculate double integral:  
//-----  
z=feval(x,y,f) // Matrix z(i,j)=f(x(i),y(j))  
Sij = 0 // Initiate Sij  
for i = 2:2:n // Sum Sij along x-axis  
    for j = 2:2:m // Sum Sij along y-axis  
        Sij = Sij + z(i-1,j-1)+z(i-1,j+1)+z(i+1,j-1)+z(i+1,j+1)...  
            +4*(z(i-1,j)+z(i,j-1)+z(i,j+1)+z(i+1,j))+16*z(i,j)  
    end  
end  
integral = (Dx*Dy/9)* Sij // Evaluate integral  
  
endfunction
```

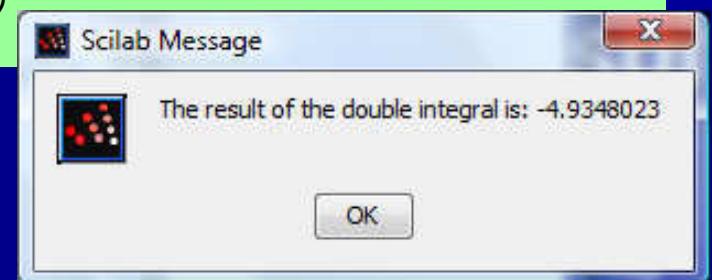
Ex 5-4: Simpson's rule, script & result

Now comes the function $f(x,y)$ that we want to integrate. We start by defining integration limits and steps

An interesting problem emerges: How should one define the calling argument $f(x,y)$? If it is entered as $f(x,y)=y*\cos(x)+ x*\sin(y)$, Scilab will complain that x and y are not defined. The solution is `deff()`

And finally: the answer as displayed on the Message box (the last digit of the earlier demo was more accurate)

```
// Define integration parameters:  
//-----  
x0 = 0; // Lower bound for x  
xn = %pi; // Upper bound for x  
n = 100; // # of subintervals of x  
y0 = %pi/2; // Lower bound for y  
ym = 2*%pi; // Upper bound for y  
m = 100; // # of subintervals of y  
  
// Define function & calculate integral:  
//-----  
deff('[z]=f(x,y)','z = y*cos(x)+x*sin(y)');  
I = simpson_double(x0,xn,n,y0,ym,m,f)  
messagebox('The result of the double integral is:...  
'+string(I))
```



The exact answer is
 $-\pi^2/2 = -4.934802199\dots$

Ex 5-4: Simpson's rule, discussion



- I had big problems with this one. Scilab repeatedly insisted on coming up with the wrong answer. I tried to find the error in several ways:
 - Checked manually that the earlier answer (-4.9348022 , or $-\pi^2/2$) was correct
 - Changed trigonometric functions to exponential equivalents at no avail
 - Checked the algorithm by comparing with solved examples from math and Matlab books
- Finally, when I plugged in the equation in the now several times changed script, **the result came out right**. Most likely I had written $\sin(x)$ instead of $\cos(x)$
- **Lessons learned:** It's hard to see bugs in one's own program
- Another thing: The script uses **nested loops** (for $i = \dots$; for $j = \dots$; ... end; end;). This **should be avoided in Scilab** as far as possible, because the performance is poor in such cases

15. Working with GUIs

The term GUI relates both to Scilab's embedded windows and to user defined interactive windows



[Return to Contents](#)

Introduction

- Scilab's GUI interface was updated with version 5. Old tutorials (e.g. Campbell et al.) are therefore of limited value
- Brief discussions of GUIs can be found in **Kubitzki** and in **Antonelli & Chiaverini** (you can read Scilab scripts in German and Italian even if you don't speak the language)
- Although the GUI interface has improved, the Scilab team still cannot be proud of their achievement
- GUIs is a large subject; the Help Browser identifies about 50 GUI-related functions. We'll be able to cover only a part of them (as always)
- We have earlier seen cases with the dialogue box (`x_dialog()` in Ex. 1-3) and the messagebox (`messagebox()` in Ex. 5-4)
- The first discussion below is about how to tailor Scilab's windows
- Following that we shall look at some user defined dialog windows. A "real" GUI is presented in Example 6-1

Tailoring windows (1/2)

There are four main functions for tailoring either the Console or the Graphics Window:

Addmenu(<gwin>,button, <optional arguments>)	adds new buttons or menus in the main and/or Graphics Window command panels
delmenu()	deletes buttons or menus created by addmenu()
setmenu()	activates buttons or menus created by addmenu()
unsetmenu()	deactivates buttons or menus created by addmenu()

- The numeric **gwin** argument, if present, tells on which Graphics Window the button should be installed
- The button **argument** is a character string that defines a shortcut on the **menu bar**

Tailoring windows (2/3)

- Optional arguments are:
 - submenus character string with names of submenu items
 - action definition list of the type action=list(flag, proc. name)
- This is not the whole truth. The book by Das, which is a collection of Scilab's Help function texts, contains more hints
- As a demonstration of the above said, here is a command that adds the menu Graphics, with submenus New Window and Clear Window, to the Console's menu bar:

The screenshot shows the Scilab Console window. On the left, a yellow box contains the following Scilab code:

```
-->addmenu('Graphics',['New window','Clear window','Close window'])  
-->Graphics = ['scf()','clf()','xdel()']  
Graphics =  
!scf() clf() xdel() !
```

On the right, the Scilab Console window is displayed. A red arrow points from the word "Create" to the "Graphics" tab in the menu bar, which is highlighted in blue. Another red arrow points from the "Create" label to the newly added "Graphics" tab. The "Graphics" tab has a dropdown menu open, showing three options: "New window", "Clear window", and "Close window".

Tailoring windows (3/3)

- You can convince yourself that the added Console menu works by clicking on “New window” to open the Graphics Window and click on “Close window” to close it again
- As the following steps we can deactivate the created menu by the command `unsetmenu()` and delete it with `delmenu()`:

The image shows two screenshots of the Scilab Console window. The top screenshot shows the menu bar with the 'Graphics' option highlighted. The bottom screenshot shows the console window with two lines of code: `-->unsetmenu('Graphics')` and `-->delmenu('Graphics')`. The first line is circled in orange, and an arrow points from it to the second line, which is also circled in orange.

```
-->unsetmenu('Graphics')  
-->  
-->delmenu('Graphics')
```

Interacting with the Graphics Window (1/4)

- Scilab has numerous commands for interacting with the Graphics Window; among these are:

xclick()	Waits for a mouse click, returns a) the number of window where the click occurs, b) position of the click, and c) the number of the mouse button used (left, center, right)
xgetmouse()	Returns the current position of the mouse
seteventhandler()	Sets an event handler for the current Graphics Window
seteventhandler('')	Removes the handler

- The script below is adapted from Help/xgetmouse. It draws a rectangle on the Graphics Window. The rectangle starts off at the location of the mouse pointer at the first click of the left-hand button, and freezes the rectangle at the second click

Interacting with the GW (2/4): script (1/2)

Look for a description of data_bounds under Help/axes_properties (not very helpful)

According to Help/ xclick the first vector element should be numeric, but Scilab requires a name

Look at the arguments of xrect(), it is those that we later play with

The third vector element is set to -1, or mouse pointer has moved (see Help/event handler functions)

```
// rectangle_selection.sce
// The script demonstrates the use of the mouse-related
// commands xclick(), xgetmouse() and xrect() when they
// are used to draw a rectangle in the Graphics Window
// -----
clear,clc,clf;

// Initialize drawing process:
//-----
a = gca(); // Get current Axes
a.data_bounds = [0 0;100 100]; // Boundaries for x & y coordinates
xtitle('Click left mouse button & drag to create a rectangle. ...
Click a second time to freeze') // Display instruction
show_window(); // Put Graphics Window on top

// Start drawing rectangle in the Graphics Window:
//-----
[button,x_coord,y_coord] = xclick(); // Point of mouse button click
xrect(x_coord,y_coord,0,0) // Start rectangle at mouse pointer x & y coordinates
rectangle = gce(); // Get rectangle handle
mouse = [x_coord,y_coord,-1]; // Mouse pointer 1x3 matrix
```

Interacting with the GW (3/4): script (2/2)

The loop starts by checking the status of the mouse. Recall from the previous slide the vector

mouse = [x_coord,
y_coord, -1]

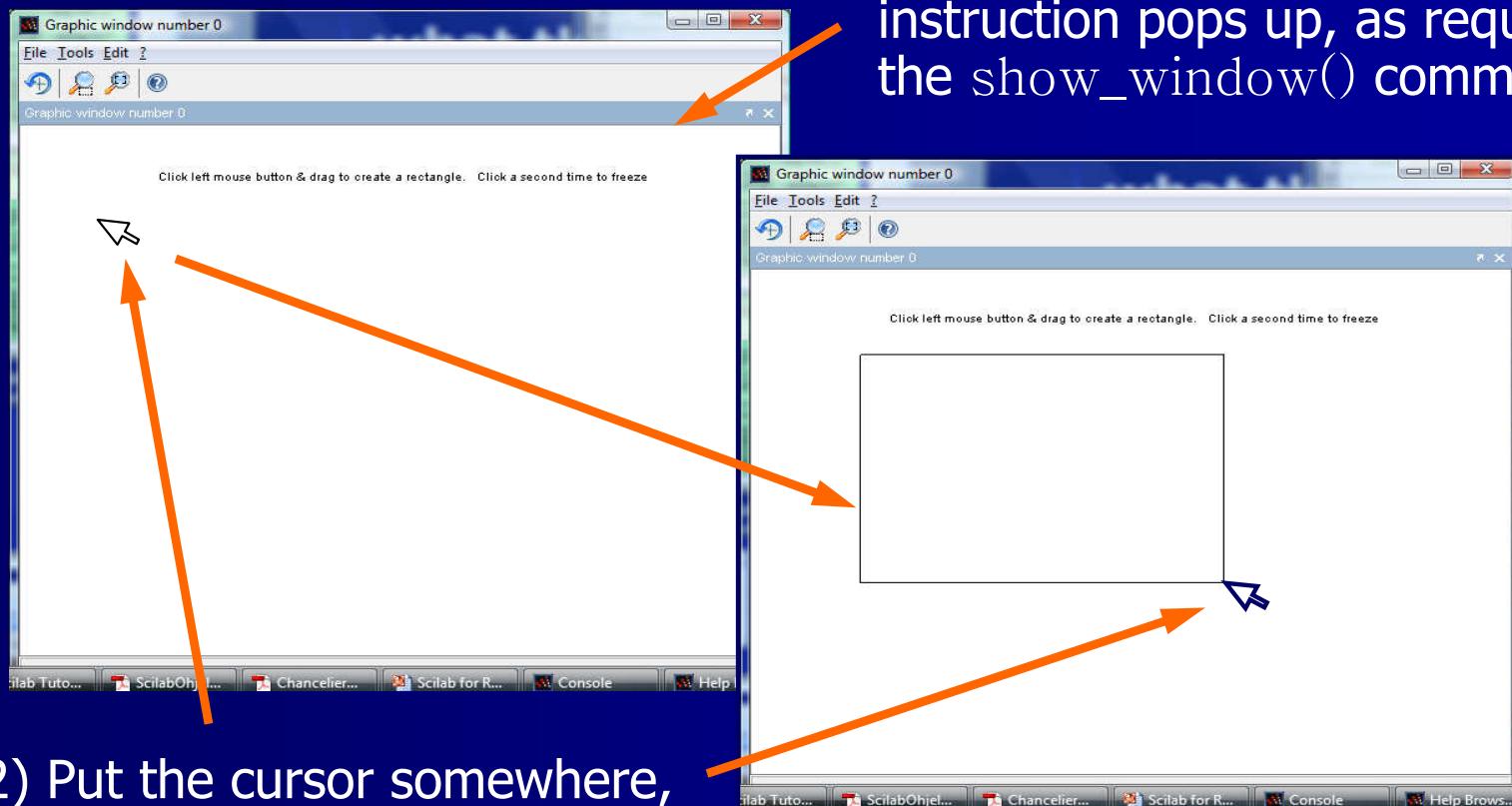
Following that, new data are calculated for the rectangle

The finishing touch is to define new **handle values** (see xrect() arguments above)

```
// Execute mouse commands for rectangle:  
//  
while mouse(3) == -1 do          // Repeat until second click  
    mouse = xgetmouse();           // Check mouse position  
    x_coord1 = mouse(1);           // Mouse location in x-plane  
    y_coord1 = mouse(2);           // Mouse location in y-plane  
    x_origin = min(x_coord,x_coord1); // Define x origin  
    y_origin = max(y_coord,y_coord1); // Define y origin  
    width = abs(x_coord-x_coord1); // Define width of rectangle  
    height = abs(y_coord-y_coord1); // Define height of rectangle  
    rectangle.data = [x_origin,y_origin,width,height];  
    // Change rectangle origin, width and height  
end
```

The while-do-end loop runs forever unless a second mouse button click changes the condition `mouse(3)==-1`. Should a timeout condition be added to the loop?

Interacting with the GW (4/4): what it does



2) Put the cursor somewhere,
click and drag, and click a
second time to freeze

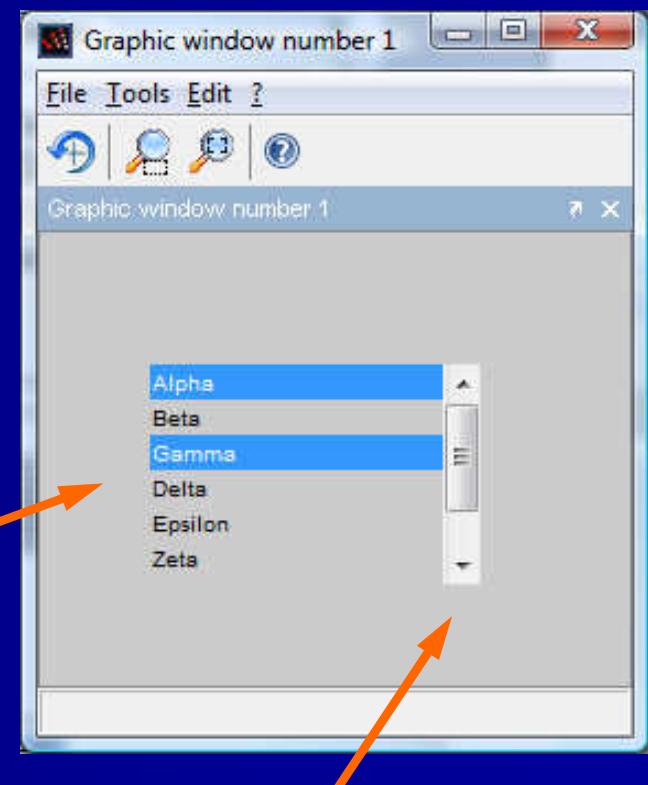
1) The Graphics Window with
instruction pops up, as required by
the `show_window()` command

What do you do with this
feature? Beats me....

GUI demo 1: Introducing figure() & uicontrol()

- Here figure() generates the figure (opens the Graphics Window), uicontrol() creates the graphical user interface object in the GW, and two of the items on the list are highlighted with set()
- The central argument in this case is 'listbox', which defines the list

```
// uicontrol-1.sce /  
  
// A basic GUI exercise /  
  
clc; xdel();  
  
f = figure();           // Create a figure  
h = uicontrol(f,'style','listbox',...    // Create a listbox,...  
    'position',[50 300 150 100]);        // h = handle  
set(h,'string','Alpha|Beta|Gamma..    // Fill the list  
    |Delta|Epsilon|Zeta|Eta|Tau");  
set(h,'value',[1 3]); // Highlight items 1 and 3 in the list
```



Note the **Scrollbar**, it pops up when the height is too small (100) for all items

GUIs: pop-up window functions

- Scilab has several commands for creating pop-up windows.
Note that `x_message()` is **obsolete** and will not work in Scilab 5.2 and later versions; `messagebox()` has to be used instead:

Command	Feature
<code>messagebox()</code>	Message presentation (see Demo 2, Cases 1, 2 & 7)
<code>x_choose()</code>	Alternative selectable from list (Demo 2, Case 3)
<code>x_choices()</code>	As previous but with multiple choices (Demo2, Case 5)
<code>x_dialog()</code>	Window with multi-line dialog (Demo 2, Case 4)
<code>x_mdialog()</code>	As previous but with multiple string parameters
<code>x_matrix()</code>	Vector/matrix input window (Demo 2, Case 6)
<code>list()*</code>	Creates a list of objects (Demo 2, Case 5)

*) Matlab's `struct()` is also available in Scilab

GUIs: messagebox()

The syntax of the messagebox() function is the following:

```
messagebox ("message", "title", "icon", ["buttons"], "modal")
```

Message that you want to convey

Box title (the default is "Scilab Message")

Icon to be placed in the box

1xn vector of strings with button legends

"modal" tells Scilab to wait for user actions (otherwise 0 is returned)

Definable icons are:
"error", "hourglass", "info", "passwd", "question", and "warning"

GUI demo 2: creating pop-up windows (1/5)

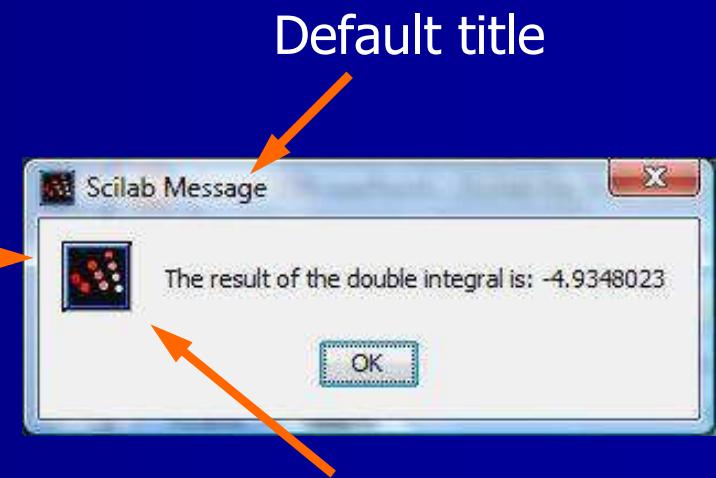
Case 1: Recall that this pop-up window was created by adding the command `messagebox`=('The result of the double integral is: '+ string(I)) at the end of the script of Example 5-4

Case 2:

