Lecture 1

Introduction

1 Course goals

Most of your engineering courses teach you theory and analytic methods using the techniques of mathematics (calculus, linear algebra, differential equations, and so on). The purpose is to train you to understand significant engineering problems from a fundamental theoretical perspective. You need this foundation to be able to design effective solutions to difficult problems – it's practically impossible to solve a problem you don't thoroughly understand.

Unfortunately, the set of engineering problems that can be solved analytically (with pencil, paper and even a calculator) is limited to relatively simple and idealized cases. Indeed, many "textbook problems" are carefully designed so that you can end up with the "exact answer." However, most applications in your engineering career will involve more complex systems for which you can mathematically formulate the problem (if you've learned your theory) but cannot solve the resulting equations analytically. People have long realized that in these cases numerical solutions are possible. Yet before the development of digital computers, when calculations had to be done by hand or slide rule, the application of "numerical computing" was fairly limited. As a result in the past engineers were forced to make many simplifying assumptions, build scale models, use trial and error and so on.

In the last several decades the development of computer technology has revolutionized engineering. Almost any problem that can be formulated mathematically can be solved using numerical techniques. Systems as complex as the supersonic flow of air through a jet engine or the operation of an integrated circuit containing millions of transistors are now routinely analyzed in minute detail using numerical methods implemented on computers.

This is the motivation behind "EE 221 Numerical Computing for Engineers." The purpose of this course is for you to:

1. learn fundamental numerical methods so you can understand how to formulate numerical solutions to difficult engineering problems, and
2. develop programing skills using Scilab/Matlab so you are able to effectively implement your numerical solutions.

The beginning of the course will be devoted to basic programing techniques, control structures, input/output, graphics and the like. For the remainder of the course we will use those skills to learn and implement numerical methods for important classes of problems.

2 Course topics

The main topics we will cover are listed below. Additional topics and programming project examples will be covered as time permits.

Scilab/Matlab Basics

- Introduction
- Arrays
- Programming structures
- Input and output
- 2D plots
- 3D plots and animation

Numerical methods

- Root finding
- Polynomials
- Linear algebra
- Linear systems
- Nonlinear systems
- Interpolation
- Optimization
- Curve fitting
- Numerical calculus
- Random numbers
- Matrix diagonalization

3 Course structure and grading

Your grade will be composed of the following components

- 20% Homework, typically twelve assignments in total, *no late homework accepted*. Your lowest score is dropped. I heavily weight the *effort* you put into your assignments. Assignments typically have a pencil and paper component to be turned in as hard copy, and a programming component to be submitted electronically via the course website.
- 80% Exams, 40% for each of two exams. The exams will include programming problems to be done in the lab during the specified exam times as well as pencil-and-paper problems.

4 Scilab and Matlab

Computers basically do lots of relatively simple calculations really fast. Numerical methods are typically mathematical statements of algorithms that (more-or-less) solve analytically intractable problems by doing lots of adding, subtracting, multiplying and dividing along with logic operations to guide program flow and termination. Computing runs the gamut from hand-held calculators to massively parallel supercomputers. Calculators are fairly straight-forward, and you will use one regularly throughout your coursework. Supercomputers are often dedicated to specific types of problems for which highly optimized computer code, typically written in the C/C++ or Fortran languages and compiled, is under ongoing development by teams of researchers. In the middle lies the class of problems appropriate to a single PC.

It's a good idea to use a computing environment that best fits your problem. You want to get a solution in a reasonable time with a minimum of effort. Simple arithmetic or evaluation of elementary functions can readily be done with a $15 calculator. It would be a waste to boot up a computer and open a programing environment in this case. For larger problems requiring a computer there are many options. Spreadsheets are useful for many basic engineering situations. They are quite visual and have graphic “programing” environments that are fairly intuitive. Why write a computer program if you can solve the problem with a spreadsheet? At the “high end” you can use a compiled language such as C with lots of time devoted to optimizing your code for the utmost performance. In between is a large class of problems where you want the best of both approaches. This is the target for computing environments such as Scilab and Matlab.

