OPEN SOURCE ENGINEERING

openeering

A SCILAB PROFESSIONAL PARTNER

powered by ENGIN SOFT

NUMERICAL ANALYSIS USING SCILAB: NUMERICAL STABILITY AND CONDITIONING

In this Scilab tutorial we provide a collection of implemented examples on numerical stability and conditioning.

Level 

www.openeering.com

Step 1: The purpose of this tutorial

The purpose of this tutorial is to provide a collection of Scilab examples that are typically part of numerical analysis courses.

Here we provide some classical examples on numerical stability and conditioning.

An introduction to

NUMERICAL ANALYSIS USING SCILAB

(numerical stability and conditioning)

Step 2: Roadmap

In this tutorial we provide the definitions of well-conditioned problem and stable algorithm and some related examples. These examples are useful to understand how much attention we have to pay when selecting a numerical solution method for a given problem.

Step 3: Conditioning of a problem

Numerical stability and conditioning are two concepts that should not be confused.

A problem is called ***well-conditioned*** if a small perturbation of the input data (relative error), leads to small variations of the results (relative error), i.e. of the same magnitude order.

On the other hand, an ***ill-conditioned*** problem strongly amplifies the input relative error in the output dataset.

Denoting by $\mathcal{P}$ the problem under consideration, if d represents the input data and r the output results, it is possible to define the ***condition number*** of the problem through the following inequality:

$$\frac{\|\Delta r\|}{\|r\|} \leq cond(\mathcal{P}) \cdot \frac{\|\Delta d\|}{\|d\|}$$

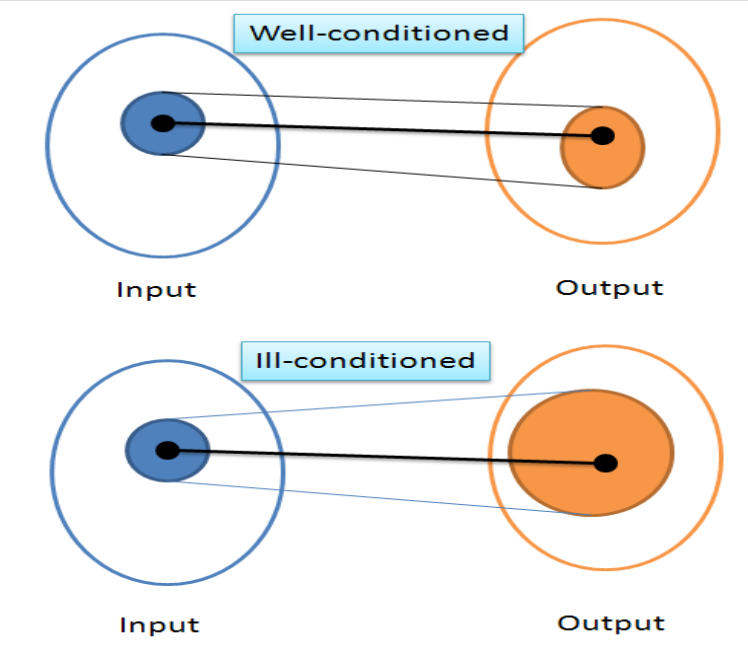
where $\|\cdot\|$ is a given norm which is able to measure the involved quantities. The condition number bounds the propagation of the input relative error in the output results, it is closely related to the maximum accuracy that can be attained in the solution. Whenever $r = 0$ or $d = 0$, the condition number is defined as

$$\|\Delta r\| \leq cond(\mathcal{P}) \cdot \|\Delta d\|.$$

Conditioning refers to the numerical problem and there is no connection with the rounding error and the solution strategy.

The numerical solution strategy should take into account the kind of problem we are trying to solve.

A graphical comparison is shown on the right.