```
-->m = messagebox('Division by 0: Continue?','WARNING',['Yes' 'No'])  
m =
```

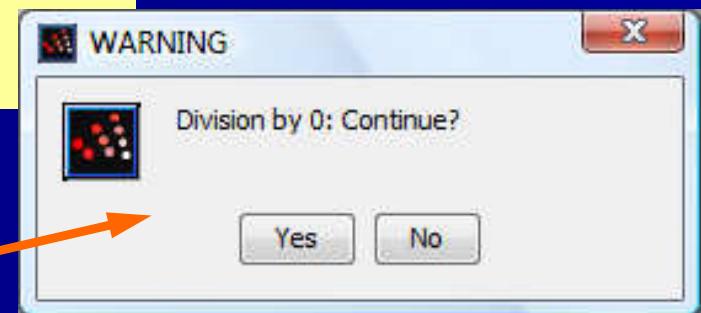
0.

Case 2 is wrong! The Yes/No buttons have no meaning since the case is not declared "modal" and Scilab by design returns the default zero (0)



Default title

Default icon



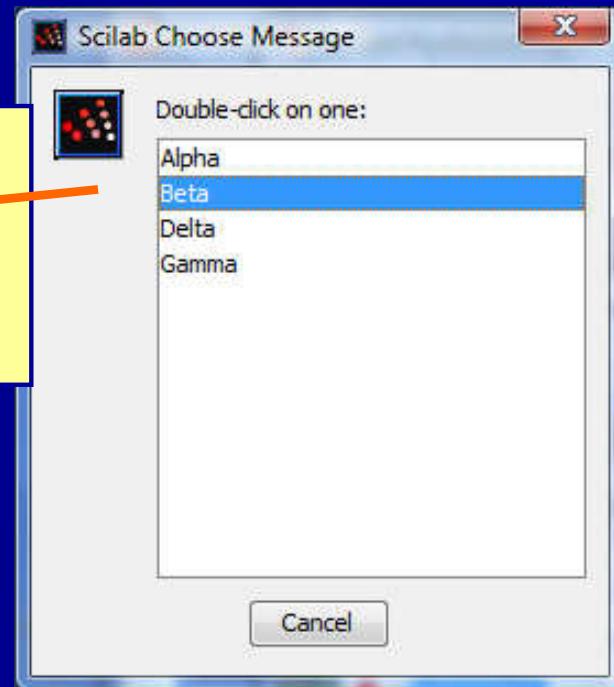
GUI demo 2: creating pop-up windows (2/5)

Case 3: `x_choose()` with four alternatives

```
-->ans=x_choose(['Alpha','Beta','Delta','Gamma'],'Double-click on one:')
```

```
ans =
```

2.



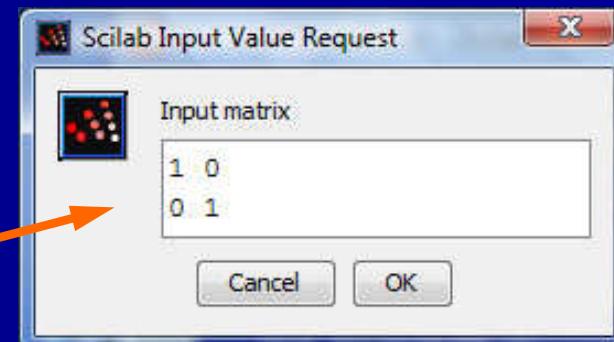
Case 4: `x_dialog()` with input transformed from string to matrix

```
-->answer=evstr(x_dialog('Input matrix',[1 0';0 1']))
```

```
answer =
```

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

Change matrix as needed, click OK



GUI demo 2: creating pop-up windows (3/5)

```
// x-choices_demo.sce  
  
// Demonstrates the x_choices() command /  
  
clear,clc;  
  
list1 = list('Food grading',3,['Excellent','Good','Passable','Poor']);  
list2 = list('Service grading',2,['Excellent','Good','Passable','Poor']);  
list3 = list('Interior grading',4,['Excellent','Good','Passable','Poor']);  
answer = x_choices('Grade restaurant food..  
    service & interior',list(list1,list2,list3))
```

Case 5: x_choices()
with four alternatives
for three cases



```
-->answer  
answer =  
3. 2. 1.
```

Pick your
choices, click
OK, and
Scilab returns
the answer as
a vector*

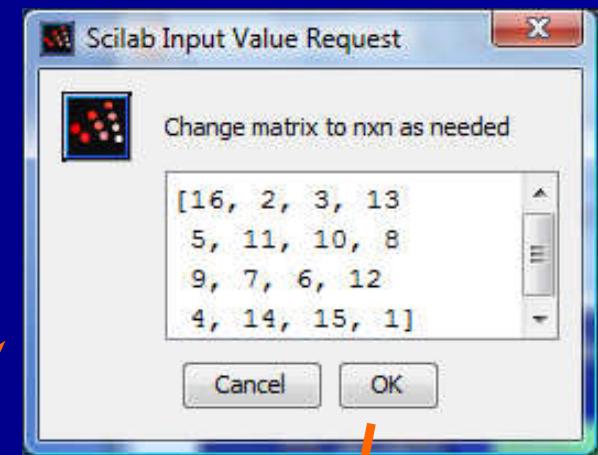
*) Scilab 5.1.1 returns the answer automatically,
with 5.3.1 & 5.3.2 it must be asked for (a **bug?**)

GUI demo 2: creating pop-up windows (4/5)

Case 6: Compute determinant for a matrix A that is given by `x_matrix()`. The assumption is a 3x3 identity matrix



```
// x-matrix_demo.sce  
  
// Demonstrates the use of x_matrix() //  
  
clear,clc;  
  
A = x_matrix('Change matrix to nxn as needed',eye(3,3));  
det(A) // Calculate determinant  
clean(det(A)) // Clean det(A) for small values
```



Change (here to 4x4 Magic Square) and click OK

```
ans =  
- 1.450D-12  
ans =  
0.
```

The answer is the same as earlier in Chapter 5. Problem: It works in Scilab 5.1.1 but not in 5.3.1 and 5.3.2

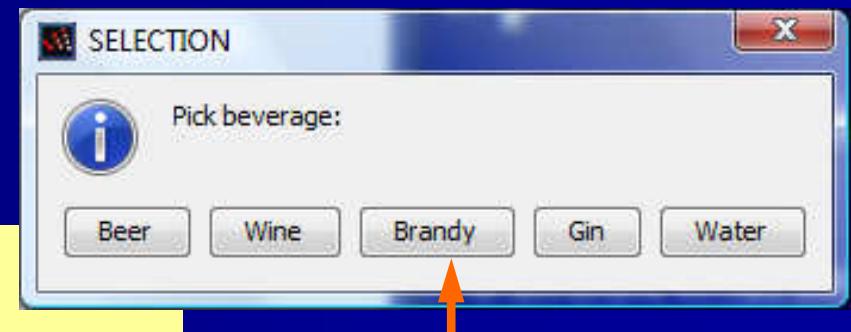
GUI demo 2: creating pop-up windows (5/5)

Case 7: Create an info list of beverage choices using list() and messagebox()

```
-->bew = ['Beer','Wine','Brandy','Gin','Water'];  
  
-->m = messagebox('Pick beverage','SELECTION','info',bew,'modal');
```

```
-->m  
m =
```

3.



Here "Brandy" is selected and the answer returned*

```
-->r = messagebox('Pick','Title','',[1,2],'modal')  
r =
```

2.

*) Same problem repeats. Scilab does not return the answer automatically (in Scilab 5.3.1 it did so with a simpler case, but not in 5.3.2 any more)

GUI: computer screen size & color depth

- The computer screen size is needed if we want to position a GUI at a specific position in the field of view
- For that we need information of the computer screen size. It can be extracted with the argument screensize_xx. There are more alternatives for the _xx suffix, check Help/root_properties for details →
- Another alternative is the number of display color resolution bits. It can be found with the argument screendepth →
- These arguments are used with the function get(), meaning “find out.” See Example 6-1 for a practical case

```
-->get(0,"screensize_px")
ans =
1. 1. 1280. 800.

-->get(0,"screensize_pt")
ans =
0. 0. 960. 600.

-->get(0,"screensize_norm")
ans =
0. 0. 1. 1.

-->get(0,"screendepth")
ans =
24.
```

GUI demo 3: opening a predefined GW, script

- This demo shows how to open a new Graphics Window with predefined size and position
- The size is defined relative to the computer's screen size in points
- The position in the middle of the screen has to be found by trial and error

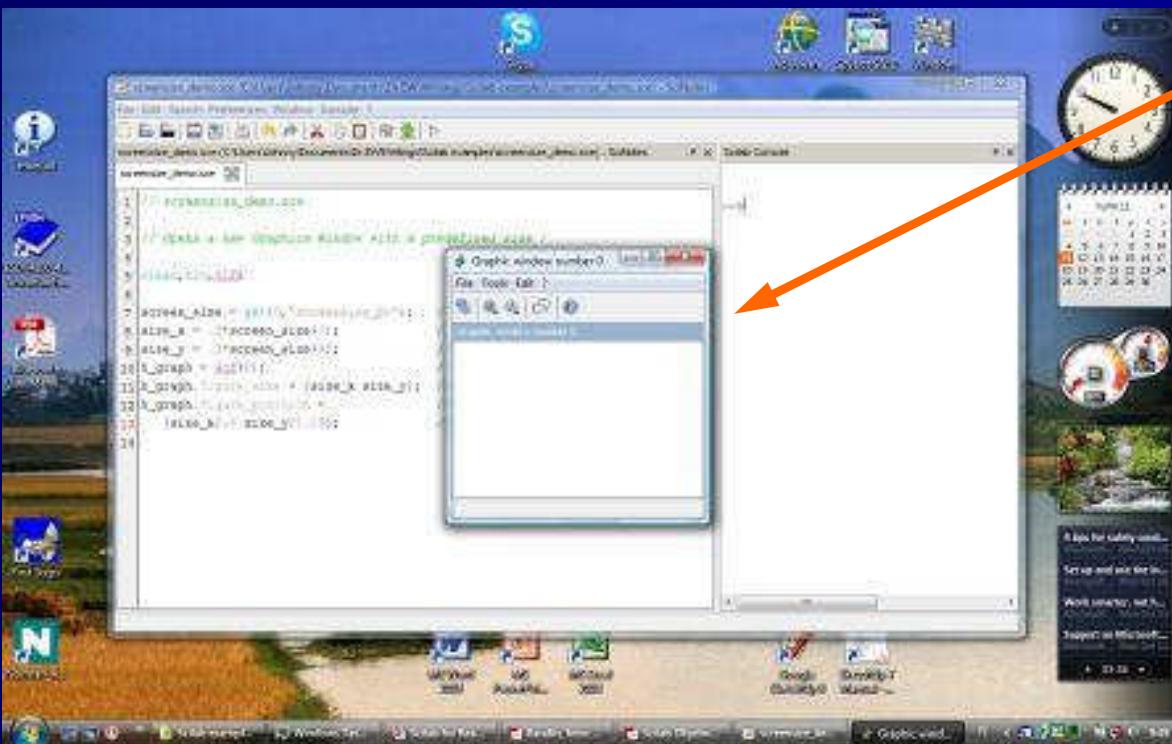
```
// screensize_demo.sce

// Opens a new Graphics Window with a predefined size & location /

clear,clc,clf;

screen_size = get(0,"screensize_pt");           // Find computer screen size
size_x = .3*screen_size(3);                     // .6*screensize_pt 3rd element
size_y = .5*screen_size(4);                     // .8*screensize_pt 4th element
h_graph = scf(0);                            // Open Graphics Window
h_graph.figure_size = [size_x size_y];          // Define GW size
h_graph.figure_position =...                   // Position GW in the...
                                              // [size_x/.6 size_y/1.15];      // middle of the screen
```

GUI demo 3: predefined GW, screenshot



The small GW opens in the middle of the screen (the picture has been compressed and looks muddled)

Note however that the GW size is not exactly in proportion to the defined ratio of the screen size, and it also changes if we select screensize_px instead of screensize_pt (the location changes as well)

GUI shortcomings

- GUIs are not perfected in Scilab. The (messy) text on GUIs in WIKI.Scilab.org/howto/ tells of very old bugs that remain unsolved
- Apart from what is mentioned in Demo 2, Cases 5-7, and in the end discussion of Ex 6-1, I have experienced **problems** with
 - Demo 1, where the listbox may (or may not) flow over the window frame
 - Ex 6-1, where the labels of the slider and first radiobutton sometimes open with reduced font size
- WIKI.Scilab.org/howto/ also mentions the following limitations:
 - Scilab does not allow vertical sliders
 - checkbox == radiobutton
 - slider has only smallstep, no side arrows (and as I found out with Ex 6-1, Scilab gets a lockup when I drag the slider)
 - foreground color is always grey
 - pressed radio/check always pale red (have not tried it)
 - only pushbutton, radiobutton, checkbox, and slider support callback
- The usual recommendation is to use Tcl/Tk when advanced GUI solutions are needed—another program for you to learn

16. File handling

We need file handling e.g. to process measurement data



[Return to Contents](#)

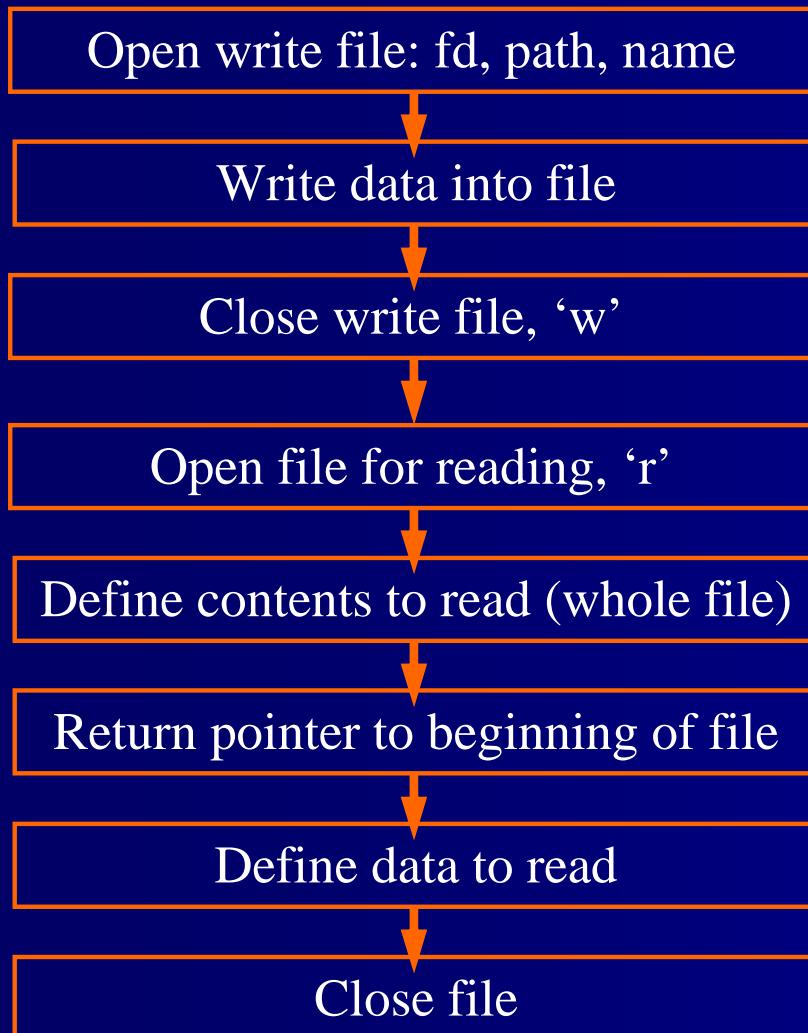
File handling: introduction

- In engineering, data on external files often originate in automated measurements. The data has to be read by Scilab before it can be processed. We'll focus our discussion on this aspect of file handling
- Scilab has a set of commands for file handling, beginning with the commands `mopen()` that opens a file, and `mclose()` that closes it. Between those two we use e.g.:*

<code>mfprint</code> , <code>fprintMat()</code>	Write data to a file (<code>fprintMat()</code> for matrix files)
<code>mfscanf()</code> , <code>fscanfMat()</code>	Read a file (<code>fscanfMat()</code> for matrix files)
<code>mseek()</code>	Move the pointer
<code>menf()</code>	Check end of a file
<code>size()</code>	Check size of an object

*) The full set of i/o functions (~60 in all) can be found under Help/Files: Input/Output functions. Recall the related `load()` function in Chapter 10.

File handling: demo 1 (1/5), introduction



In this demo Scilab creates the data file that is then read; later we'll see how to read from text files created by other programs

Script sequences are shown to the right. The script demonstrates the use of the functions `mopen()`, `mclose()`, `mfprintf()`, `mseek()`, and `mfscanf()`

Pay attention to the following steps: **open** as 'w' file, **close** 'w' file, **open** as 'r' file, **close** 'r' file. The stack pointer moves down as we write into the file and must be returned to the top before we begin to read

File handling: demo 1 (2/5), script

Create the text (.txt) file with mopen(). fd = file descriptor.

Note the argument
'w' ("write") that is
used to create a
new file

Then fill the file with data (in this case created by t) using mfprintf(). Note the odd argument ' %6. 3f\n' that defines the output size (explained below)

```
// file_exercise1.sce

// The script demonstrates the process of 1) creating a text file      /
// on Scilab, 2) closing it, 3) opening it again to be written into,    /
// 4) writing the actual data into the file, 5) reading certain        /
// pieces of data from the file, and 6) closing the read file,         /
// Notice that both close operations are necessary!                      /

clear,clc;

// Create and open a text file for the exercise:
//-----
fd = mopen('H:\Dr.EW\Writings\Scilab examples\file_exercise1.txt','w');