Scilab and Matlab are numerical computing environments that are extremely useful for the type of calculations performed by engineers. This environment combines some of the best features of a graphing calculator and a traditional computer programming environment. You will have an opportunity to use this tool extensively in your engineering coursework and most likely in your career.

Matlab (www.mathworks.com) is a commercial product, and expensive. Although the student version is only about $50 (but expires when you are no longer a student), a single “regular” license is more than $2,000. On the other hand, Scilab is a free, open-source tool (www.scilab.org). They have quite a bit in common. If you learn one you can transfer that knowledge to the other very easily. In this course we will emphasize Scilab. At the same time, I will mostly emphasize those aspects of Scilab that are identical or very similar to Matlab so that the conversion is essentially transparent. Where there are differences I will point those out.

4.1 History

(From http://en.wikipedia.org/wiki/Matlab)

> Short for "matrix laboratory", MATLAB was invented in the late 1970s by Cleve Moler, then chairman of the computer science department at the University of New Mexico. He designed it to give his students access to LINPACK and EISPACK without having to learn Fortran. It soon spread to other universities and found a strong audience within the applied mathematics community. Jack Little, an engineer, was exposed to it during a visit Moler made to Stanford University in 1983. Recognizing its commercial potential, he joined with Moler and Steve Bangert. They rewrote MATLAB in C and founded The MathWorks in 1984 to continue its development.

(From http://en.wikipedia.org/wiki/Scilab)

> Scilab was created in 1990 by researchers from INRIA and École nationale des ponts et chaussées (ENPC). It was initially named Ψlab[12] (Psilab). The Scilab Consortium was formed in May 2003 to broaden contributions and promote Scilab as worldwide reference software in academia and industry.[13] In July 2008, in order to improve the technology transfer, the Scilab Consortium joined the Digiteo Foundation. […] Since July 2012, Scilab is developed and published by Scilab Enterprises.

There are other free/open-source Matlab alternatives in addition to Scilab. One of the most widely used is GNU Octave. In some ways Octave is arguably even more Matlab compatible than Scilab. However, it's primarily written for Linux systems as is not as “clean” to install on Windows PCs or Macs. The Python programing language has been gaining ground in the numerical computing community, but differs significantly from Matlab. For these reasons I prefer Scilab as the “best” free/open-source Matlab alternative.