(Well-conditioned versus ill-conditioned problem)

As an example, from the previous tutorial we have the following inequality:

$$\varepsilon_r^{\oplus}(x,y) \leq Eps + (1+Eps)\left(\frac{|x|}{|x+y|}\varepsilon_r(x) + \frac{|y|}{|x+y|}\varepsilon_r(y)\right)$$

That is:

$$\varepsilon_r^{\oplus}(x,y) \leq \max\{K_x, K_y\}\left(\varepsilon_r(x)+\varepsilon_r(y)\right)$$

where we have defined

- $K_x = \frac{|x|}{|x+y|}$ is the condition number relative to the input x;
- $K_y = \frac{|y|}{|x+y|}$ is the condition number relative to the input y.

From this inequality, when $x + y \to 0$ the condition numbers K_x and K_y grows up to infinity and hence the relative error on the result can be arbitrarily huge.

Step 4: Example: Intersection of two straight lines

In this example we consider the intersection of two straight lines r_1 and r_2 given by the equations:

$$\mathrm{r}_1: 1.0\,x + 3.5\,y = 8.0$$

and

$$\mathrm{r}_2: 2.1x + 7.0y = 16.1$$

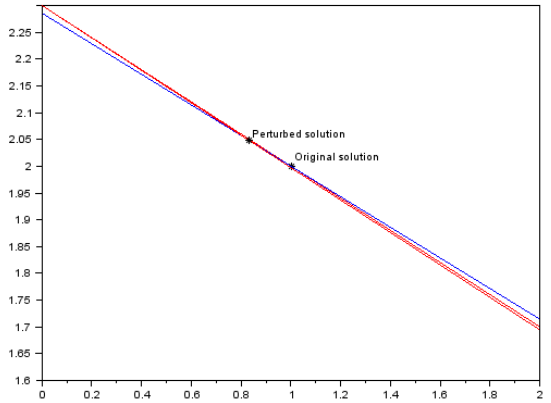
The intersection point is $(1.0\,,2.0)$. Now, if we change the coefficient of the second straight line from 2.1 to 2.12, the new solution becomes $(0.8333\,,2.048)$.
If we analyze the relative error of the system, we can see that the input relative error is approximately $\frac{\|\Delta \boldsymbol{d}\|}{\|\boldsymbol{d}\|} = \frac{|2.1-2.12|}{|2.1|} \approx 9.5 \times 10^{-3}$, while the relative errors on the solution are 1.67×10^{-1} for the variable x and 2.38×10^{-2} for y respectively. These values can be considered too high when compared to the order of magnitude of the input error. This is due to the fact that the two straight lines r_1 and r_2 are almost parallel (***ill-conditioned***).

Now, let's consider the system where r_1 is replaced by the orthogonal line r_3 to the line r_2,

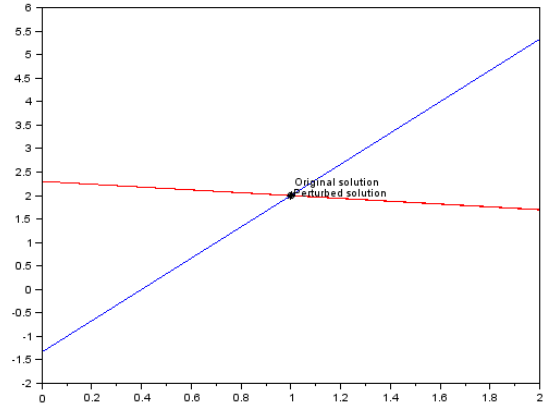
$$\mathrm{r}_3: -3.3333\,x + y = -1.3333.$$

Using the previous perturbation, the solution now moves from $(1.0,2.0)$ to $(0.9992\,,1.9974)$. In this case we have the same relative input error, but the relative errors on the solution are 7.86×10^{-4} for the variable x and 1.31×10^{-3} for y, which can be considered small (same order of magnitude of the input). This is due to the fact that the two straight lines r_3 and r_2 are almost orthogonal (***well-conditioned***).

The Scilab script is reported in the function ***stabline.sce***.