// Create data and write into the exercise file:
//-----
t = (1:1:18);                                // Integers from 1 to 18
mfprintf(fd,'%6.3f\n',t);
```

File handling: demo 1 (3/5), script cont..

After that the file has to be closed

Then opened again to be read ('r')

Next we read in its entirety (the -1)

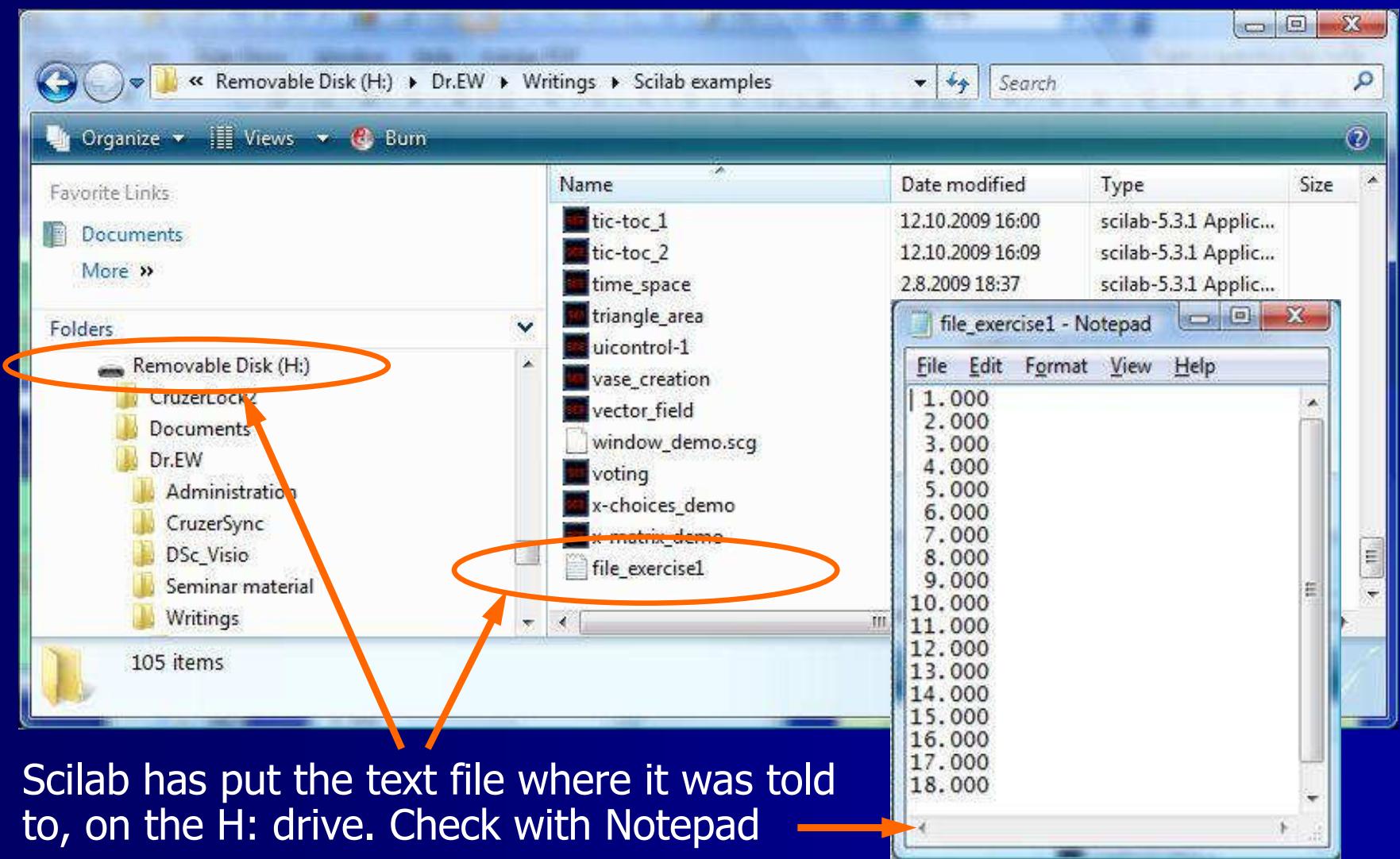
But the pointer must be returned to the top..

before we can define which data we want to see

Finish by closing the file (see below for note on mclose())

```
// Close exercise file:  
//-----  
mclose(fd);  
  
// Open the exercise file for reading:  
//-----  
fd = mopen('H:\Dr.EW\Writings\Scilab examples\file_exercise1.txt','r');  
  
// Read and format file contents:  
//-----  
contents = mfscanf(-1,fd,'%f') // -1 means entire file contents  
  
// Return position pointer to file beginning:  
//-----  
mseek(0,fd) // Following mfscanf(-1, , ) the pointer is at the end  
  
// Read some data from the file:  
//-----  
five_data = mfscanf(fd,'%f %f %f %f %f') // First five data  
three_data = mfscanf(fd, '%f %f %f') // Next three data  
[n,data_9,data_10,data_11] = mfscanf(fd,'%f %f %f') // Three specific..  
// elements  
  
// Close the file:  
//-----  
mclose(fd)
```

File handling: demo 1 (4/5), the .txt file



File handling: demo 1 (5/5), check

The defined read variable contents brings up the contents of the text file on the Console

We can then pick out specific elements from the list

```
-->contents  
contents =  
  
1.  
2.  
3.  
4.  
5.  
6.  
7.  
8.  
9.  
10.  
11.  
12.  
13.  
14.  
15.  
16.  
17.  
18.
```

```
-->five_data  
five_data =
```

```
1. 2. 3. 4. 5.
```

```
-->three_data  
three_data =
```

```
6. 7. 8.
```

```
-->data_11  
data_11 =
```

```
11.
```

```
-->n  
n =
```

```
3.
```

```
-->data_9:11  
ans =
```

```
9. 10. 11.
```

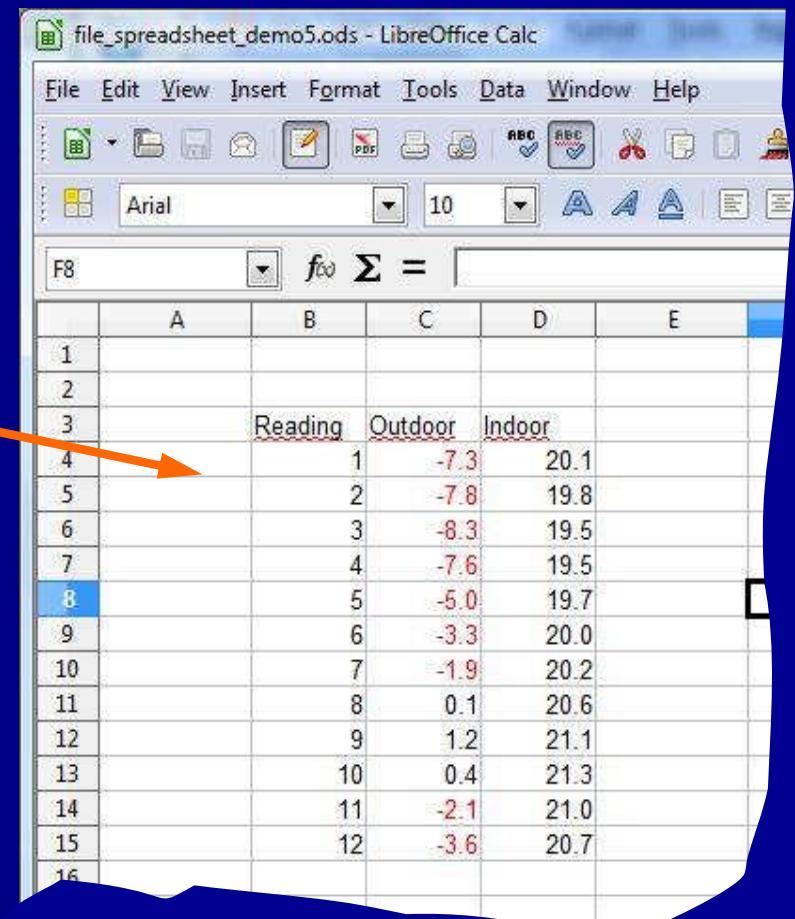
The variables five_data , three_data , and data_11 were defined in the script

n is the # of elements (-1) in the vector it belongs to (4-1)

We can also address specific elements in the column vector and get the answer as a row vector

Spreadsheet data (1/7): Creating data

- Scilab does not interface directly with spreadsheet programs. The data has to be saved as a text file
- I started with the new kid on the block, **LibreOffice Calc**. The data is the output from an indoor/outdoor temperature measurement
- The process of saving LibO and OOo data as a .csv text file is explained later
- If you do it in Excel you just save it as Text (Tab delimited). Do not select Unicode Text because Scilab cannot read it



	A	B	C	D	E
1					
2					
3		Reading	Outdoor	Indoor	
4		1	-7.3	20.1	
5		2	-7.8	19.8	
6		3	-8.3	19.5	
7		4	-7.6	19.5	
8		5	-5.0	19.7	
9		6	-3.3	20.0	
10		7	-1.9	20.2	
11		8	0.1	20.6	
12		9	1.2	21.1	
13		10	0.4	21.3	
14		11	-2.1	21.0	
15		12	-3.6	20.7	
16					

Spreadsheet data (2/7): Data saved as .csv file

- And here is the LibO data saved as `file_spreadsheet_demo5.csv` and seen in WordPad (the figure 5 reflects the fact that it is my fifth attempt to get it right)
- Let's see if Scilab can read the `.csv` file. There are two command options:
 - `M = fscanfMat()` for a matrix of real numbers (text data is ignored)
 - `[M,text] = fscanfMat()` for a string matrix
- The output for both alternatives are shown on the next slide
- After that we can write a script to plot the data



	"Reading"	"Outdoor"	"Indoor"
1	-7.3	20.1	
2	-7.8	19.8	
3	-8.3	19.5	
4	-7.6	19.5	
5	-5.0	19.7	
6	-3.3	20.0	
7	-1.9	20.2	
8	0.1	20.6	
9	1.2	21.1	
10	0.4	21.3	
11	-2.1	21.0	
12	-3.6	20.7	

Spreadsheet data (3/7): .csv file read by Scilab

```
-->M = fscanfMat('I:\file_spreadsheet_demo5.csv')  
M =
```

```
1. - 7.3 20.1  
2. - 7.8 19.8  
3. - 8.3 19.5  
4. - 7.6 19.5  
5. - 5. 19.7  
6. - 3.3 20.  
7. - 1.9 20.2  
8. 0.1 20.6  
9. 1.2 21.1  
10. 0.4 21.3  
11. - 2.1 21.  
12. - 3.6 20.7
```

M = fscanfMat()

[G,text] = fscanfMat()

```
-->[G,text] = fscanfMat('I:\file_spreadsheet_demo5.csv')  
text =
```

G = "Reading" "Outdoor" "Indoor"

```
1. - 7.3 20.1  
2. - 7.8 19.8  
3. - 8.3 19.5  
4. - 7.6 19.5  
5. - 5. 19.7  
6. - 3.3 20.  
7. - 1.9 20.2  
8. 0.1 20.6  
9. 1.2 21.1  
10. 0.4 21.3  
11. - 2.1 21.  
12. - 3.6 20.7
```

Note: If you work with MS Excel you use of course the ending .txt instead of .csv (CSV stands for Comma Separated Variable)

Spreadsheet data (4/7): script for plotting (1/2)

The fscanfMat()
command cannot be
split on two rows
(even if it is not
needed in this case)

The size(name,'r')
function is used to
determine the number
of matrix rows

Matrix columns form
separate vectors

```
// spreadsheet_data_plot.sce
// The script reads data from the test file      /
// file_spreadsheet_demo5.csv, determines its   /
// length, and plots its two measurement sets   /
clear,clc,clf;

// Open the file, determine number of rows,
// and form vectors of its columns:
//-----
data_file = fscanfMat('file_spreadsheet_demo5.csv');
           // Opens text file
rows = size(data_file,'r');    // Determine number of rows
readings = data_file(:,1);     // Column 1, reading # (redundant)
outdoor = data_file(:,2);      // Column 2, outdoor temperature
indoor = data_file(:,3);       // Column 3, indoor temperature
```

Spreadsheet data (5/7): script for plotting (2/2)

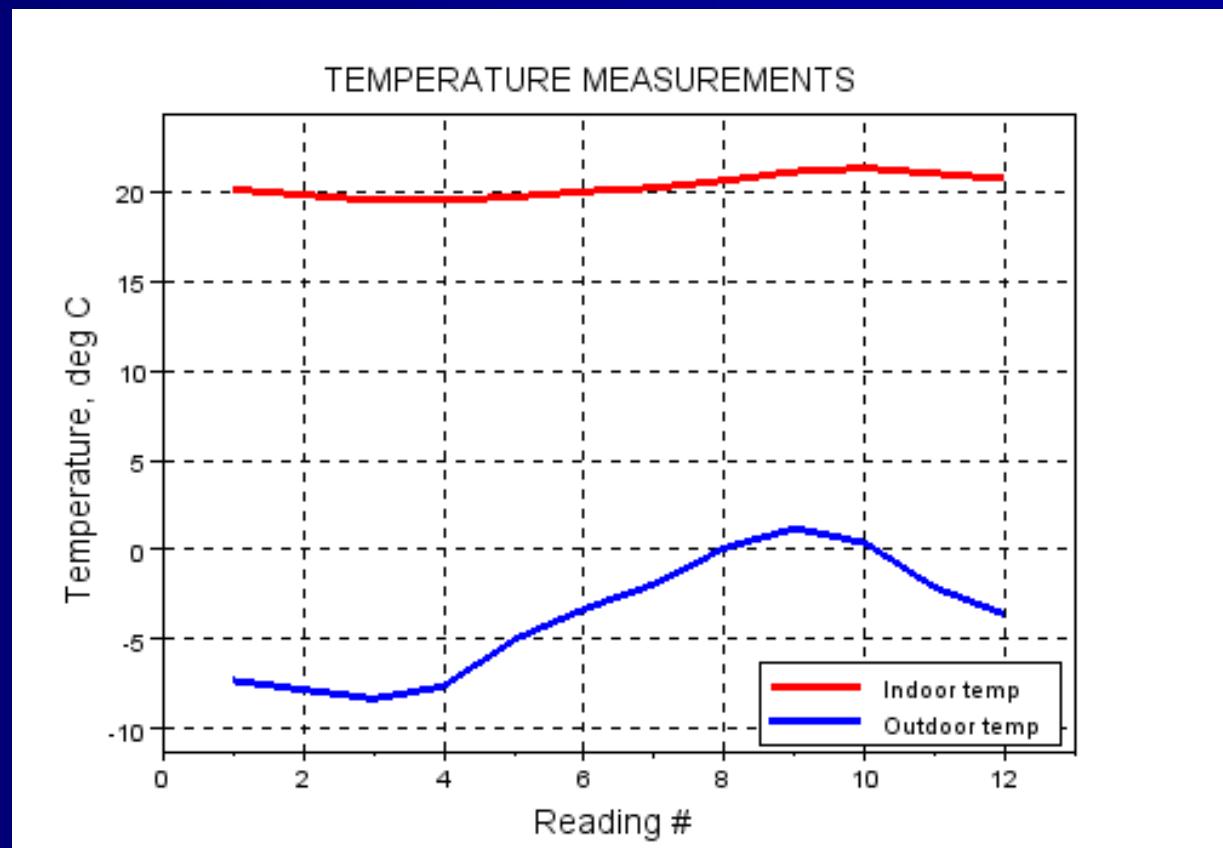
The plot command uses the obsolete `plot2d()` syntax that we have seen before. The reason for having it here is that `plot2d()` with the `frameflag` argument of the new syntax does not work when two graphs should be fused into one plot: The second plot destroys the first one, and when the `rect` argument is included Scilab responds with an error message (know it, tried it)

```
// Assume outdoor temp always lower
// than indoor and form plot commands:
//-----
ymin = min(outdoor);           // Determine min temp
ymax = max(indoor);           // Determine max temp
dy = (ymax-ymin)/10;           // Define frame margin
rect = [0,ymin-dy,rows+1,ymax+dy]; // Define plot frame
x = linspace(1,rows,rows);
plot2d(x,indoor,5,'011',' ',rect) // Plot inodr temp
plot2d(x,outdoor,2,'000')        // Plot outdor temp
xgrid(1)                        // Add grid
xtitle('TEMPERATURE MEASUREMENTS','Reading #',...
      'Temperature, deg C')
legend('Indoor temp','Outdoor temp',4)
```

Spreadsheet data (6/7): plot

Simple plot, but the main point with this exercise is to show how to go from spreadsheet data to a text file and then to plot the data

And then we turn to the question of how to create text files with LibreOffice Calc and OpenOffice.org Calc (next slide)



Spreadsheet data (7/7): Text data in LibO & OOo

Save as Text CSV (.csv)
and select Tab in the Field delimiter dropdown menu
of the window that opens.
That's it

The saved .csv file looks messy
if you open it with Excel, but it
is ok in Notepad and WordPad

	A	B	C
1	"Reading"	"Outdoor"	"Indoor"
2	1-7.320.1		
3	2-7.819.8		
4	3-8.319.5		
5	4-7.619.5		
6	5-5.019.7		
7	6-3.320.0		
8	7-1.920.2		
9	80.120.6		
10	91.221.1		
11	100.421.3		
12	11-2.121.0		
13	12-3.620.7		
14			
15			
16			
17			

mopen()

- The mopen() function is of course more intricate than what one can understand from the discussion above. Forgetting binary and text files, the general structure of mopen() is:

[fd <,err>] = mopen(file_name <,mode>)

where

- file_name is the entire path of the file, including its name
- mode defines what to do with the data, e.g.:
 - r, read an existing file
 - w, create a new file & write into it, alt. overwrite data in existing file
 - a, append, open a file and add data to the end
- fd, file descriptor, temporary name of the file
- err, error parameter. err = 0 if the file is successfully opened, err <> 0 if file opening failed (merror() is a function related to the err argument)
- It can be a good idea to check the err parameter after a file has been opened (has not been done in Demo 1)

mclose()

- A file that has been opened with mopen() should be closed with the mclose(fd) command even if it is automatically closed when Scilab closes. However, pay attention to the following ambiguous statement in Scilab's Help Browser:

"mclose must be used to close a file opened by mopen. If fd is omitted mclose closes the last opened file.

Be careful with the use of [mclose('all')] ... because when it is used inside a Scilab script file, it also closes the script and Scilab will not execute commands written after mclose('all')."

mfprintf(), fprintfMat()

- The mfprintf() command is used to convert, format ,and write data in an opened text file
- The general structure of mfprintf() is:

```
mfprintf(fd, '<text a> format_1 <text b> format_2  
<text c> format_3...', value_1, value_2, value_3...)
```

- Which means that each value that we want to print is declared with an optional **text**, the **format** to be printed in (both within a single pair of quotation marks), and the **value** to be printed
- Format declarations are given on the next slide
- The format demo two slides down should give a better grasp of what it all means. If you ask me, it looks really messy...
- The fprintfMat() command is used to write a matrix in a file. See Help for details

Format definitions

- Recall the arguments '%6.3f\n' and %f in [File handling Demo 1.](#) They are part of a set of format definitions:
 - %d for integers (e.g. 1230)
 - %f for floating point presentation (e.g. 12.30987)
 - %e for exponentials (e.g. 1.2345e+002)
 - %s for text (string) presentation (e.g. Hello World!)
 - %6.3f to define the output size
 - the 6 is for the total number of figures
 - the 3 is for the number of figures after the decimal point
 - \n "go to a new line"
 - \t "use a horizontal tabulator"
- Some definition combinations, like %6.3f\n, are possible

Format demo: script (1/2)

This demo aims at clarifying the use of format declarations:

```
// file_format_demo.sce

// Demonstrates the use of mfprintf() format definitions.    /
// Pay attention that with several variable to be printed,   /
// all formats are declared (inside a single pair of citation  /
// marks) before the variables are defined.                  /

clear,clc;

// Create a new test file for writing:
//-----
fd = mopen('H:\Dr.EW\Writings\Scilab examples\file_format_demo.txt','w');