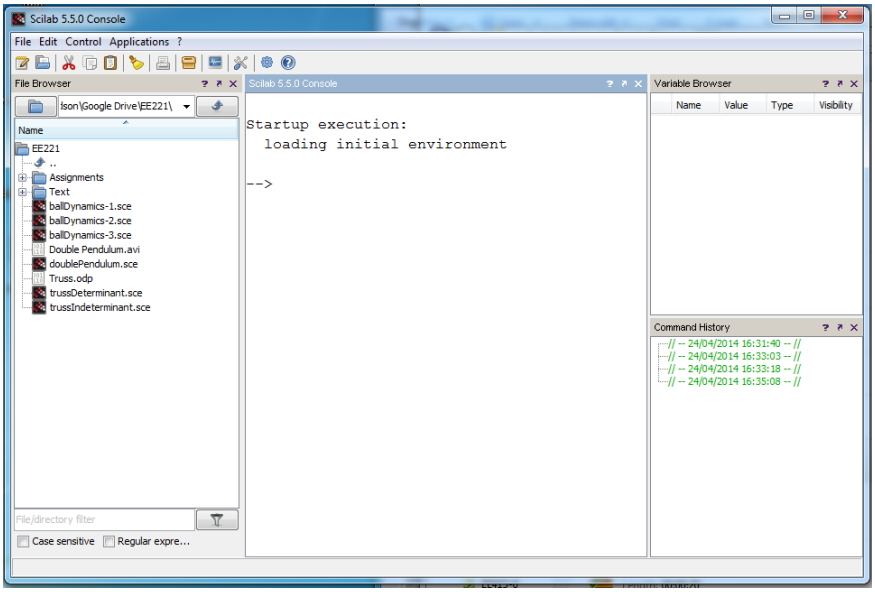
5 Installing and running Scilab

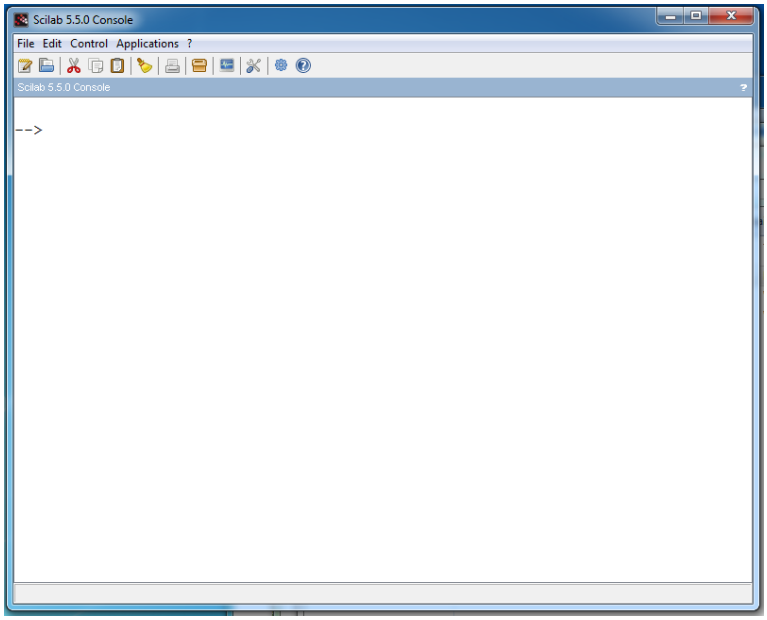
Scilab can be downloaded from www.scilab.org. There are executable versions for Windows, Mac and Linux. You can also download the complete source code. On Windows simply download the installation program (scilab-5.5.0_x64.exe is about 128MByte). Double click and agree to the license to install. The default on all options is recommended. When installation is complete you should have a Scilab icon on your desktop that looks some like this

There should also be a Scilab entry in the Start Menu. When you start Scilab you should see a window something like that shown below.

In the middle is the “console” where we will do most of our work. The other windows can X'd out for now. We can open them later if need be. This leaves just the console.

Type “help” at the prompt, or use the “?” menu to open the help browser. From there you can search or browse the documentation. This is a good way to find out about all the capabilities of Scilab. In this class we will use only a small subset of these.





6 Interactive use

There are two basic ways to use Scilab/Matlab. As an interactive environment you can type a command directly into the console at the prompt and Scilab will print the results. Then type another command and so on. This is how we will start out using Scilab, as essentially a fancy calculator. Later we'll learn how to use the editor to write structured programs.

6.1 Arithmetic

From the command line you can enter arithmetic expressions involving the addition, subtraction, multiplication and division operators (+ - * /). For example

```
-->10+6/3-4*2
 ans  =
    4.
```

The `ans` variable stores the result of the last calculation and can be used in the next calculation. Here is the same series of calculations performed one step at a time.

```
-->10
 ans  =
    10.

-->ans+6/3
 ans  =
    12.

-->ans-4*2
 ans  =
    4.
```

Operator precedence rules apply. Operations are performed from left to right. Multiplication and division are done before addition and subtraction. These can be overridden with parentheses. For example

```
-->3+2/4
 ans  =

    3.5
```

first divides 2 by 4 and then adds 3 to get 3.5; the + operator appears first (from left to right) but multiplication has precedence over addition. However

```
-->(3+2)/4
 ans  =

    1.25
```