(Example of an ill-conditioned problem)



(Example of a well-conditioned problem)

Step 5: Zeros of polynomials

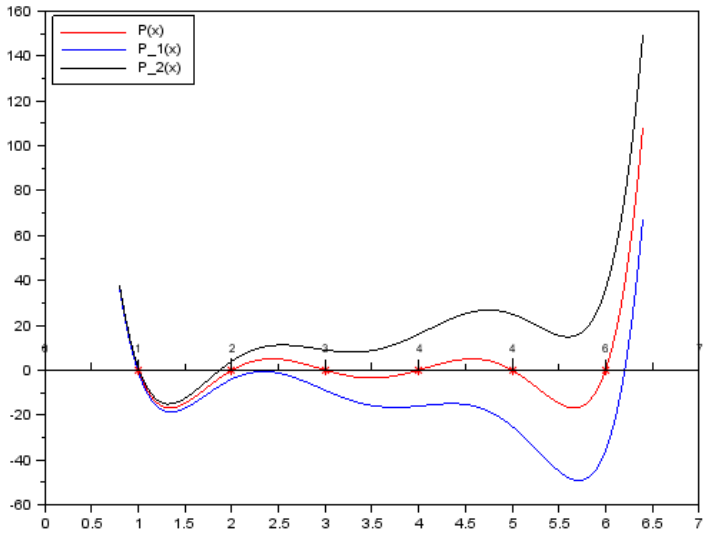
In this example we consider the problem of finding the real zeros of the polynomial

$$\begin{aligned} \mathrm{P}(x) &= (\mathrm{x}-1)(\mathrm{x}-2)(\mathrm{x}-3)(\mathrm{x}-4)(\mathrm{x}-5)(\mathrm{x}-6) \\ &= x^6 - 21x^5 + 175x^4 - 735x^3 + 1624x^2 - 1764x + 720 \end{aligned}$$

and of $\mathrm{P}_1(x)$ and $\mathrm{P}_2(x)$, which are obtained changing the coefficient of x^2 first with 1623 and then with 1625.

As we may notice in the figure on the right, in $\mathrm{P}_1(x)$ and $\mathrm{P}_2(x)$ some roots are moved from real to imaginary.

The Scilab script is reported in the function ***stabpoly.sce***.



(Example of stability for zeros of a polynomial)

roots(P(x))	roots(P_1(x))	roots(P_2(x))
6	+6.20616 +0.00000i	+5.70372 +0.38517i
5	+4.56804 +0.73363i	+5.70372 -0.38517i
4	+4.56804 -0.73363i	+3.35663 +0.63983i
3	+2.33290 +0.14639i	+3.35663 -0.63983i
2	+2.33290 -0.14639i	+1.87066 +0.00000i
1	+0.99195 +0.00000i	+1.00865 +0.00000i

(Roots of the three polynomials)

Step 6: Exercise #1

Let's consider the following polynomial

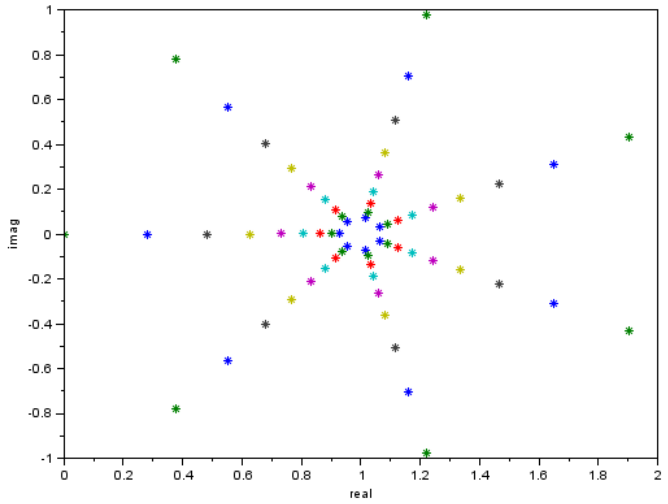
$$P_\varepsilon(x) = (x - 1)^7 - \varepsilon$$

which has 7 unitary roots for $\varepsilon = 0$.

Plot, in a complex plane, the roots of $P_\varepsilon(x)$ for different values of ε.