// Some variable to play with:
//-----
A = 123.45678901;
a = 0.3;
b = 1.23e-02;
c = a + %i*b;
text = 'Hello World';
```

Just initial declarations here.
The real stuff is on the next slide

Format demo: script (2/2) & text file

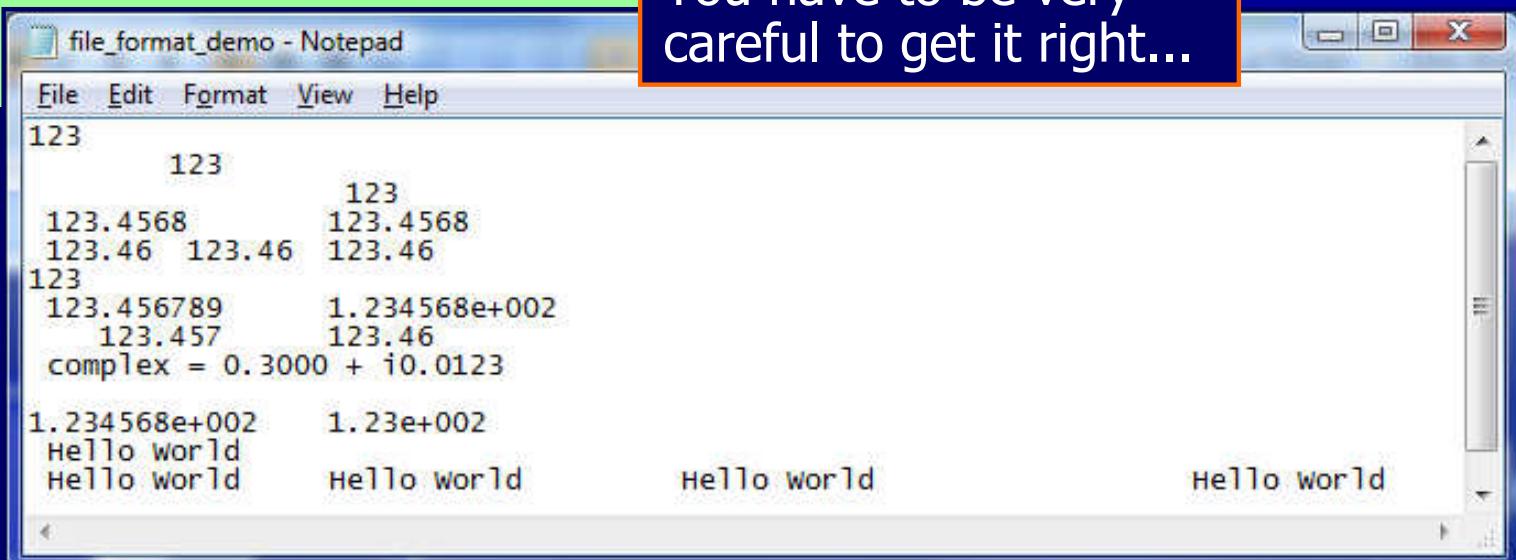
```
// Several outputs to be demonstrated:  
//-----  
mfprintf(fd,'%d\n %10d\n %20d\n %8.4f\t %8.4f\n %5.2f\t %5.2f\t %5.2f\n',...
    A,A,A,A,A,A,A);  
mfprintf(fd,'%d\n %f\t %e\n %10.3f\t %6.2f\n complex = %3.4f + i%3.4f\n\n',...
    A,A,A,A, real(c), imag(c));  
mfprintf(fd,'%e\t %5.2e\n %s\n %5s\t %10s\t %15s\t %20s\t\n',...
    A,A, text, text, text, text, text);
```

```
// Close the opened file:
```

```
//-----  
mclose(fd);
```

Remember
to close!

You have to be very
careful to get it right...



No optional
text is used
in any of
the cases

mfscanf(), fscanfMat()

- We used mfscanf() in Demo 1 to read (scan) data from a file. Two examples of its use:
 - contents = mfscanf(-1, fd, '%f'). With this argument it reads the whole contents of the file and formats it
 - four_values = mscanf(fd, '%f %f %f %f'). Reads the four first data in the file
 - After reading data, the stack pointer remains where it is and we must use the mseek(n,f) command to shift it to a new location. The first row in the stack is numbered 0, as indicated by mseek(0,fd) in Demo 1
- In the discussion of spreadsheet data we used the fscanfMat() function to read the data contained in a .csv file. The function has two alternative call sequences:
 - fscanfMat(filepath,<opt_arg>) to read the numeric part only of scalar matrix data in a text file
 - [M,text] = fscanfMat(filepath,<opt_arg>) to read the data and include the first non-numeric lines
 - The default optional argument is %1g. Check with Help for other options

17. Animation

A brief introduction to creating
dynamic graphics



[Return to Contents](#)

Introduction

- Animations are a sequence of plots on the Graphics Window; executed by showing a plot, freezing it while an incremental shift is being calculated, and then swapping the old plot for the new one.* With correct speed and increments it gives the illusion of continuous movement
- There are two main modes for creating animations:
 - **Real time mode.** The animation runs while the script is being executed, with the speed being determined by the incremental shifts and computer speed. The execution can be influenced (slowed down) by the realtimeinit() and realtime() functions
 - **Playback mode.** Possible in Matlab with the getframe and movie commands, but **Scilab lacks this alternative**
- A tool for producing animations is the pixmap handle command and the show_pixmap() function. Example 6-2, however, does not use the pixmap command

*) Unless one wants to retain the whole sequence, as in Example 6-2.

Demo 1 (1/4): Introducing pixmap & xfarc()

- This demo is adapted from Antonelli & Chiaverini. It exhibits in particular the pixmap and show_pixmap() pair of commands
- pixmap="on"/"off"
 - The pixmap mode* is used to achieve a smooth animation. With the handle command pixmap="on" the display is refreshed only when called on by the command show_pixmap()
 - Compare this case with the drawlater() - drawnow() pair in ordinary plotting
- The script uses the xfarc() function to fill the moving pie. Related Scilab functions are xfarc(), xarcs(), and xarc()
- xfarc() is used instead of xfarc() because the latter has no provision for defining plot color by arguments, and its Axes handle gca() does not recognize any children that would allow colors to be defined

*) Also called “double buffer mode” because the picture is first created in one buffer before being pushed to the second (the Graphics Window).

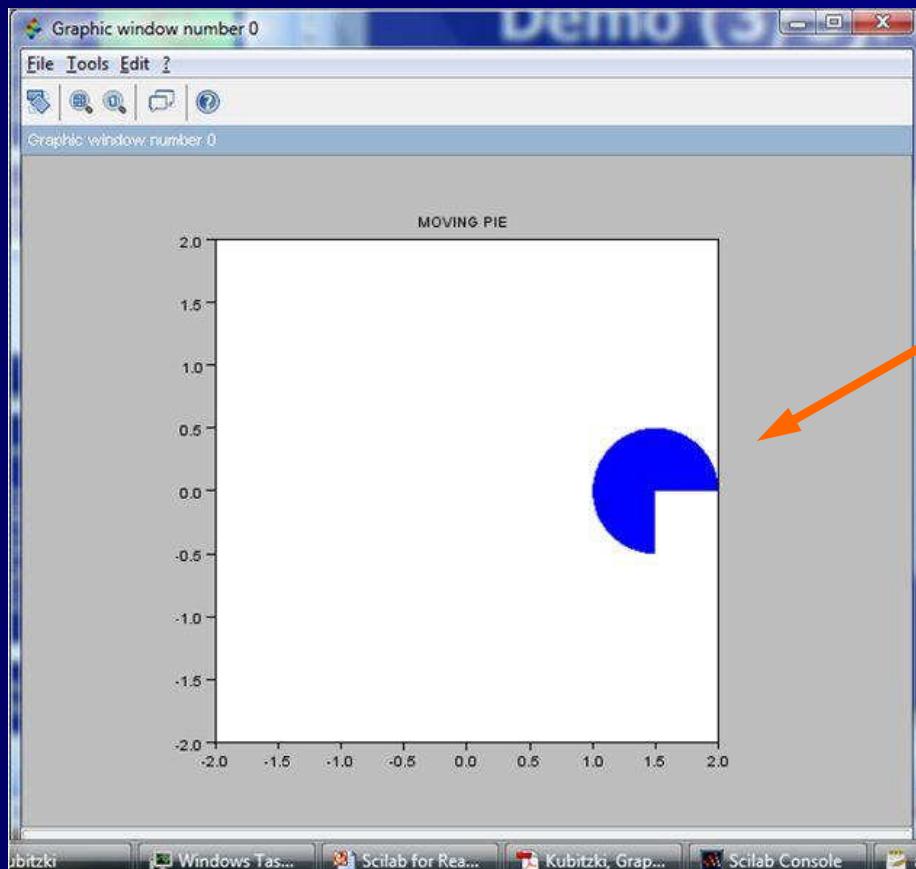
Demo 1 (2/4): moving pie, script

- Missing x and y values are substituted by (%nan)
- Only frameflag=3 works in this case
- Note the imaginary values of theta1 & theta2. Their relative values (2π & 10π) determine the five loops that the pie makes before finishing the full circle
- xfarc() requires six vector values as its argument. The color code is optional (the default color is black)

```
// animation_ball.sce
// Creates a cut pie that makes 5 loops while
// moving around a circle. Demonstrates the use of
// the pixmap - showPixmap() pair of commands,
// and the use of xfarc() in drawing
//
clear,clc;

steps = 250;      // # of animation steps
r1 = 0.5;         // Pie size
r2 = 0.5;         // Loop size
f = gcf();        // Figure handle
f.pixmap = "on"; // Create before display
for i=1:steps
    clf();          // Erase pie after each step
    plot2d (%nan,%nan,frameflag=3,... // Define figure
            rect=[-2,-2,2,2],axesflag=1)
    xtitle('MOVING PIE');
    theta1 = i*2*pi/steps;
    theta2 = i*10*pi/steps;
    c = [cos(theta1)+r2*cos(theta2),...
          sin(theta1)+r2*sin(theta2)]; // Define pie...
    xfarc([c(1)-r1, c(2)+r1, 2*r1,... // Plot pie...
           2*r1, 0, 360*48]', 2); // color=2
    f.background = color('grey');
    showPixmap(); // Display created graphics
end
f.pixmap = 'off'; // Exit pixmap mode
```

Demo 1 (3/4): moving pie, frozen plot



Here is the blue pie in its combined start and finish position

The completion of the full circle in 250 steps takes about 10 seconds with my 1.6 GHz dual-core processor

Demo 1 (4/4): discussion

- Odd things happened while I tried to get this one going
- The Graphics Window mostly opened as shown above, but I have also seen a black ball (that was before I changed it to a pie) on a red background surrounded by a yellow frame topped by a red title—with the animation running just as smoothly as it should
- When I changed `frameflag=3` to `frameflag=2` the dot rotated around to the lower left-hand corner, and when I changed back again Scilab told that the handle is not valid any more. Just go on and reload...
- I also saw the size of the Graphics Window change from execution to execution for no obvious reason
- In short, these events give the feeling that animation—together with GUIs—is **not a top priority of the Scilab team**

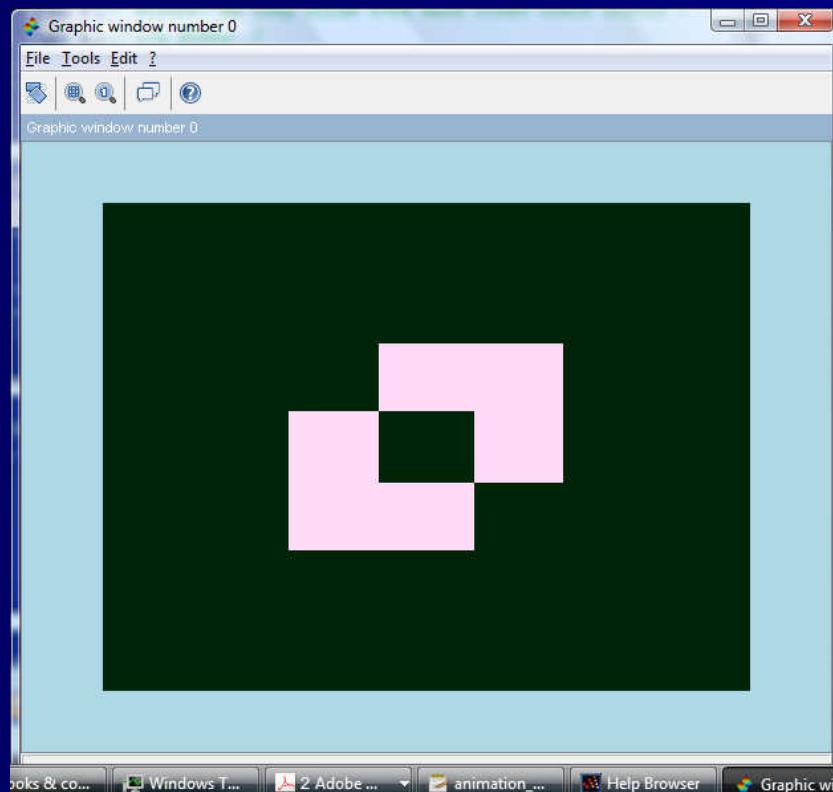
Demo 2 (1/2): moving rectangles

- This demo is adapted from Chancelier et al.
- It's an attempt to demonstrate the use of the XOR command in `f.pixel_drawing_mode='xor'`, here NOR instead of XOR for reasons told below
- The rectangles move on top of a grey background
- The rectangles are drawn with `xfrect()` without color code, they are therefore black
- The rectangles move diagonally from corner to corner in 200 steps

```
// animation_rectangles.sce
```

```
// Two rectangles slide diagonally over the Graphics /  
// Window. As they slide over each other their colors /  
// are NORed. The solution is only partly successful /  
  
clear,clc,clf();  
  
f=gcf();  
f.pixmap='on';      // Double buffer mode  
f.pixel_drawing_mode='nor';    // NOR mode  
f.background=color("lightblue");  
  
ax=gca();  
ax.data_bounds=[0,-4;14,10];    // Plot limits  
ax.margins=[.1 .1 .1 .1];      // Plot framed  
ax.background=color("lightgrey");  
max_pos = 10;                  // Max position of rectangles  
  
k=%nan;                      // Auxiliary parameter  
xfrect(k,k,4,4);              // First black rectangle  
e1 = gce();  
xfrect(max_pos-k,max_pos-k,4,4); // Second rectangle  
e2=gce();  
  
for k=linspace(1,10,200)        // Animation loop  
e1.data(1:2)=k;  
e2.data(1:2)=max_pos-k;  
show_pixmap()                  // Show double buffer  
end
```

Demo 2 (2/2): frozen plot



Here is the animation in progress. The NOR function does its job, but otherwise **something is quite wrong**: We do not have black rectangles moving across a light grey background

The problem is that the command `f.pixel_drawing_mode='nor'` operates on the whole screen, not just on the moving rectangles as intended by Chancelier et al. For that reason the XOR operation they use is even worse than NOR

I decided to leave the demo in this state. Those who are interested can find **a better solution in Steer's Scilab Graphics, p. 28**

Demo 3 (1/3): a 3D object, script (1/2)

- Now we'll look at a geometrically more challenging object, a 3D plot that moves both in azimuth and elevation
- Data bounds are not defined separately, they are changing with surface mesh resolution
- The first plot command only defines axes labels

```
// rotating_surface.sce  
  
// The 3D surface is first rotated and then /  
// tilted, after which its position is locked /  
  
clear,clc,clf;  
  
// Initialize:  
//-----  
f=gcf();  
f.pixmap="on";  
clear_pixmap();  
t=%pi/20*(-20:20); // Bounds & mesh resolution  
  
// First plot command, defines labels:  
//-----  
plot3d1(t,t,sin(t)*cos(t),%nan,%nan,...  
'x_axis@y_axis@z_axis');
```

Demo 3 (2/3): a 3D object, script (2/2)

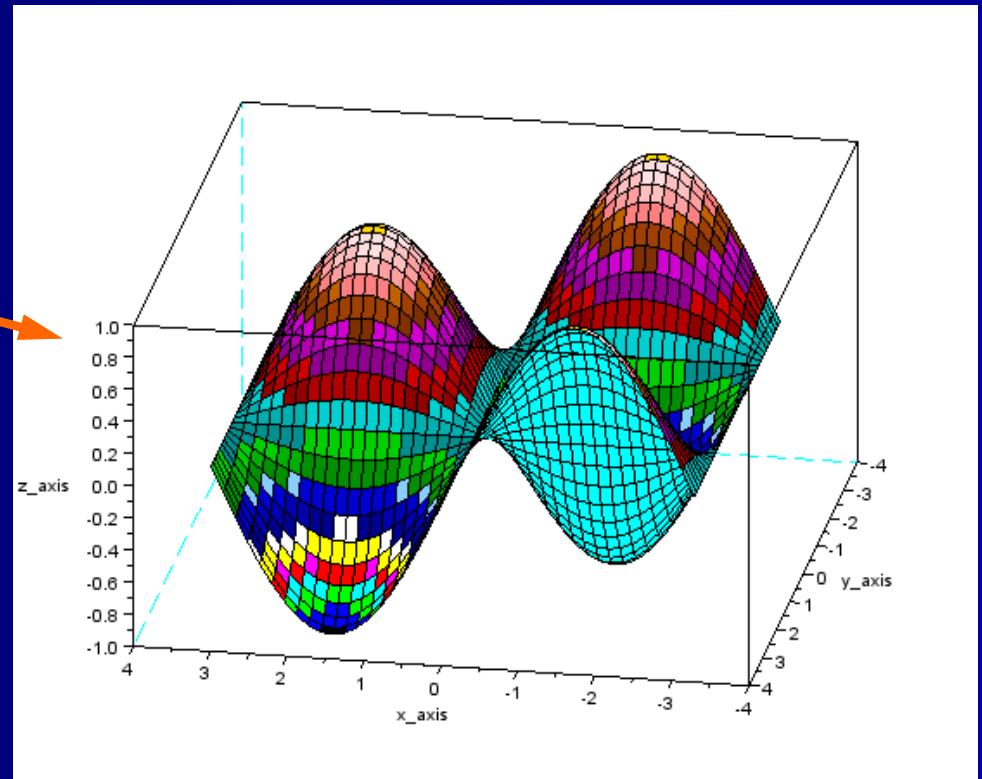
- The surface rotates around the z axis, starting at 45° and finishing at 100°, while the tilt angle is constant at 45°
- When finished rotating, the surface tilts around the x axis from 45° up to 80°, with the rotation angle constant at 100°
- With my 1.6 GHz laptop the animation does not run perfectly smoothly, the jumps from step to step are noticeable

```
// Set speed and turn object:  
//-----  
step = 2;           // Step size --> 1/speed  
for angle1 = 25:step:100,    // Rotate loop  
    plot3d1(t,t,sin(t)*cos(t),angle1,45)  
    show_pixmap();  
end  
for angle2 = 45:step:80,      // Tilt loop  
    plot3d1(t,t,sin(t)*cos(t),100,angle2)  
    show_pixmap();  
end  
f.pixmap="off";
```

Demo 3 (3/3): a 3D object, plot

The surface has reached its destination: rotated to 100° (azimuth) and tilted to 80° (elevation)

While testing various parameters I saw this message on the Console (long list). It disappeared when I re-run the script