Here the parenthesis tell Scilab to perform the addition operation first `3+2=5`, followed by division `5/4=1.25`. I find it good practice in all but the simplest cases to use parentheses to make the order of operations explicit. It is perfectly fine to use redundant parenthesis if this aids the readability of an expression. For example the parenthesis in the expression

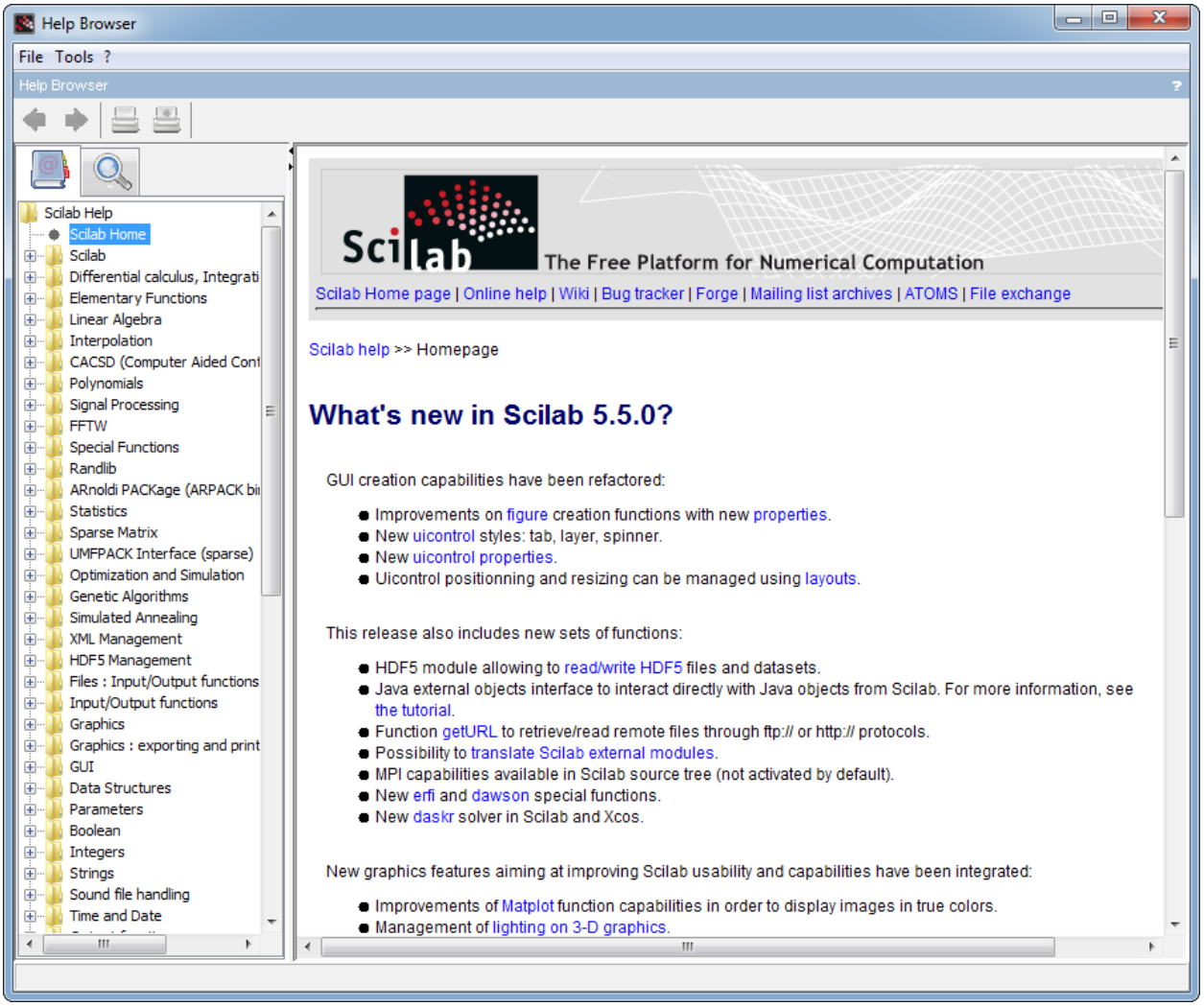
```
-->3+(2/4)
 ans  =

    3.5
```

have no effect, but you might find they make the order of operations explicit.