Hint: Use the Scilab function ***logspace*** to distribute the value of ε in a log space subdivision between 10^{-8} and 1. The function ***logspace(d1,d2, [n])*** distributes n points between 10^{d1} and 10^{d2}.

The Scilab script with the solution is reported in the function ***ex1.sce***.



(Results on roots perturbation with ε varying in $[10^{-8},1]$)

Step 7: Numerical stability

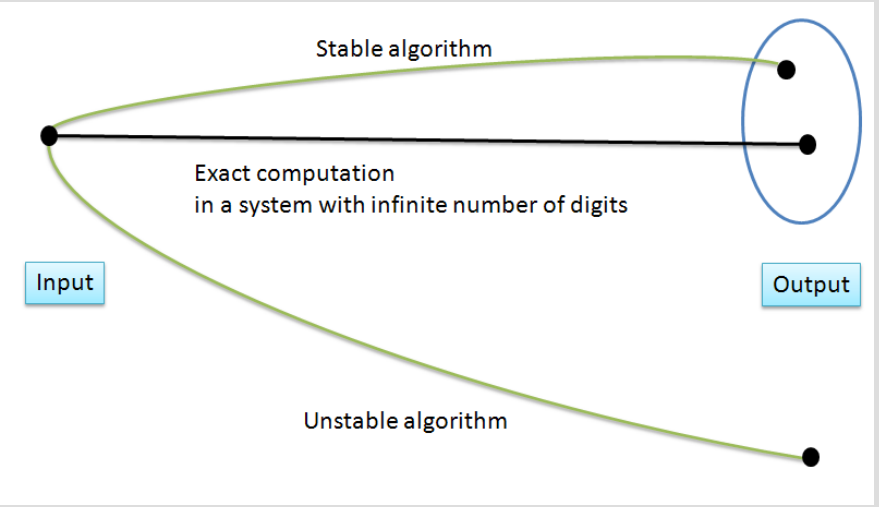
To solve a given problem we use a certain algorithm and its numerical implementation. Sometimes many algebraically equivalent solution strategies are available, but numerically they can lead to different results. This is due to computer's arithmetic, which can propagate errors in a more or less relevant way (see the Openeering tutorial on numerical errors).

Algorithms that do not magnify these errors are said to be ***numerically stable***.

On the other hand, if an algorithm is ***numerically unstable***, at a given point, the errors do not remain bounded and tend to grow up in an uncontrolled way corrupting completely the final result.

Hence, even when a problem is well-conditioned, if we try to solve it with an unstable algorithm, the obtained results will be meaningless.

The following examples refer to a comparison between stable and unstable algorithms for two given problems.



(Stable and unstable algorithm with respect to a solution obtained using an exact analytical procedure with an infinite number of digits)