```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException  
at javax.swing.plaf.basic.BasicTextUI$RootView.paint(Unknown Source)
```

.....

18. Miscellaneous

A hotchpotch of philosophy and
realism that hopefully is of use



Return to Contents

The problem-solving process

The problem-solving process for a computational problem typically goes through the following steps (not a textbook definition):

1. Define the problem (answer the question "What's the problem?")
2. Outline a way of solving the problem (block diagram, DFD, etc.)
3. Define equations and/or algorithms (solve the math problem)
4. Transform steps 2 & 3 to a software outline or architecture
5. Do the coding in steps and test each step before proceeding
6. Validate the solution (does it do what it should do, with all input values (especially 0 & ∞), and nothing more than it should do?)

The boundaries between these steps can be blurred, iterations are mostly needed, and one or two of the steps may be more important than the others. Each step also requires a number of subtasks to be performed. But in general it helps to have this approach in mind when attacking a problem.

Good program structures

- Keep in mind that a program is characterized by a) its structure and b) by what it does
- Give variables clear and meaningful names; use single letters only for x, y and z axes, loop counters (j,k), and the like
- Split the code into logical entities with the help of subroutines
- Separate structural entities by empty rows and headline comments
- Indent rows (loops, print commands, etc) for increased clarity
- Be liberal with the use of comments, keep space between the command and its row comment
- **Simple is beautiful;** a good program is short and unambiguous
- For a more thorough discussion, see textbooks on software engineering

"Always program as if the person who will be maintaining your program is a violent psychopath that knows where you live."

Martin Golding

Programming pitfalls (1/4)

- Computer programs cannot be tested to guarantee 100% reliability. There is the danger of both hidden and—in retrospect—obvious bugs.* Avoiding common programming pitfalls should be a minimum goal and requires that we are familiar with them
- Basic Scilab tutorials do not pay much attention to programming pitfalls. “Eine Einführung in Scilab” by Bruno Pinçon (original in French, for me German is far easier) is an exception. Its last chapter briefly discusses programming pitfalls. **Worth taking a look at**
- A search on the web for “Matlab pitfalls” provides some hints. There is also a useful discussion in Chapter 9 of Hahn, Valentine: *Essential Matlab for Engineers and Scientists*, 3rd ed., Butterworth-Heinemann, 2007

*) The term “bug,” according to anecdote, was coined in 1947 when Grace Hopper (“Grandma Cobol”) of the US Navy identified a computer problem being caused by a moth in a relay (/tube/connector, the story varies). The original “bug” was thus a hardware related problem and lethal to the bug.

Programming pitfalls (2/4): error types

- Programming errors can broadly be grouped into the following types
 - Logical errors, meaning errors in the algorithm used to solve a problem
 - Syntax errors, meaning mistakes in the construction of Scilab statements
 - Rounding errors, meaning errors due to limited computer accuracy
- **Logical errors** are mostly the result of our limited understanding of the problem at hand and/or our limited knowledge of algorithms in general and Scilab in particular
- **Syntax errors** are generally speaking due to human limitations: oversight, carelessness, forgetfulness, and the like. Typical cases are misspelling, mismatched quote marks, wrong type of arguments, etc.
- **Rounding errors** arise from truncations due to hardware limitations, digitization of mathematical functions, converting between decimal and binary mathematics, etc.
- There is also a **fourth type**, namely errors made by Scilab system designers and programmers. They show up as bugs, performance limitations, poor user interfaces, and the like

Programming pitfalls (3/4): error messages

“Incompatible vector lengths”
would be a better error message

```
-->[1 2 3] + [4 5]
!--error 8
Inconsistent addition.
```

```
-->[1 2 3] * [4 5 6]
!--error 10
Inconsistent multiplication.
```

This message is misleading if what you intend is $[]' * []$, but ok if you aim at elementwise multiplication $[] .* []$ (but “Wrong multiplication” is better still)

```
-->sqrt = 5^2 + 3*17
Warning : redefining function: sqrt
```

. Use funcprot(0) to avoid this message

```
sqrt =
76.
```

Here you can see that the warning “redefining function” does have a meaning. I have improperly used `sqrt` as a variable name, but Scilab recognizes it is a built-in function. The answer is correct, but one should rather change the variable name. Check help name if you are uncertain if an intended variable name is reserved

Programming pitfalls (4/4): the endless loop

I have several times mentioned the risk of creating an endless loop, so let's look at this little beast

When you execute the script you have to **crash the program to stop it**. The easiest way is to press the Close button on the Console and then reload Scilab

Why does the loop not end?
Because we die from old age before the variable n by chance gets **exactly** the value 0.5



```
// endless_loop.sce
// Demonstrates an endless loop. /
// Execution ends only by crashing /
// the program (click on the Close /
// button (X) on the Console) /
n = .1;
dt = getdate();
rand('seed',1000*dt(9) + dt(10));
while n ~=0.5;
    n = rand(0,'normal');
end;
disp(n)
```

Have you forgotten about seeding rand functions? If so, go back to Ex 1-3 (lotto draw)

Debugging (1/2)

- We are already familiar with Scilab's rudimentary embedded debugger that provides error messages on the Console (a separate debugger window may come with Scilab 6.0)
- Another debugging tool is the pause, resume, abort set of statements. Read section 6.7 in *Introduction to Scilab* by Michaël Baudin for an explanation
- My suggestion for painless programming is **stepwise development**, meaning to
 - Develop the script inside out, starting with the central equation (or similar "kernel") and executing it using a simple plot or display command. Correct the "kernel" until it works to satisfaction
 - Extend the script stepwise by adding subroutines, loops, plot commands, handle commands, etc. and test (execute) after each added step
 - The advantage with stepwise development is that, first, bugs are isolated to a specific part of the script and easy to identify and ,second, one gets a feeling of satisfaction from each added bug-free step

Debugging (2/2): validation

- Finally, even when a script seems to behave correctly we must validate it. **Don't judge a bird by the colors of his feathers**
- To validate you can (among other things):
 - Take a critical look at the solution: is it logically **sound**, do you really know what the program **does**—and what it **does not** do?
 - Check for and eliminate redundancies (I have found surprisingly many in the textbook examples that I have borrowed)
 - Run it for some special cases for which you know the answer. If no model cases are available, check at least that the answers it provides are plausible and magnitudes are correct
 - Test for “unusual events” (e.g. where you could end up dividing by zero), extreme values (e.g. infinite), conditions leading to loop lockup, overlooked rounding errors, stack overruns, etc.
 - Work through the program by hand to see if you can spot where things could start going wrong
 - Ask somebody cleverer than yourself for a second opinion

Speeding up Scilab (1/4): introduction

- There are ways to speed up the execution of Scilab programs.
The three major rules are:
 - Replace loops by vectorized operations.* Particularly with the for loop one should aim at its vectorized alternative
 - Use subroutines whenever possible
 - Avoid time consuming algorithms like Runge-Kutta
- Speed-up—particularly if we move from loops to vectorized functions—requires that we adopt new thinking. It's a new learning effort. But vectors are, after all, what Scilab is all about!
- However, there is a problem with learning vectorized operations: Textbooks tell us to use them but pay little attention to the subject and their few examples are very basic

*) Scilab does not support Matlab's `vectorize()` function.

Speeding up Scilab (2/4): vector-based functions

- This case is adapted from Baudin. The task is to compute the sum of odd integers [1,99]
- In the first case we use a nested while...if...end...end structure, picking odd integers with the modulo() function



```
// add_demo1.sce  
  
clc;  
add = 0;  
i = 0;  
while ( i < 100 )  
    i = i + 1;  
    if ( modulo( i, 2 ) == 0 ) then  
        continue;  
    end  
    add = add + i;  
end  
disp( add )
```

2500.

- Below is the alternative vectorized solution. Clean and simple! Advantages:

- Higher level language, easier to understand
- Executes faster with large matrices



```
// add_demo2.sce  
  
clc;  
add = sum(1:2:100);  
disp(add)
```

2500.

There was a **bug** in Scilab 5.3.1 and it returned an “Invalid index” error message for the latter script

Speeding up Scilab (3/4): execution time tic()..toc()

- Execution time can be measured with the tic() and toc() pair of functions
- The top script computes values for sin(x) and orders the result in a table with two columns (shown to the far right for only four points). The execution time 17.389 s is for the shown script, with Scilab looping over 30,000 times
- The lower (vectorized) script performs the same task. The execution time is 9 msec, about 2000 times faster than with the for...end loop!

```
// measure_time1.sce
clear,clc;
x=[]; // Initiate vector
y=[]; // Ditto
tic(); // Start stopwatch
for t=0:0.0002:2*%pi
    x=[x; t];
    y=[y; sin(t)];
end
time=toc(); // Stop watch
disp(time) // Display time
```

0. 0.
2. 0.9092974
4. - 0.7568025
6. - 0.2794155
0.014

17.389

```
// measure_time2.sce
clear,clc;
tic();
t = (0:0.0002:2*%pi)';
[t,sin(t)]
disp(toc())
```

0.009

Speeding up Scilab (4/4): two more ideas

Replace loop by ones ():

```
tic();  
for i = 1:100000  
    x(i) = 1;  
end  
disp(toc())
```

77.481

```
tic();  
x = ones(100000,1);  
disp(toc())
```

0.005

Replace nested loops with
length (find ()):

```
tic();  
k = 0;  
for i = 1:1000000  
    x = rand(1,1);  
    if x < 0.2 then  
        k = k + 1;  
    end  
end  
disp(['k =' string(k)])  
disp(['time =' string(toc())])
```

!k = 200660 !

!time = 10.142 !

In this case the execution time is reduced by a factor of 34. Not nearly as much as in the earlier cases, but still a significant improvement (typical in practice)

```
tic();  
k = length(find(rand(1000000,1) < 0.2));  
disp(['k =' string(k)])  
disp(['time =' string(toc())])
```

!k = 199649 !

!time = 0.298 !

Discrepancy in time measurements (1/2)

I wanted to check Scilab's computation time for a cased given in a textbook on Matlab by Hahn & Valentine. First I did it on the Console and then on the Editor, but the results did not match:

```
-->tic();  
  
-->s = 0;  
  
-->for n = 1:100000  
-->s = s + n;  
-->end  
  
-->time = toc();  
  
-->disp(time)
```

97.531

The result is 97.531 seconds on the Console. Clearly not true because the answer came up without delay

It is only 0.453 s when done on the Editor. That's more like it

```
// scilab-matlab_loop.sce  
  
clc;  
tic();  
s = 0;  
for n = 1:100000  
    s = s + n;  
end  
time = toc();  
disp(time)
```

0.453

Let's try with vectorized functions (next slide)

Discrepancy in time measurements (2/2)

And here is the same in vectorized form:

```
-->tic();  
-->n = 1:100000;  
-->s = sum(n);  
-->time = toc();  
-->disp(time)
```

32.994

Now the Console tells
of a threefold
improvement in
computation time,
but still not true...

and the Editor
agrees about the
improvement, but
the discrepancy
remains

```
// scilab-matlab_vectorized.sce  
clc;  
tic();  
n = 1:100000;  
s = sum(n);  
time = toc();  
disp(time)
```

0.016

Conclusion: There is a bug either in my approach or in Scilab; but Scilab seems to execute faster than Matlab on the old Pentium II processor that Hahn & Valentine used

ATOMS (1/6): installing new toolboxes

- Recall the **problems with ATOMS** that I mention in [Chapter 1](#)
- ATOMS (AutomaTic mOdule Management for Scilab) allows the user to download and install external toolboxes (modules)
- There is reason to take a look at which modules might be of use, since specialized tools can limit the time needed to solve a problem
- Start by hooking up the computer on the Internet and clicking on the ATOMS icon on the Console. If you are unlucky you will see the following message on the Console (I was unlucky and others have been as well):