A very useful feature of Scilab/Matlab is that by pressing the up or down arrow keys (Ctrl-P and Ctrl-N) on your keyboard, you can cycle through the command line history. This allows you to easily repeat previous commands. The commands can also be edited using the arrow and Backspace keys.

Exercise 1: Use the console to calculate $1-1/2+1/3-1/4+1/5$



6.2 functions

Scilab/Matlab also has many built-in functions. The help menu provides complete listings. For example, the trig functions `sin, cos, tan` as well as their inverses `asin, acos,` and `atan` are available. You use parentheses to denote the argument.

```
-->sin(0.5)
 ans  =
    0.4794255

-->asin(ans)
 ans  =
    0.5
```

In Scilab the atan function is an "overloaded function" which allows you to provide different numbers of arguments. The expression `atan(z)` returns the angle between -π/2 and π/2 for which the tan function is `z`. It is a "two-quadrant" inverse tangent. The expression `atan(y,x)` returns the angle between -π and π for which the tan function is *y/x* and the angle corresponds to

the polar angle of the rectangular coordinates point `(x,y)`. In Matlab the same functionality is provided by `atan2(y,x)`.

The standard trig functions operate in radians. Thus `cos(1)` is the cosine of 1 radian. Since it is often useful to work in degrees, Scilab provides the functions `cosd, sind, atand` and so on where the appended "`d`" indicates degrees. So

```
-->atan(1)
 ans  =

    0.7853982

-->atand(1)
 ans  =

    45.
```

The power operation is performed by the ^ operator.

```
-->2^3
 ans  =
    8.
```

Negative and fractional powers are supported.

```
-->3^(-0.37)
 ans  =
    0.6659861
```

Square roots are evaluated with the sqrt(x) command.

```
-->sqrt(2)
 ans  =
    1.4142136
```

The exponential function e^x is implemented as exp(x).

```
-->exp(3.2)
 ans  =
    24.53253
```

The natural logarithm function $\ln(x)$ is, somewhat confusingly, implemented as log(x).

```
-->exp(3)
 ans  =
    20.085537

-->log(ans)
 ans  =
    3.
```

In many engineering books $\log(x)$ is used to refer to the base 10 logarithm. In Scilab/Matlab the base 10 logarithm is denoted by log10(x).

```
-->log10(2)
 ans  =
    0.30103

-->10^ans
 ans  =
    2.
```

This distinction has caused problems for many an engineering student because we are so used to using log(x) to denote the base-10 logarithm. You should pay particular attention to this.

> **Exercise** 2: In the console, calculate $\tan(0.5)$ and $\sin(0.5)/\cos(0.5)$. Calculate $\log_{10}(2)$, $\ln(2)$, $\ln(10)$ and $\ln(2)/\ln(10)$.

6.3 variables

You can define variables and then operate on those. For example,

```
-->x = 2
 x  =
    2.

-->x^2+3*x+7
 ans  =
17.
```

Typing in the variable name displays the variable value. If no such variable exists you'll get an error message.

```
-->x
 x  =
    2.

-->y
  !--error 4
undefined variable : y
```

6.4 strings

Variables are typically numerical in nature. However, variables can be assigned text values using quotation marks, as in the following examples.

```
-->a = 'this is a string'
 a  =
 this is a string

-->b = ' and here is some more text'
 b  =
  and here is some more text
```

Note that in Scilab you can also use double quotes as in "`this is a string`" but not in Matlab. Therefore *if you use single quotes in Scilab your code will be more Matlab compatible.*

You can concatenate two strings with the + operator in Scilab. In Matlab you can form an array or use the `strcat` function.

```
-->a+b //a+b in Scilab, in Matlab [a,b] or strcat(a,b)
 ans  =
 this is a string and here is some more text
```

String are particularly useful for labeling and describing numerical output.

6.5 constants (Scilab)

There are certain pre-defined constants built-in to Scilab. (Scilab calls these "permanent variables.") These are prefixed by the % symbol. For example,

```
-->%pi
 %pi  =
    3.1415927
```

The % prefix “protects” the constant and keeps you from redefining it.

```
-->%pi = 3
         !--error 13
redefining permanent variable
```

Many people find this notation messy but it has a very real advantage (see next section). If you don't like this you can simply create a regular variable with the same value, as in

```
-->pi = %pi
 pi  =
    3.1415927
```

6.6 constants (Matlab)

Matlab implements constants as pre-defined, but unprotected variables. These are not prefixed by the % symbol. For example,

```
>> pi
ans =
    3.1416
```

This is a weakness of Matlab, in my opinion, because you can accidentally redefine these, as in

```
>> pi = 3
pi =
     3
```

If you later use the variable pi thinking it's value is π you'll end up with erroneous results.

> **Exercise** 3: In the console, assign the value 0.25 to the variable y. Then calculate y^2-2y+3.

6.7 complex numbers (Scilab/Matlab with a wrinkle)

Complex numbers are created by making use of the imaginary unit i, as in math texts. For example $z=2+3i$ is a complex number with “real part” 2 and “imaginary part” 3. In Scilab the imaginary unit is the constant `%i`, as, for example, in

```
-->z = 2+3*%i
 z  =

    2. + 3.i
```

Note that you need to explicitly include the multiplication operator *. You cannot just juxtapose two variables to denote multiplication. If you don't like the % sign notation, you can define a variable to equal `%i` and use that instead. For example,

```
-->j = %i
 j  =
    i

-->z = 1+2*j
 z  =
    1. + 2.i
```

In Matlab the imaginary unit is the predefined variable i (also j). Again this is dangerous because you can redefine this variable accidentally. In fact i and j are very commonly used as index variables in for loops.

```
>> i
ans =
        0 + 1.0000i

>> i = 2
i =
     2
```

To avoid this problem Matlab defines `1i` as a protected variable equal to the imaginary unit

```
>> 2+3*1i

ans =

   2.0000 + 3.0000i
```

In Matlab you can leave out the * operator as in

```
-->3+2i
 ans  =
    3.0000 + 2.0000i
```

which is a nice feature that Scilab does not have. Scilab/Matlab functions can operate on and return complex numbers.

```
-->exp(%i*3)
 ans  =
  - 0.9899925 + 0.1411200i
```

6.8 semicolon notation

When you type in an expression Scilab/Matlab echoes the answer to the console. If you terminate the expression with a semicolon, nothing is echoed back. This is very useful if you are not interested in intermediate results and only want to see a final answer. For example,

```
-->x = 2;

-->y = 3;

-->z = tan(y/x);

-->w = sqrt(z)
 w  =
    3.7551857
```

Exercise 4: In the console, calculate $\sqrt{-4}$ and assign the value to variable z. Verify that $z^2=-4$. Set $u=2i$ and calculate $z-u$.

6.9 Numerical format

Scilab allows you some control over the default appearance of numerical output. Later we will see how to format displayed numbers in great detail. The format command allows you to select either “v” or “e” formats. For example

```
-->format('v');
-->10*%pi
 ans  =
    31.415927

-->format('e');
-->10*%pi
 ans  =
    3.142D+01
```

The "D+01" notation represents the exponent of 10 in scientific notation. "D" stands for "double precision" (more on that later). So, the last line above indicates a value $3.1412 \cdot 10^{1}$. You can also control the number of digits that Scilab outputs using an optional second argument, as in

```
-->format('v',5);
-->%pi
 %pi  =
    3.14
-->format('v',10);

-->%pi
 %pi  =
    3.1415927
```

In Matlab you can choose from various predefined formats such as

```
>> format short
>> pi
ans =
    3.1416

>> format long
>> pi
ans =
   3.141592653589793
```

See "help format" for more details.