```
atomsDownload: The following file hasn't been downloaded:
```

```
- URL          : 'http://atoms.scilab.org/5.3/TOOLBOXES/32/windows.gz'  
- Local location : 'C:\Users\Johnny\AppData\Local\Temp\SCI_TMP_2772_.atoms\1_TOOLBOXES.gz'
```

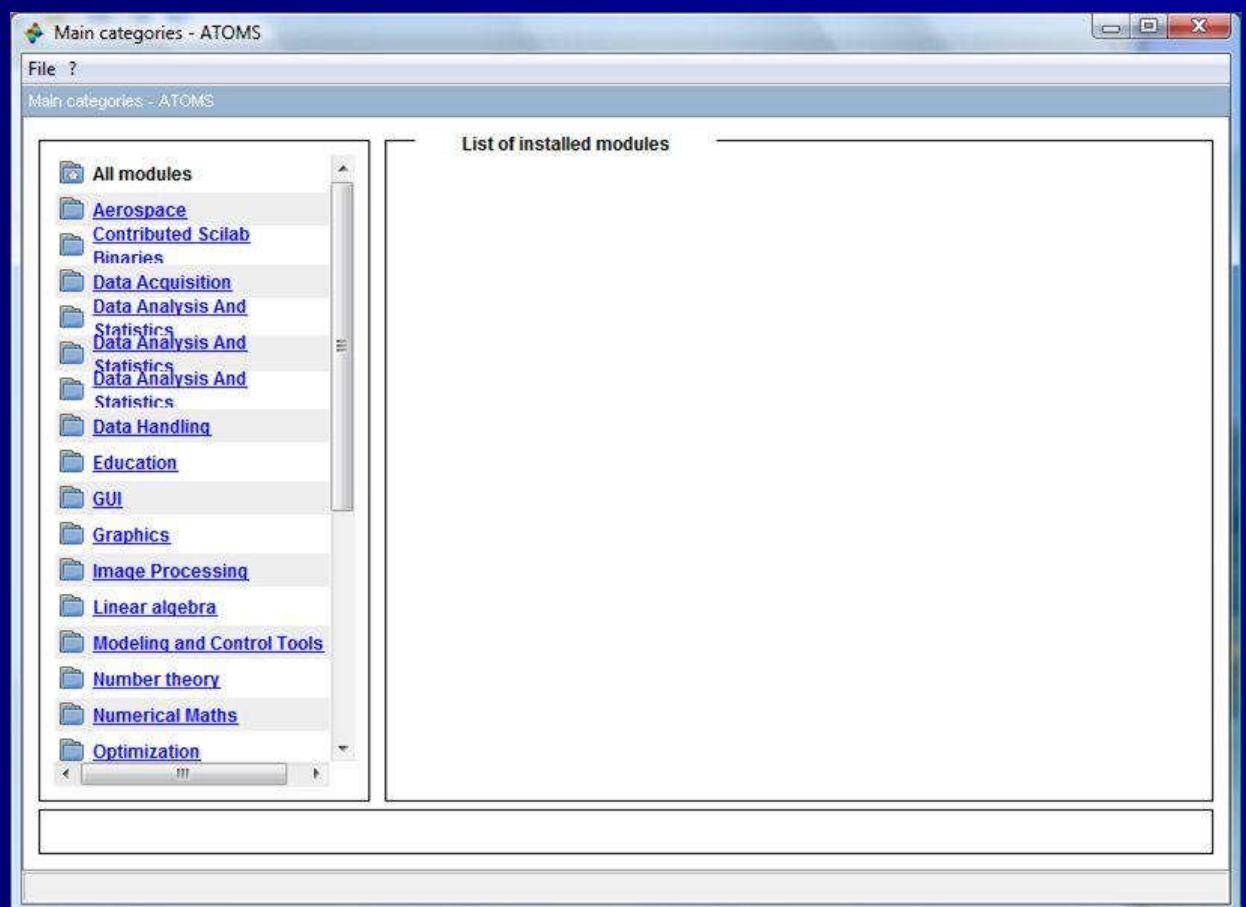
- The bug (Scilab **bug** #8942) remains unsolved and its true influence is unknown to me. The Scilab team gives the unhelpful suggestion to download the mentioned file

ATOMS (2/6): what's available

This is the ATOMS main window. Texts overlap a bit, but basically it is a list of contents

Go ahead and try to find something of interest, even if there isn't much for us engineers

Another problem is that there is **little information** about what the modules really can do for us

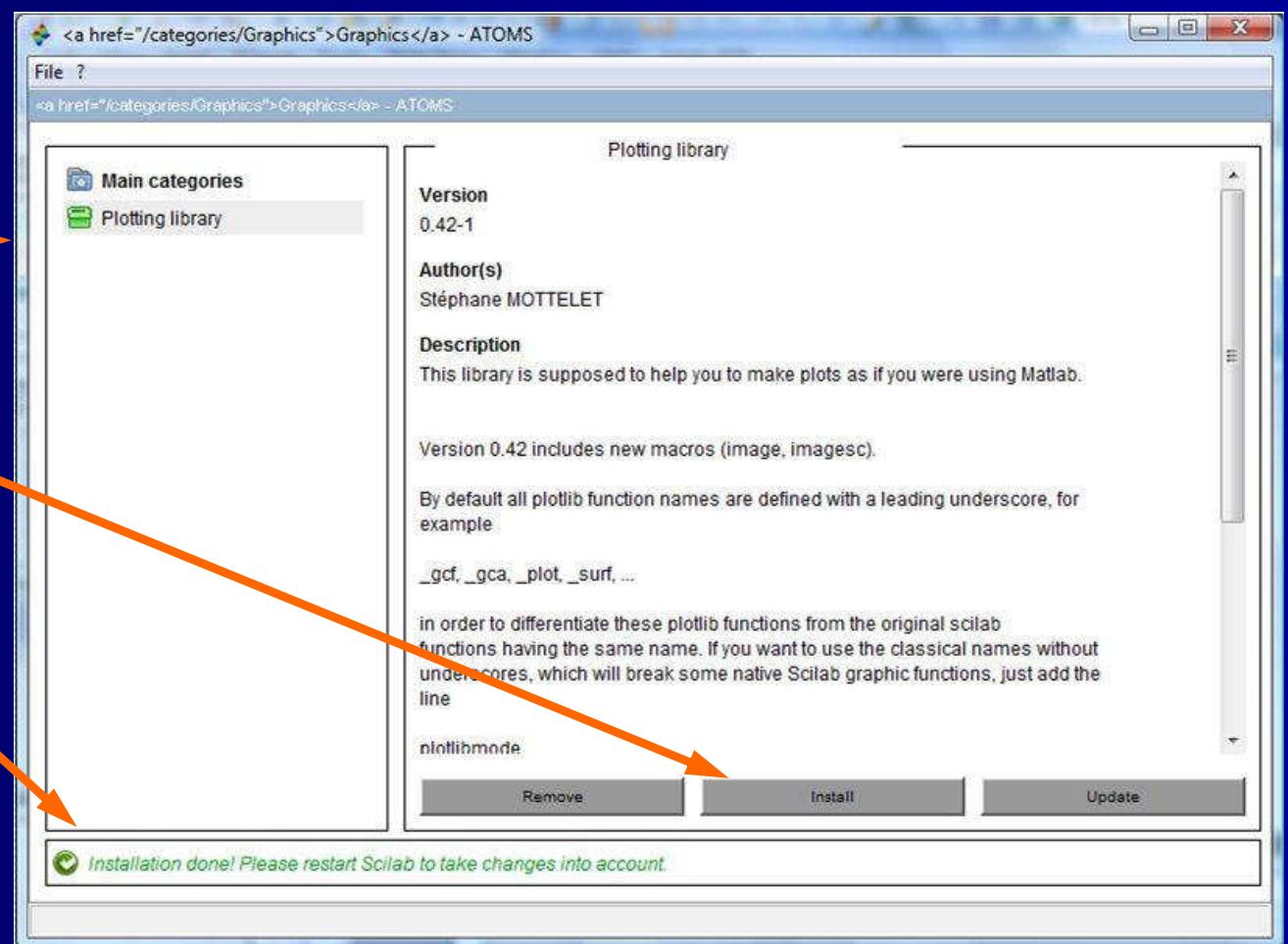


ATOMS (3/6): install

I decided to try
Stéphanie
Mottelett's
"Plotting library"
(this version
gives problems
with a Vista PC!)

Click on Install

An installation
message opens
at the bottom,
and after a good
while Scilab tells
that the module
has been
installed



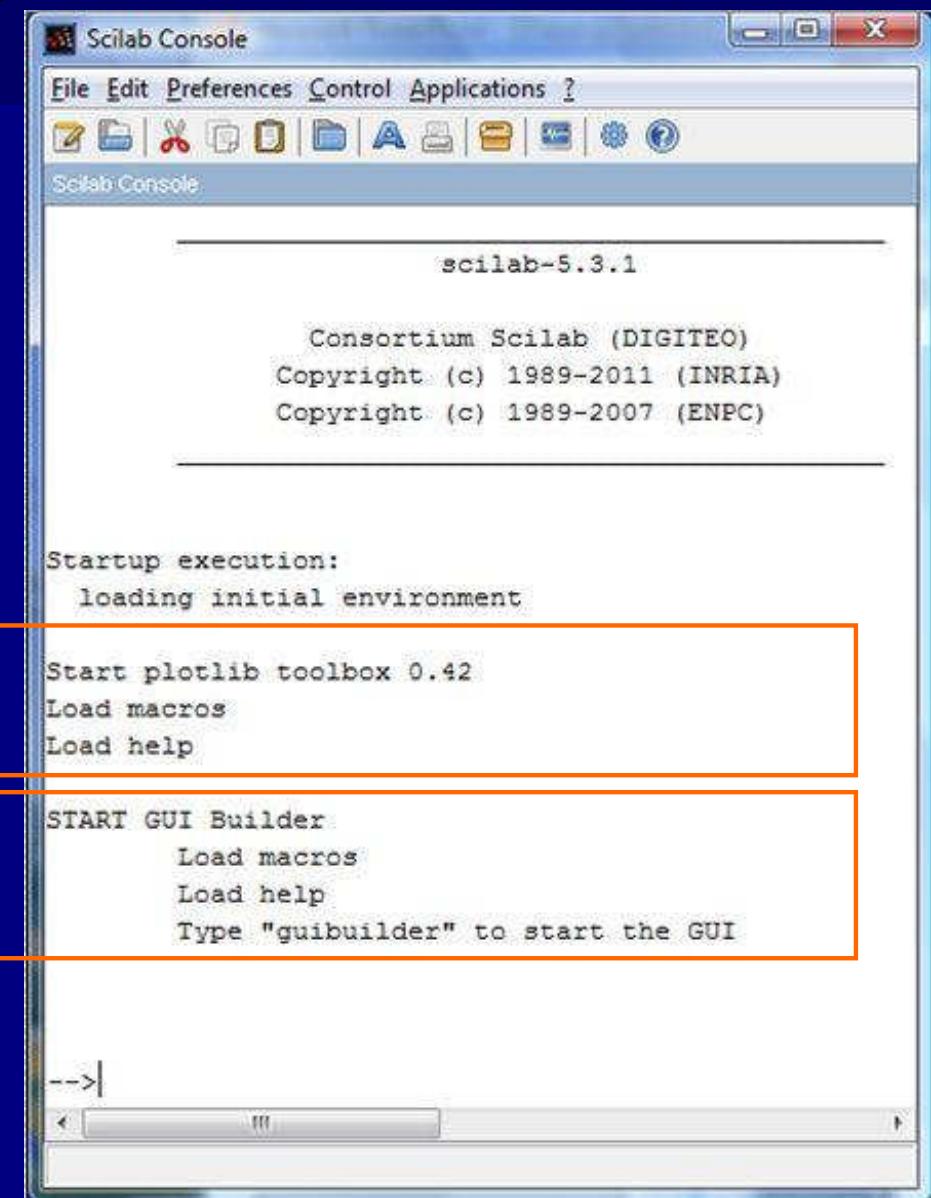
ATOMS (4/6): new info

I also installed the GUI Builder by TAN Chin Luh

When Scilab is restarted it informs about the installed toolboxes

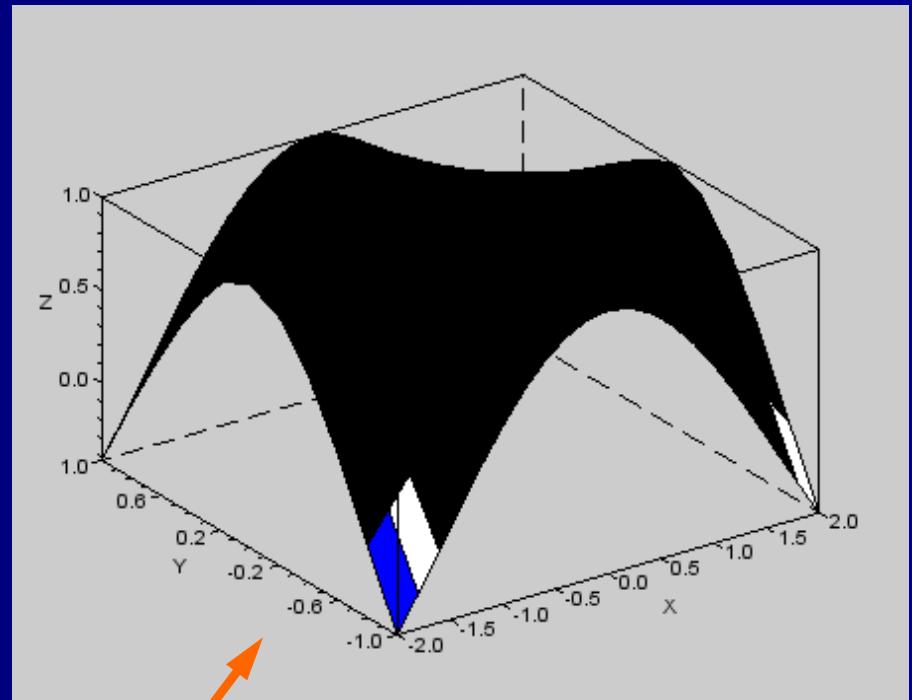
Question: What is needed to make use of the installed modules?

Check with the Help Browser and yes, at the very end of the list of contents are new additions: "Matlab-like plotting library" and "A Graphic User Interface Builder"



ATOMS (5/6): Check with Matlab's quiver3()

```
// matlab_quiver3.sce  
  
// Test Matlab's quiver3() function /  
  
clear,clc,clf;  
  
[X,Y]=meshgrid(-2:0.2:2,-1:0.2:1);  
Z=cos(X.*Y);  
surf(X,Y,Z);  
hold on  
[U,V,W] = surfnorm(X,Y,Z);  
quiver3(X,Y,Z,U,V,W,'r');  
Legend 'Surface normals'  
colormap gray  
hold off
```



I tested Matlab's quiver3() function with the script in Help, but **something is wrong**. The plot is wrong and Scilab yells about hold on, which it should know by now

hold on
!--error 4
Undefined variable: hold

at line 10 of exec file called by :
es\matlab_quiver3.sce', -1

ATOMS (6/6): discussion of problems

- My troubles began in earnest with the execution of Matlab's quiver3() function. No matter what I did, all Scilab scripts turned out garbage plots
- The situation was confused by simultaneous events: Apart from toolboxes I had also installed Scilab 5.3.2, had some Windows updates arriving, and saw hiccups with both MS Word and Windows. There was no problem with Scilab 5.1.1
- Windows had been running for three years so I decided to reinstall it. Only after this process I suspected ATOMS
- To cut the story short, the problems were due to the Plotlib toolbox. I uninstalled it and Scilab 5.3.2 worked normally again
- **Lessons learned:** Install only one toolbox at a time and test it and Scilab immediately. Uninstall the toolbox in case problems emerge



Building a script library

- Over time we accumulate a huge number of programs. How should we administer them, how can we later find what we need?
- This presentation demonstrates alternative ways of **commenting scripts**—a most important subject when a program has to be modified in the future
- Pay attention to program names. Descriptive names help to identify individual programs among other programs in a large file
- Build your library of Scilab scripts in a logical manner. In this work I have partly saved scripts on a thumb stick drive, in the file H:\Dr.EW\Writings\Scilab examples\, under the assumption that this presentation points to where to find a particular script. This is not a the way to do it continuously, so give your own **documentation system** a thought—including the backup solution!
- One option is to maintain a **spreadsheet catalogue** of programs with information on what a specific script does, where it is located, which functions it contains, etc. An advanced solution is documentation software of the type used in requirements management

19. Examples, Set 6

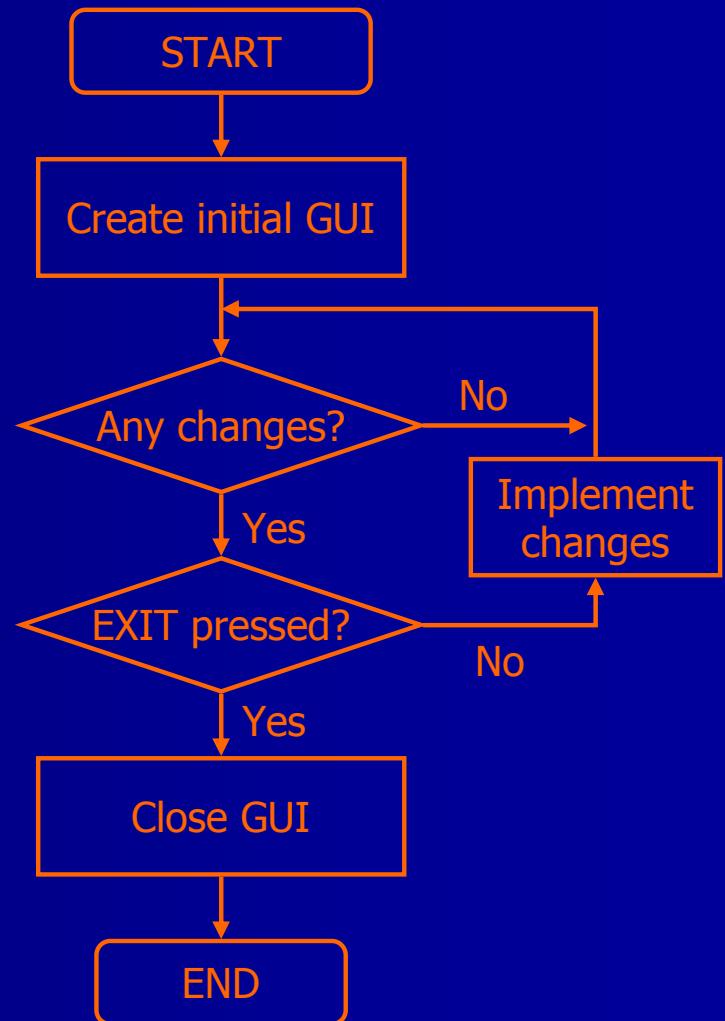
Additional examples, mostly
related to Chapters 15-19



[Return to Contents](#)

Example 6-1: user defined GUI, introduction

- This example is a modification of the similar in Antonelli, Chiaverini: *Introduzione a Scilab 5.3*, pp. 74-80
- The task is to create a GUI in the Graphics Window (GW). The GUI consists of
 - A sine plot
 - A slider for changing the angular frequency of the plot
 - Two “radiobuttons” by which the properties of the plotted graph can be changed
 - An exit button that closes the GUI
- The process is shown as a flow diagram to the right



Ex 6-1: user defined GUI, script (1/6)

Go to the MAIN
program below if you
want to proceed in a
logical order

The first subroutine,
initial_GUI(), creates
the initial sine plot
within the Graphics
Window; including
title and axes labels

The initial angular
frequency ω is defined
as 5 Hz

There is really nothing
special here

6) *italian.sce*

```
// Generates on the Graphics Window (GW) a GUI that contains a sine plot (plot2d), a slider by which to adjust the angular frequency of the sine function, two radiobuttons that change the style and color of the sine graph, and a pushbutton that closes the GW

clear,clc;

// **** SUBROUTINES **** //

// Declaration of initial plot in GUI:
//-----
function initial_GUI()
t = linspace(0,7,200);
w = 5;
plot2d(t,sin(w.*t),.. // Initial angular frequency
rect = [0,-1.1,7,1.1]); // Initial plot w=5 rad/s
a = gca();
a.axes_bounds = [0.2,0,.8,1]; // Frame dimensions & location
xtitle("GUI DEMO WITH sin (wt)",...
"Time [s]","Amplitude");
a.font_size = 3; // Axes mark size
a.x_label.font_size = 3; // x_label size
a.y_label.font_size = 3; // y_label size
a.title.font_size = 3; // Title size
endfunction
```

Ex 6-1: ... script/...

The next two subroutines respond to user commands (slider and radiobuttons respectively), and point to the fourt subroutine, new_GUI_data()

An existing plot is erased

The slider goes from end to end in 10 steps

The if-then-else-end constructs register the status of whichever radiobutton has been clicked

```
// Functions for changes wrt user actions:  
//-----  
function update_slider()  
    new_GUI_data();  
endfunction  
  
function update_radio()  
    new_GUI_data();  
endfunction  
  
// Redefine plot in GUI:  
//-----  
function new_GUI_data()  
    t = linspace(0,7,200)  
    drawlater();  
    a = gca();  
    if (a.children~=[]) then  
        delete(a.children);  
    end  
    w = h_slider.value/10;  
    plot2d(t,sin(w.*t));  
    if (h_radio1.value == 0) then  
        a.children.children.polyline_style=1; // Basic style: line  
    else  
        a.children.children.polyline_style=3; // IF clicked: bars  
    end  
    if h_radio2.value==0 then  
        a.children.children.foreground=1; // Basic color: black  
    else  
        a.children.children.foreground=2; // IF clicked: blue  
    end  
    drawnow();  
endfunction
```

// IF slider movement
// GOTO new_GUI_data()

// IF radiobutton click
// GOTO new_GUI_data()

// Delay changes

// IF frame contains graph...
// then delete graph

// Slider range: 10 steps

// Check status of style button
// Basic style: line

// IF clicked: bars

// Check status of color button
// Basic color: black

// IF clicked: blue

Ex 6-1: user defined GUI, script/...

The Main program first deletes an existing GW

The size and location of the new GW is defined as a function of the total screen size

Here we pick up the initial plot that comes in the GW (GUI)

The next thing that we add to the GUI is the EXIT button. Note how many arguments the uicontrol() function has

```
// **** MAIN **** //  
  
xdel();  
funcprot(0);  
  
// Define window size & position:  
//-----  
screen_size = get(0,"screensize_px"); // Find computer screen size  
size_x = .7*screen_size(3); // .7*screensize_px 3rd element  
size_y = .7*screen_size(4); // .7*screensize_px 4th element  
h_graph = scf(0); // Open Graphics Window  
h_graph.figure_size = [size_x size_y]; // Define GW size  
h_graph.figure_position =... // Position GW in the...  
[size_x/5 size_y/6]; // middle of the screen  
  
// Open GUI with initial plot:  
//-----  
initial_GUI();  
  
// Add EXIT button:  
//-----  
h_stop = uicontrol (h_graph,...  
"style","pushbutton",...  
"string","EXIT",...  
"fontsize",14,...  
"backgroundColor",[1 0 0],... // Red button RGB  
"foregroundColor",[1 1 1],... // White label RGB  
"position", [85 size_y-210 50 50],...  
"callback","xdel(0)"); // CLOSE GW if button pushed
```

Ex 6-1: user defined GUI, script (4/6)

Here is the `uicontrol()` command that controls the slider

`strcat()` is a function that we have not met before. Note that `w` and `rad/s` are surrounded by **double asterisks** ('' w '' and '' rad/s ''), not by quotation marks (strings within a string)

This is the initial label below the slider

And the `uicontrol()` that takes care of label changes

```
// Add slider & label:  
//-----  
h_slider= uicontrol(h_graph,...  
    "style","slider",...  
    "Min",0,...  
    "Max",100,...  
    "value",50,...  
    "position",[10 size_y-270 180 20],... // Slider size & location  
    "callback","update_slider();... // GOTO to update_slider()  
foo = strcat(['' w = '' string(h_slider.value/10)...  
    '' rad/s '')];h_text_slider.string = foo");  
slidelbl = strcat(["w = 5 rad/s"]); // Define initial label  
h_text_slider = uicontrol(h_graph,...  
    "style","text",...  
    "horizontalalignment","center",...  
    "string",slidelbl,...  
    "fontsize",14,...  
    "backgroundColor",[1 1 1],... // White background  
    "position",[10 size_y-310 180 20]); // Field size & location
```

Ex 6-1: user defined GUI, script (5/6)

The first radiobutton controls the style of the plotted sine curve (a solid line is the default, turns to a bar graph when the radiobutton is clicked)

The commands are quite similar to the ones of the slider, except that the foo command is missing