7 Capturing an interactive session

Sometimes you want to save all the commands and results from an interactive Scilab/Matlab console session. The easiest way to do this is to use the Edit menu. First click "Edit => Select all" followed by "Edit => Copy" after which you can paste the results into Notepad or any other text editor.

A more "premeditated" approach is to use the `diary` command. This opens a file and echos all console output to the file.

```
-->diary('part1.txt')
 ans  =

    1.

-->str = 'everything I type should be saved'
 str  =

 everything I type should be saved

-->str = ' in file part1.txt'
```

```
 str  =

  in file part1.txt

-->x = 3
 x  =

    3.

-->sqrt(3)
 ans  =

    1.7320508

-->diary(0)
```

The file part1.txt then contains

```
-->str = 'everything I type should be saved'
 str  =

 everything I type should be saved

-->str = ' in file part1.txt'
 str  =

  in file part1.txt

-->x = 3
 x  =

    3.

-->sqrt(3)
 ans  =

    1.7320508

-->diary(0)
```

that is, everything in the interactive session after the `diary` command. The diary command also works in Matlab. The only difference is that the file closed with the command `diary('off')` rather than `diary(0)`.

8 Numerical limitations

If

$$x=\left(1+10^{-20}\right)-1$$

then what is x? Obviously $x=10^{-20}$. Let's verify this with Scilab (or Matlab)

```
-->x = (1+1e-20)-1
 x  =

    0.
```

Scilab says $x=0$, which is wrong. Why? Is 10^{-20} too small for Scilab to represent?

```
-->x = 1e-20
 x  =

    1.000D-20
```

Clearly that's not the problem. Instead we are seeing a *round-off error*. Scilab, like your calculator, is limited in the number of digits it can use to represent a number.

Suppose we represent numbers in scientific notation as

$$x = \pm d_1 . d_2 d_3 d_4 \cdot 10^{\pm e_1 e_2}$$

where $d_1, d_2, d_3, d_4, e_1, e_2$ are decimal digits (0-9). If $y = +1.000 \cdot 10^{+00}$ and $z = +1.000 \cdot 10^{-20}$ then $y + z$ is exactly

$$y + z = +1.00000000000000000001 \cdot 10^{+00}$$

However, in our format we are limited to four digits of accuracy. Therefore we have to "round off" the exact value as

$$y + z = +1.000 \cdot 10^{+00}$$

The tiny value 10^{-20} "fell off the end" so to speak. If we then calculate $x = (y + z) - 1$ we get $x = 0$.

Scilab stores numbers in "IEEE 754 double-precision binary floating-point format" using 64 bits or 8 bytes of computer memory. One bit accounts for the $\pm$ sign, eleven bits for the exponent and 52 bits for the fraction or "significand" or "mantissa." Since $2^{52} \approx 5 \cdot 10^{15}$ we can say that a double precision number has roughly 15-16 decimal digits of accuracy. If a calculation requires more digits of accuracy than this then it will not produce a correct result.

Now that's a lot of accuracy, but not enough to correctly calculate $(1 + 10^{-20}) - 1$. Unfortunately this can be a very practical problem for us when working with numerical methods. Consider the definition of the derivative of the function $f(x)$

$$\frac{df}{dx} \equiv \lim_{\delta \to 0} \frac{f(x + \delta) - f(x)}{\delta}$$

We might estimate this numerically as

$$\frac{df}{dx} \approx \frac{f(x + \delta) - f(x)}{\delta}$$

for δ "very small." Ideally the approximation should get better and better.

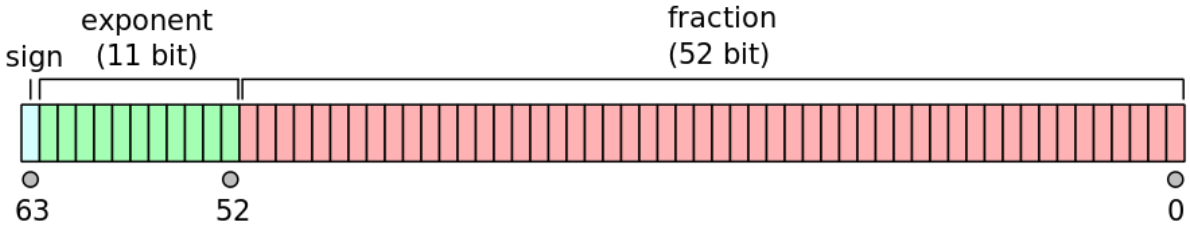


Fig. 1: IEEE double precision floating point format.

Let's test this with $f(x)=e^x$ for which $\left.\frac{df}{dx}\right|_{x=0}=e^0=1$.

```
-->delta = 1e-3;

-->(exp(delta)-exp(0))/delta
 ans  =
    1.000500166708385

-->delta = 1e-6;

-->(exp(delta)-exp(0))/delta
 ans  =
    1.000000499962184

-->delta = 1e-9;

-->(exp(delta)-exp(0))/delta
 ans  =

    1.000000082740371

-->delta = 1e-12;

-->(exp(delta)-exp(0))/delta
 ans  =

    1.000088900582341

-->delta = 1e-15;

-->(exp(delta)-exp(0))/delta
 ans  =

    1.110223024625157
```

Notice as we decrease δ, initially the approximation to the derivative improves. For $\delta=10^{-9}$ we get

$$\frac{df}{dx}\approx 1.000000082740371$$

which is accurate to about 8 digits. But further reduction of δ actually leads to worse accuracy. For $\delta=10^{-15}$ we have

$$\frac{df}{dx}\approx 1.110223024625157$$

which is not even accurate to 2 digits! In fact going to $\delta=10^{-20}$ results in

```
-->delta = 1e-20;

-->(exp(delta)-exp(0))/delta
 ans  =

    0.
```

which is completely wrong! The lesson is that we need to consider numerical limitations *very carefully* when we develop and implement numerical algorithms.

Exercise 5: $\frac{d}{dx}\sin(x)=\cos(x)$. Set $x=1$ and calculate $\cos(x)$. For $h=10^{-1},10^{-6},10^{-12},10^{-18}$ calculate $[\sin(x+h)-\sin(x)]/h$ and compare to $\cos(x)$.

9 References

Numerical methods books – used as fundamental references throughout these notes

1. Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T., *Numerical Recipes in C*, Cambridge, 1988, ISBN: 0-521-35465-X.
2. Recktenwald, G., *Numerical Methods with Matlab: Implementation and Application*, Prentice Hall, 2000, ISBN: 0-201-30860-6.
3. Heath, M. T., *Scientific Computing: An Introductory Survey*, McGraw Hill, 2002, ISBN: 0-07-239910-4.
4. Urroz, G. E., *Numerical and Statistical Methods with SCILAB for Science and Engineering Vol. 1,* BookSurge Publishing, 2001, ISBN-13: 978-1588983046.
5. http://www.mathworks.com/moler/

Software sites

1. http://www.scilab.org, official Scilab website
2. http://www.mathworks.com, official MatLab website
3. https://www.gnu.org/software/octave/, official GNU Octave website
4. https://www.python.org/, official Python website
5. http://www.gnu.org/software/gsl/, GNU Scientific Library

Wikipedia articles

1. http://en.wikipedia.org/wiki/Scilab
2. http://en.wikipedia.org/wiki/Matlab
3. http://en.wikipedia.org/wiki/GNU_Octave
4. http://en.wikipedia.org/wiki/Python_(programming_language)