```
// Add radiobutton for graph style:  
//-----  
h_radio1 = uicontrol(h_graph,...  
    "style","radiobutton",...  
    "Min",0,...  
    "Max",1,...  
    "value",0,...  
    "backgroundColor",[1 1 1],...  
    "position",[10 size_y-350 20 20],...  
    "callback","update_radio()");  
h_text_radio1 = uicontrol(h_graph,...  
    "style","text",...  
    "horizontalalignment","left",...  
    "string",-Change graph style",...  
    "backgroundColor", [.8 .8 .8],...  
    "fontsize",14,...  
    "position",[40 size_y-350 140 25]);
```

// Declare radiobutton
// Initial button value
// GOTO to update_radio()
// Declare button text
// Gray background
// Field size & location

Notice the “callback” statements. They are the beasts that make us jump up to (GOTO) the subroutine in case (here to update_radio())

Ex 6-1: user defined GUI, script (6/6)

The second radiobutton controls the color of the plotted sine curve (black is the default, turns to blue when radiobutton is clicked)

This is mostly a repetition of the commands for the first radiobutton, but the position in the GW is different

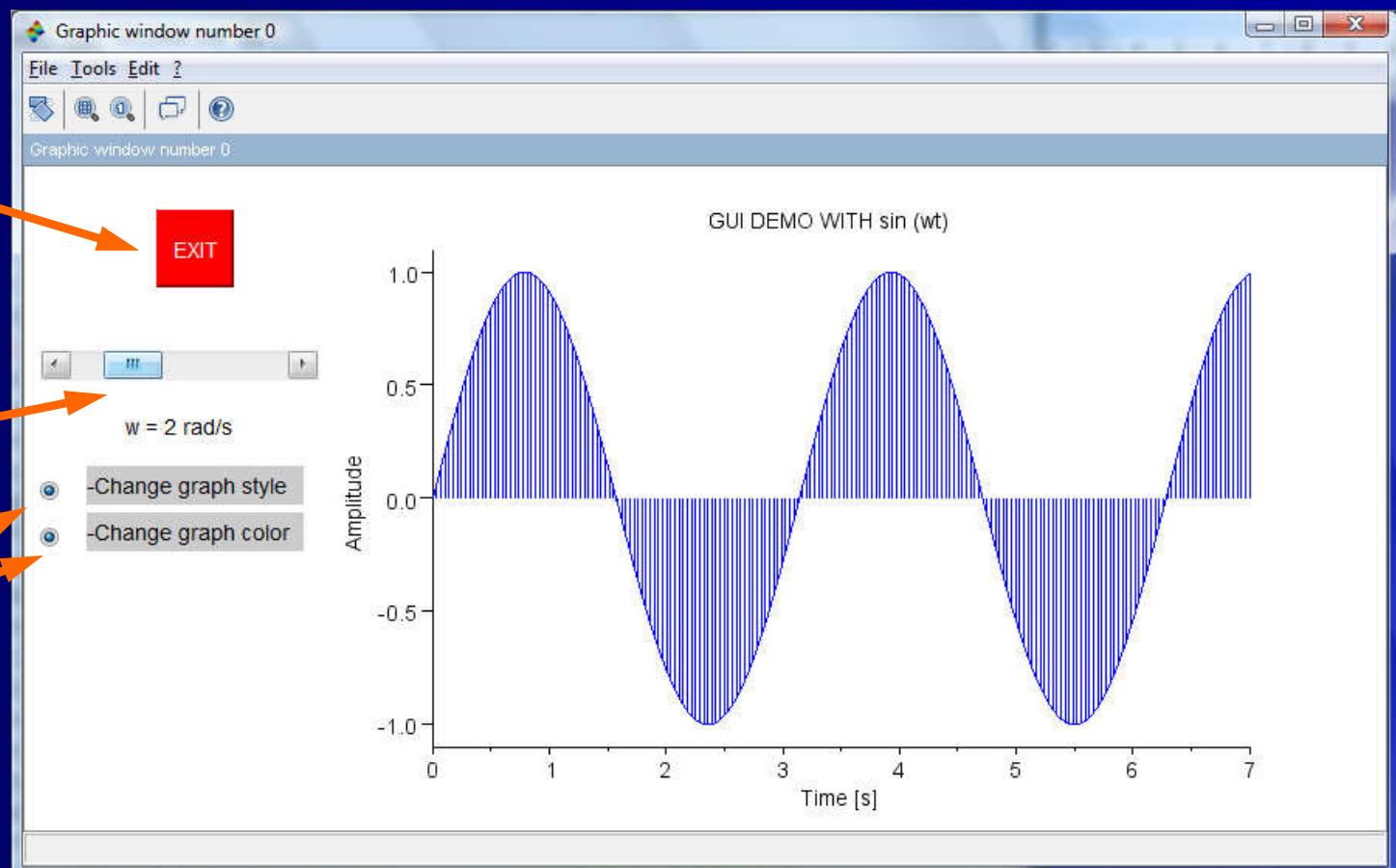
```
// Add radiobutton for graph color:  
//-----  
h_radio2 = uicontrol(h_graph,...  
    "style","radiobutton",...  
    "Min",0,...  
    "Max",1,...  
    "value",0,...  
    "backgroundColor",[1 1 1],...  
    "position",[10 size_y-380 20 20],...  
    "callback","update_radio()");  
h_radio2_txt = uicontrol(h_graph,...  
    "style","text",...  
    "horizontalalignment","left",...  
    "string",-Change graph color",...  
    "backgroundColor", [.8 .8 .8],...  
    "fontsize",14,...  
    "position",[40 size_y-380 140 25]);  
  
// **** END MAIN **** //
```

Ex 6-1: user defined GUI, and here it is

Click on
EXIT and
the window
closes

w clicked
down to 2
rad/s

Both
radiobuttons
have been
clicked



Problem: Scilab experiences a **lockup** if you drag the slider. The plot freezes and the Consol reports that the current handle no longer exists

Ex 6-1: discussion

- I copy-pasted the script from Antonelli & Chiaverini into Scilab's Editor
- The script had to be cleaned up and some redundancies could be removed
- I added the second radiobutton and organized the script in what I thought was a more logical fashion
- When I executed the script it opened up as expected, but the slider was missing
- After a frustrating weekend I did the whole thing from the beginning, but now in steps. The error was that I had moved the if–then–end construct in function new_GUI_data() after the plot2d() command
- **Lessons learned:** Do the job stepwise and test as you progress
- As for the lockup, my guess is that Scilab runs into a conflict situation when it should update the handle and the previous update still is in progress

Example 6-2: animation of a waltzing polygon (1/4)

- This demo is based on Pinçon’s “Eine Einführung in Scilab”
- The original contained errors, obsolete functions, and redundant commands. For instance, I transformed xset() functions to handle graphics commands (as explained in Chapter 7)

```
// animation_pincon_m2.sce  
//-----  
// The script plots the track of a blue polygon (rectangle)  
// with red border, as it turns around its axis while racing  
// counterclockwise in a circular loop on a black background.  
// The rectangle can be changed to a trapezoid or other shape  
// by changing element values in the matrix polygon. Changing  
// theta arguments in the matrix align gives different effects  
//-----  
clear,clc,clf;  
  
// Basic parameters:  
//-----  
steps = 100;           // Steps per circular loop  
blength = 0.6;         // Basic length of polygon  
width = 0.3;          // Basic width of polygon  
radius = 0.6;          // Radius of circular loop  
revolutions = 1;        // Number of loops to run
```

Ex 6-2: animation of a waltzing polygon (2/4)

- The matrix polygon defines length & width of edges. Change them to different values and the rectangle is modified to a different polygon
- Note the use of the %inf constant to fill missing arguments in plot2d()
- h=gca() declares h as a handle
- The handle is first used to set the background color

```
// Basic equations & definitions:  
//-----  
t = linspace(0,revolutions*2*pi,steps);  
x_axis = radius*cos(t); // x-axis of circular loop  
y_axis = radius*sin(t); // y-axis of circular loop  
polygon = [-length/2 length/2 length/2 -length/2;...  
           -width/2 -width/2 width/2 width/2];  
// Defines corners of polygon  
  
// Set scale for isometric plot:  
//-----  
plot2d(%inf,%inf,frameflag=3, rect=[-1,-1,1,1], axesflag=0)  
h = gca();  
xtitle('Waltzing polygon')  
h.background = 1; // Set background to black
```

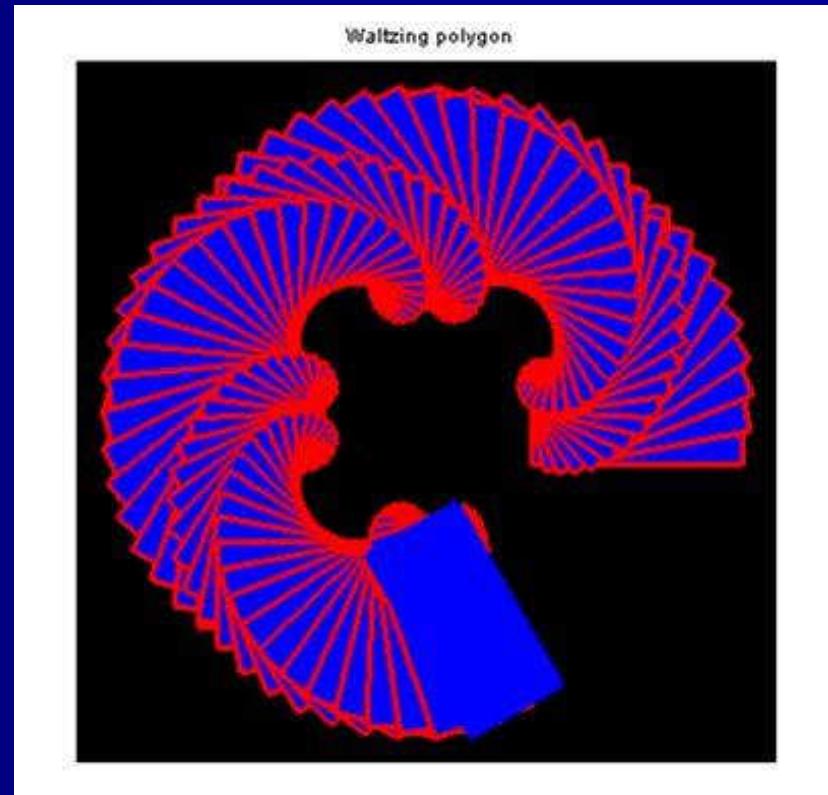
Ex 6-2: animation of a waltzing polygon (3/4)

- The matrix align turns the polygon into a new angle. Change theta values to see some interesting effects
- Here the handle is used to set the fill color of the polygon; the process is executed by xfpoly()
- Next the handle defines the border color; in this case the executing function is xpoly()

```
// Plot rectangle as it waltzes its loop:  
//-----  
turns = 3;           // Number of turns per loop  
for i=1:steps  
    theta = turns*t(i); // Angle of polygon alignment  
    align = [cos(theta) -sin(theta);...  
             sin(theta) cos(theta)]*polygon;  
    // Realigns polygon  
    h.foreground = 2;   // Set fill color to red  
    xfpoly(align(1,:)+x_axis(i), align(2,:)+y_axis(i))  
    // Fills polygon with defined color  
    h.foreground = 5;   // Change to blue for border  
    h.thickness = 3;   // Set border thickness to 3  
    xpoly(align(1,:)+x_axis(i), align(2,:)+y_axis(i),'lines',1)  
    // Draws polygon border in defined color  
end
```

Ex 6-2: animation of a waltzing polygon (4/4)

In this screenshot the polygon (rectangle) has made just over three quarters of its counterclockwise loop. At the same time it has spun $2\frac{1}{4}$ times around its axis, and has begun the last $\frac{3}{4}$ turn. There are 100 position samples on a full loop (`steps = 100;`) and it completes in a few seconds

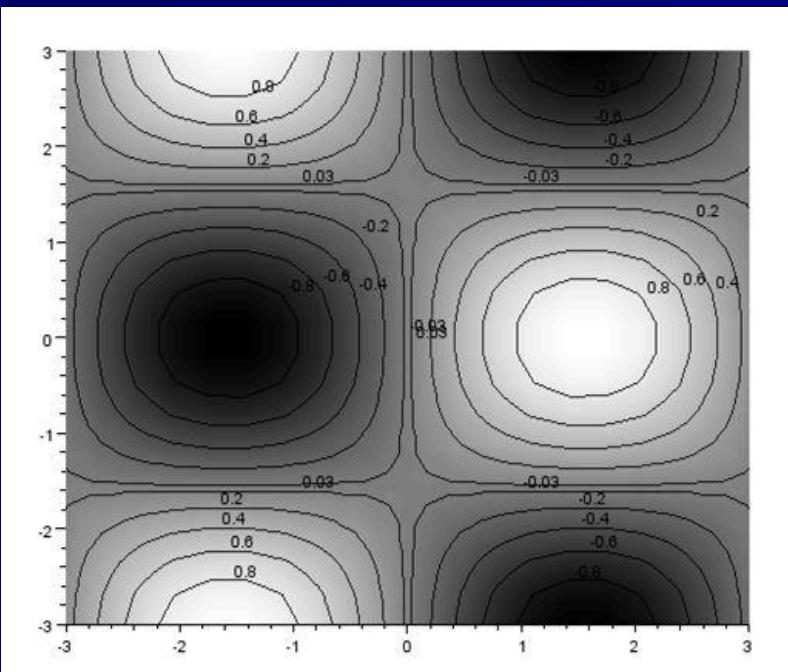


Example 6-3 (1/2) : grayplot() & contour2d()

- This example shows how a gray color scale and contour lines can be combined to create the illusion of a 3D space
- `linspace()` is multiplied by a 1×3 vector since the color map (the “third dimension”) must be a $m \times 3$ matrix. The color map can be inverted with $(1 - \text{linspace}())$ and a nonlinear amplitude function can be added to stress the effects
- The `Sgrayplot()` function smoothes the plot color compared with the basic `grayplot()` function
- Contour lines are added

```
// grayplot_demo.sce /  
  
// Gray area map with level curves using /  
// grayplot() / Sgrayplot() & contour2d() to /  
// create illusion of a 3D space /  
  
clear,clc,clf();  
  
// Color map definitions & initial declarations:  
//-----  
f = gcf();  
f.color_map = linspace(0,1,64)*ones(1,3);  
n = 20; // Plot resolution  
x = linspace(-3,3,n); // 3D plot limits  
y = x;  
  
// Plot function:  
//-----  
Z = sin(x)*cos(y); // Function to plot  
Sgrayplot(x,y,Z) // Smoothed grayplot  
  
// Define and add level curves:  
//-----  
level = [-.8 -.6 -.4 -.2 -.03 .03 .2 .4 .6 .8];  
contour2d(x,y,Z,level);
```

Ex 6-3 (2/2): grayplot() & contour2d()



The influence of the sine and cosine functions are easy to see (note that the origin is in the center of the graph)

The contour lines become white if the color map is inversed

Steps begin to show in the gray scale if the color map definition is changed to `linspace(0,1,32)`, where the argument 32 stands for halved color resolution

Change the plot function from `Sgrayplot()` to `grayplot()`, and you'll see the meaning of the variable `n=20`

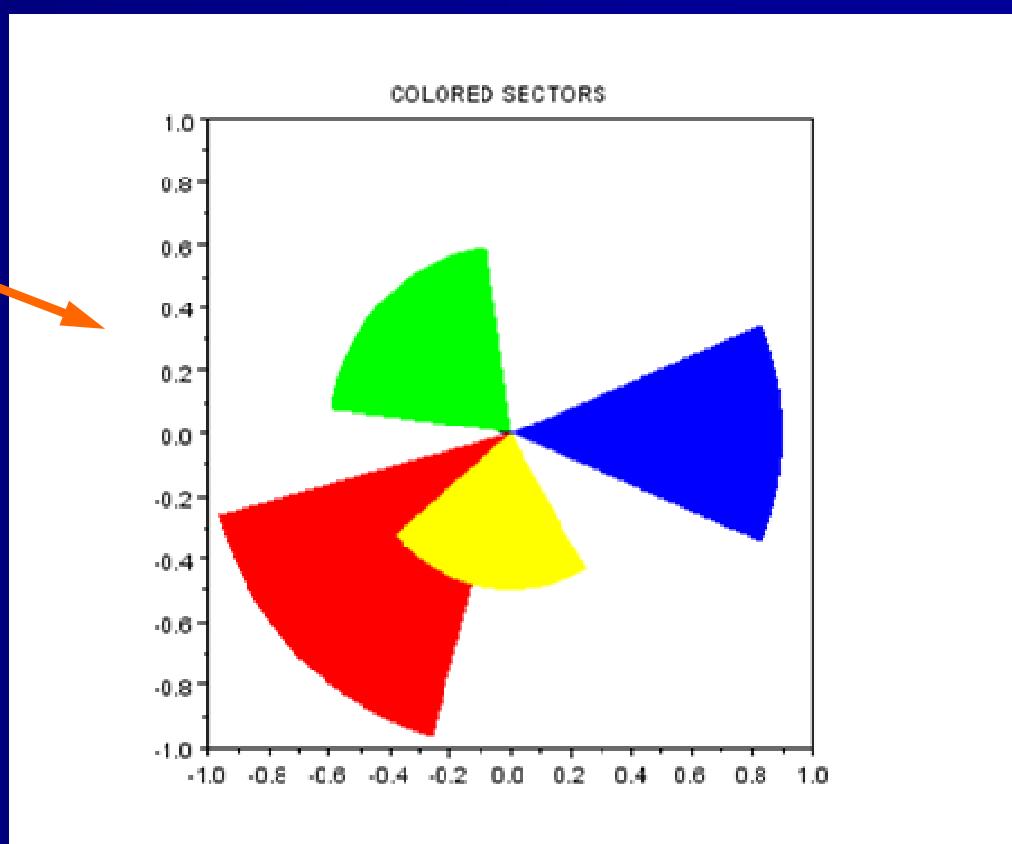
Example 6-4: sector chart, script

- This script is based on a solution by Pierre Lando and shows a method for creating a sector chart, with each sector having defined length (radius), direction, width, and color
- The solution can be seen as a more general case of Scilab's pie() function that we met in Chapter 9
- The most important function in this case is xfarc(), which we already met in the first animation demo (the arcs vector is of course also important since it governs the whole plot)

```
// Plots four colored sectors in predefined directions and with predefined widths /  
// directions and with predefined widths /  
  
clear,clc,clf;  
  
// ----- SUBROUTINE ----- /  
// The plot2d() function defines the figure, /  
// xfarc() adds colored sectors to the plot /  
  
function create_sectors(r, angle, width, col)  
    plot2d(%nan,%nan,-1,"031"," ",[-1,-1,1,1])  
    arcs=[-r;r;2*r;2*r;(angle-width/2)*64;width*64];  
    xfarc(arcs,col)                                // Add sectors  
    xtitle('COLORED SECTORS')  
endfunction  
  
// ----- MAIN ----- /  
// Define sectors:  
//-----  
rad = [.9,.6,1,.5]                                // Sector radii  
angle = [0,135,225,270]                            // Sector midpoints  
width = [45,75,60,80]                              // Sector widths  
colors = [2,3,5,7]                                  // Color definitions  
  
// Call subroutine:  
//-----  
create_sectors(rad,angle,width,colors)  
  
// ----- END MAIN ----- /
```

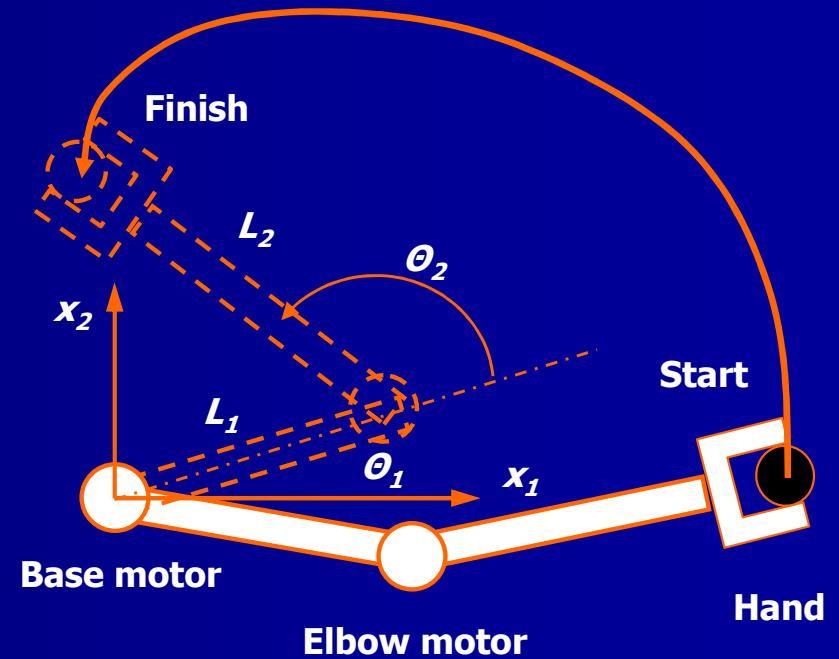
Ex 6-4: sector chart, plot

- Here is the plot, nice and beautiful. We can suppress the axes by changing the `plot2d()` argument '031' to '030'
- The overlapping yellow sector just happens to be on top (last element in the vector list). In practical applications, when doing automatic plotting of some process, we would have to put more effort into getting the plot the way we want



Example 6-5: Robot arm (1/6), introduction

- Recent web discussions on the relative virtues of Scilab, Matlab, and Octave made me take a new look at **manual conversion of Matlab scripts**
- This case with a two-dimensional moving robot arm is from Chapman, S.J.: *Matlab Programming for Engineers*, 2nd ed., (publisher & year unknown), pp. 202-206
- The case demonstrates practical application of matrices. See Chapman for a full discussion
- The original script can be found on the next slides; the converted script with added comments are on the following two slides



Ex 6-5: Robot arm (2/6), Matlab script

```
% Robot arm motion script
%
% Initial values, angles in degrees
tf = 2;
theta10 = -19*pi/180;
theta1tf = 43*pi/180;
theta20 = 44*pi/180;
theta2tf = 151*pi/180;
%
% Equations for a coefficients
T = [ tf^5      tf^4      tf^3
      5*tf^4   4*tf^3   3*tf^2
      20*tf^3  12*tf^2  6*tf ];
c = [ theta1tf-theta10; 0; 0 ];
disp('Coefficients for theta1 motion:');
a = T\c
%
% Equations for b coefficients
d = [ theta2tf-theta20; 0; 0 ];
disp('Coefficients for theta2 motion:')
```

```
b= T\d
%
% Equations of motion
L1 = 4;
L2 = 3;
t = linspace(0,2,401);
tq = [ t.^5; t.^4; t.^3 ];
theta1 = theta10 + a'*tq;
theta2 = theta20 + b'*tq;
x1 = L1*cos(theta1) + L2*cos(theta1 + theta2);
x2 = L1*sin(theta1) + L2*sin(theta1 + theta2);
%
% Plot path of hand
plot(x1,x2),...
xlabel('x_1'),...
ylabel('x_2'),...
title('Path of robot hand'),...
text(4.3,0,'t=0s: (x_1,x_2) = (6.5,0)'),...
text(0.2,2,'t=2s: (x_1,x_2) = (0,2)')
```

Ex 6-5: Robot arm (3/6), Scilab conversion (1/2)

The joint motors are controlled by the following polynomial expressions:

$$\Theta_1(t) = \Theta_1(0) + a_1 t^5 + a_2 t^4 + a_3 t^3 + a_4 t^2 + a_5 t$$

$$\Theta_2(t) = \Theta_2(0) + b_1 t^5 + b_2 t^4 + b_3 t^3 + b_4 t^2 + b_5 t$$

Matrix equations are set up and solved for coefficient vectors (**a**, **b**), using given initial values $\Theta(0)$ and final values $\Theta(t_f)$, and the results are used to plot the path of the robot hand

// *robot_motion.sce*

// *Robot arm motion in two dimensions using a fifth-degree polynomial to control the motion. See Chapman, S.J.: / "Matlab programming for Engineers," 2nd ed., for a / detailed discussion. /*

clear;clc,clf;

// **Initial values, angles in degrees:**

tf = 2; // **Finish time**
theta10 = -19*%pi/180; // **Theta 1 start position**
theta1tf = 43*%pi/180; // **Theta 1 final position**
theta20 = 44*%pi/180; // **Theta 2 start position**
theta2tf = 151*%pi/180; // **Theta 2 final position**

// **Equations for a coefficients (velocity constraints have been taken into account):**

T = [tf^5 tf^4 tf^3
 5*tf^4 4*tf^3 3*tf^2 // **Angular velocity**
 20*tf^3 12*tf^2 6*tf]; // **Angular acceleration**
c = [theta1tf - theta10; 0; 0]; // **Theta 1 movement**
a = T\c // **Coefficient vector a**
disp(['Coefficients for theta1 motion:'])
disp([string(a')])

Ex 6-5: Robot arm (4/6), Scilab conversion (2/2)

By requiring that velocity and acceleration at t=0 be zero, the polynomial coefficients a_5 and a_4 become zero. This limits the size of the T matrix (previous slide) to 3x3

The computed coefficient vectors **a** and **b** are used to define angular speeds, based upon which the hand position is defined in x_1 and x_2 coordinates

// Equations for b coefficients:

```
//-----  
d = [ theta2tf - theta20; 0; 0 ]; // Theta 2 movement  
b= T\d // Coefficient vector b  
disp(['Coefficients for theta2 motion:'])  
disp([string(b')])
```

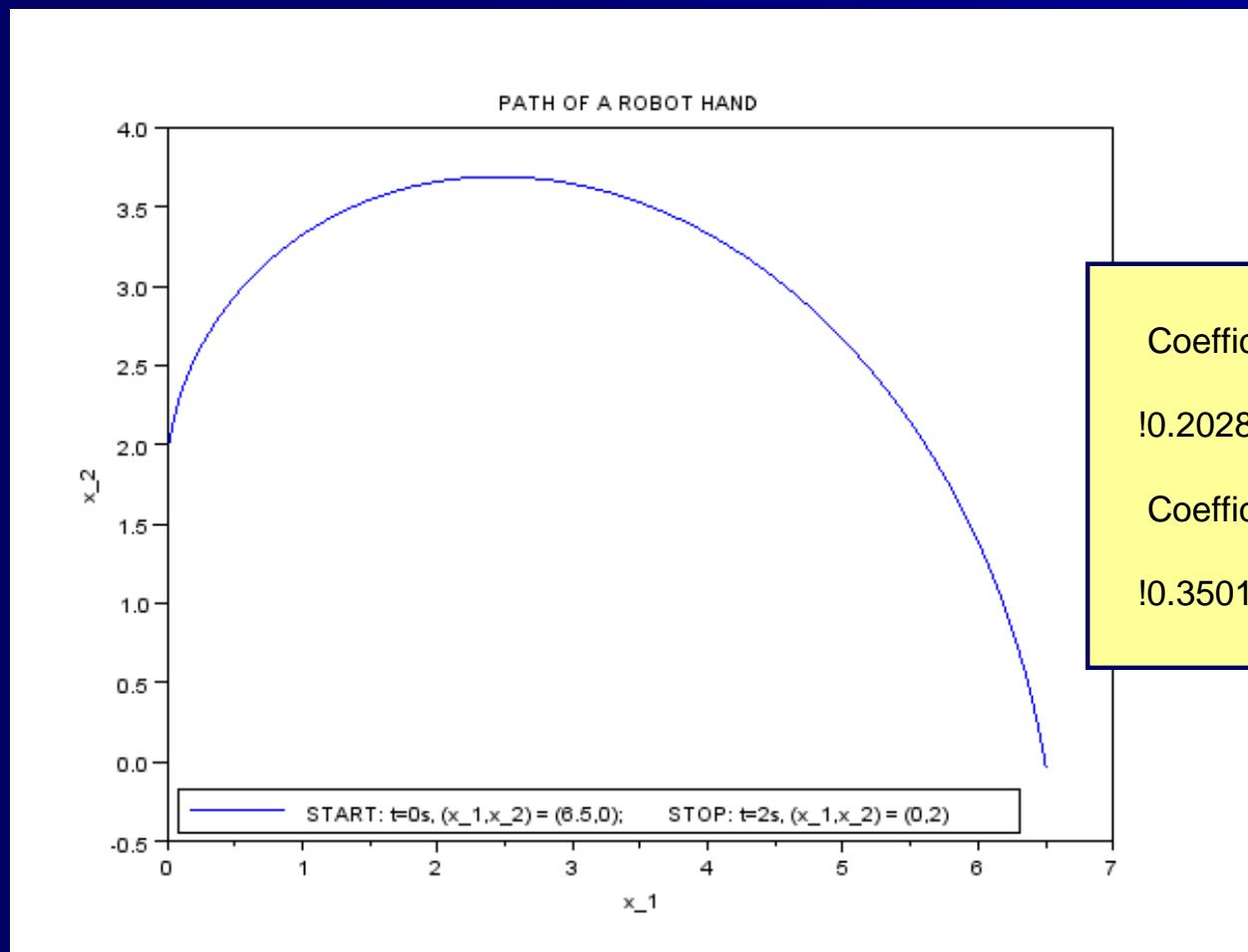
// Equations of motion:

```
//-----  
L1 = 4; // Length of upper arm [feet]  
L2 = 3; // Length of lower arm [feet]  
t = linspace(0, 2, 401); // Computation steps  
tq = [ t.^5; t.^4; t.^3 ];  
theta1 = theta10 + a'*tq; // Base motor angular speed  
theta2 = theta20 + b'*tq; // Elbow motor angular speed  
x1 = L1*cos(theta1) + L2*cos(theta1 + theta2); // x1 position  
x2 = L1*sin(theta1) + L2*sin(theta1 + theta2); // x2 position
```

// Plot path of hand, add labels & legend:

```
//-----  
plot(x1,x2),..  
 xlabel('x_1'),..  
 ylabel('x_2'),..  
 title('PATH OF A ROBOT HAND'),..  
 h1 = legend(['START: t=0s, (x_1,x_2) = (6.5,0); ..  
 STOP: t=2s, (x_1,x_2) = (0,2)'], 3)
```

Ex 6-5: Robot arm (5/6), plot & display



Coefficients for theta1 motion:
!0.2028945 -1.0144726 1.3526302 !

Coefficients for theta2 motion:
!0.3501567 -1.7507834 2.3343779 !

Ex 6-5: Robot arm (6/6), discussion

The manual conversion from Matlab scripts to Scilab was simple enough. Only the following came up:

- Matlab's % comment marks had to be changed to //
- Matlab's built-in pi function had to be changed to %pi
- Apostrophes (quotation marks) had to be rewritten, but only because copy-pasting gives the wrong type (' instead of)
- The disp() command had to be changed because Scilab does not output the a and b coefficients even if respective lines ($a=T\backslash d$ and $b=T\backslash d$) end without a semicolon (a **bug?**)
- Matlab's text() command is not recognized by Scilab (cf. Chapter 9). It allows legend beginnings to be located at precise points
 - The Help Browser does not give an answer on what to do
 - The Scilab-for-Matlab-users compendium by Beil & Grimm-Strele mentions this particular case but does not offer a solution
 - Conclusion: We have to stick to Scilab's ordinary legend commands

Example 6-6: animation with planet & moon, intro

- The task is to animate a planet with a moon rotating around it. If possible, the bodies should have different colors
- The task has its first difficulty in finding a way to keep the planet static while the moon rotates. My solution is to redraw both bodies for each step that the moon moves. Slow, but it works
- The second difficulty is to give the bodies different colors. The handle command `color_map` is nice, but it operates on the Figure level and only one color is possible for graphs on the Axes level. The presented solution is not perfect, since only the edges of the facets that form the spheres have different colors (this can be done on the Entity level)
- The third problem is with box alignment. It will be discussed on the plot slide

Ex 6-6: planet & moon, script (1/3)

- The spheres (planet, moon) are built from rectangular facets. The values of the facets are computed here, in subroutine facet()
- Basic variables for the planet

```
// planet_moon1.sce

// Animation with a moon rotating around a planet. /
// The spheres are composed of 3D X, Y, and Z   /
// facets using the surf() function to plot.      /

/clear,clc,clf;

// **** SUBROUTINE **** //
// Attach defined points to the spheres:
function [x, y, z] = facet(v, h)
    x = cos(v)*cos(h);                      // Facet x-matrix
    y = cos(v)*sin(h);                      // Facet y-matrix
    z = sin(v)*ones(h);                     // Facet z-matrix
endfunction

// **** MAIN **** //
// Define planet & moon variables:
//-----
// Planet (p), 10x10 degree grid:
vp = linspace(-%pi/2,%pi/2,18); // 18 steps vertically
hp = linspace(0,2*%pi,36);     // 36 steps horizontally
rp = 2;                        // Planet radius
```

Ex 6-6: planet & moon, script (2/3)

- Basic variables for the moon, both for the moon itself and its location in space
- GO TO subroutine facet() to compute facet matrices
- Basic plot definitions

```
// Moon (m), 20x20 degree grid & offset from origin:  
vm = linspace(-%pi/2,%pi/2,9); // 9 steps vertically  
hm = linspace(0,2*%pi,18); // 18 steps horizontally  
rm = 0.3; // Moon radius  
Rm = 2.1; // Moon offset  
Az = 0; // Moon start point  
n = 1; // # of moon revolutions  
step = 100 // # of steps/revolution  
  
// Define facets for spheres using subroutine facet():  
//-----  
[Xp,Yp,Zp] = facet(vp,hp); // Planet  
[Xm,Ym,Zm] = facet(vm,hm); // Moon  
  
// Plot commands (box, planet, moon):  
//-----  
// Define 3D box, put double buffer on, define surface:  
a = gca();  
a.data_bounds = [-5,-5,-3; 5,5,3]; // 3D box size  
f = gcf();  
f.pixmap = "on"; // Double buffer  
f.color_map = hotcolormap(32); // Surface color
```

Ex 6-6: planet & moon, script (3/3)

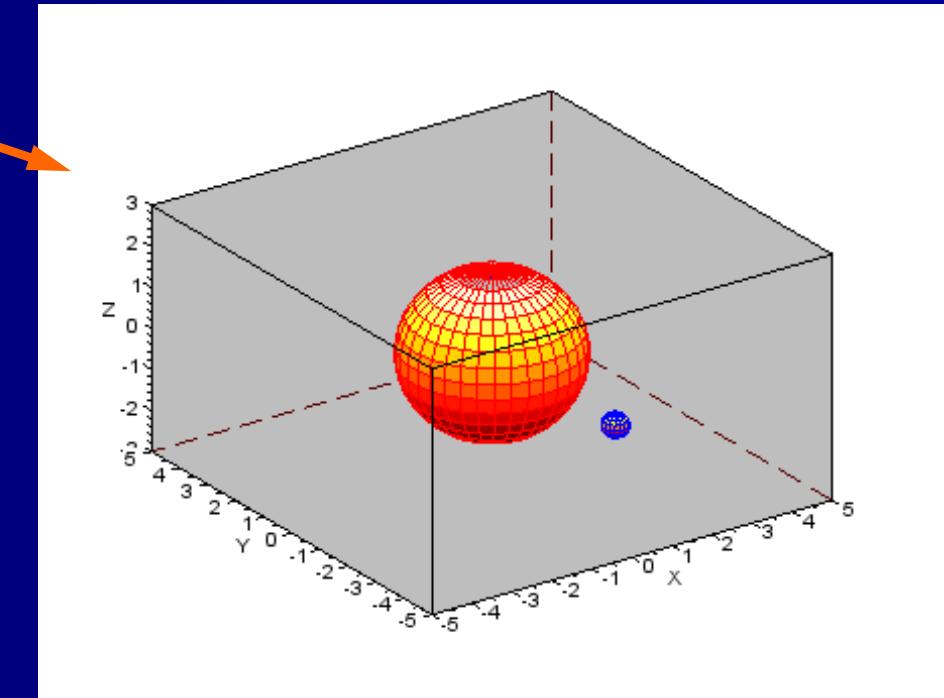
- Loop for rotation begins
- Delete old graphs
- Put color on the box
- Push planet data to first buffer
- Recalculate moon location & push data to first buffer
- `show_pixmap()` = push plot to screen

```
// Plot planet & rotating moon:  
for Az = 0 : 2*pi/step : n*2*pi  
  
// Delete previous entities (planet & moon):  
if (a.children~==[]) then // IF frame contains graph...  
    delete(a.children); // then delete graph  
end  
  
// Plot planet & define facet edges:  
a.background = color('grey'); // Box wall color  
surf(rp*Xp, rp*Yp, rp*Zp); // Plot planet  
e1 = gce();  
e1.foreground = color('red'); // Facet edge color  
  
// Plot moon & define facet edges:  
x_loc = Rm*sin(Az); // Location on x axis  
y_loc = Rm*cos(Az); // Location on y axis  
C = Rm*[x_loc, -y_loc, 0] // Moon center  
surf(C(1)+rm*Xm, C(2)+rm*Ym, C(3)+rm*Zm); // Plot moon  
e2 = gce();  
e2.foreground = color('blue'); // Facet edge color  
show_pixmap();  
end  
f.pixmap = "off";  
  
// **** END MAIN **** //
```

Ex 6-6: animation with planet & moon, plot

And here the beauty is. The moon rotates counterclockwise and is shown in its start position

As said above, this task was not without problems. One thing that I failed to do was to tilt the box somewhat differently. The handle command `a = gca();`
`a.rotation_angles =`
`[alpha,theta]` just refused to cooperate and the angles stayed at 51° and -125° respectively (a final **bug?**)



Measured with `tick(); ... tock()`, each moon step takes about 150 milliseconds to perform

20. Adieu

Final words to accompany you in
your struggle for survival of the
fittest



[Return to Contents](#)

That's it, Folks!

- We have reached the end of our journey. The road was longer and bumpier than I anticipated
- There is much more to Scilab but we are on our way if we master even this material (think of an office software package and how little of its potential you really know even if you use it daily)
- The most important next step is to do Scilab simulations on our own, to solve problems in our particular sphere of interest

Learning = hard brainwork + a tough rear end

- And for everybody's sake, keep reminding the Scilab team about the need for a comprehensive, up-to-date tutorial. To repeat an old engineering adage: **The job isn't done until the paperwork is done!**
- All the best and take care

JH

“When I think over what I have said, I envy dumb people.”
Seneca (4 B.C.—A.D. 65).