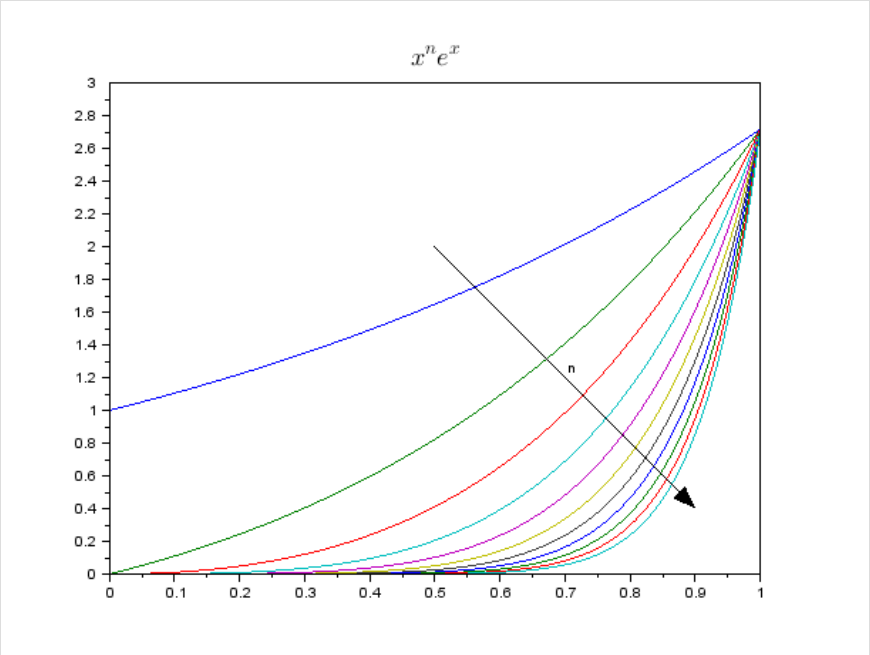
Step 8: Example of integral computation

In the next two steps, we compare two algorithms solving the following integral:

$$\mathrm{I_n} = \frac{1}{e}\int_0^1 x^n\, e^x dx, \quad n \geq 0\ (integer)$$

Both algorithms are based on the following theoretical considerations:

- $0 < \mathrm{I_n} < \int_0^1 x^n dx = \frac{1}{n+1}$;
- $\mathrm{I_{n+1}} < \mathrm{I_n}$;
- $\lim_{n\to\infty} \mathrm{I_n} = 0$



(the integrand)

Step 9: Example of integral computation (unstable formulation)

The first strategy is to develop an algorithm based on the following recursive formula:

- For $n = 0$ we have:

  $\mathrm{I_o} = \frac{1}{e}\int_0^1 e^x \; dx = \frac{1}{e}(e^1 - 1) = 1 - e^{-1}$;

- For $n > 0$ we can used integration by parts having

  $$\mathrm{I_n} = \frac{1}{e}\left(\int_0^1 x^n \, e^x dx = [x^n e^x]_0^1 - n\int_0^1 x^{n-1} \, e^x dx\right) = 1 - nI_{n-1}$$

The developed program starts from $n = 1$, where $\mathrm{I}_1 = e^{-1}$.

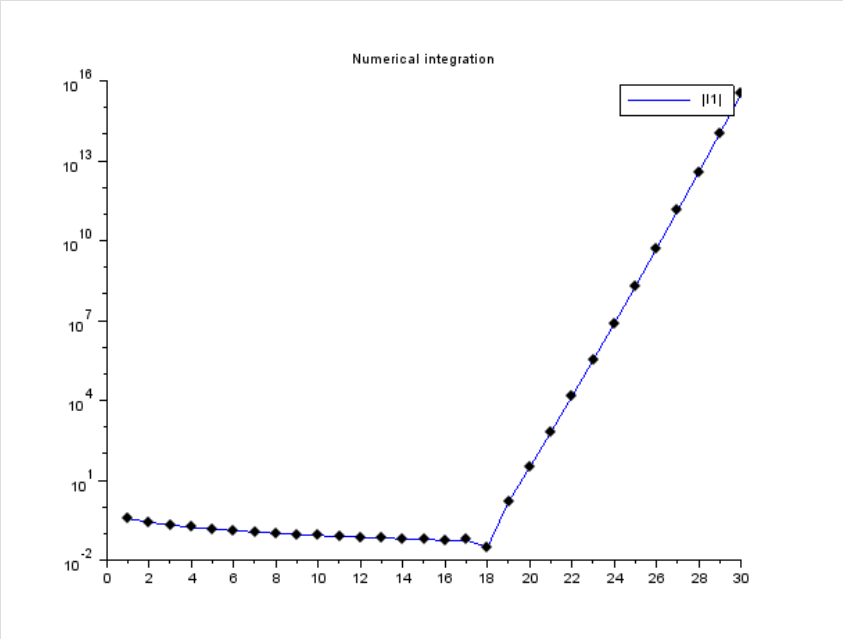
To perform the error analysis we denote by $\mathrm{I}'_\mathrm{n} = \mathrm{I_n} + \varepsilon_n$ the approximate value of the integral at step n with respect to the exact value $\mathrm{I_n}$ and making an error ε_n. Hence, it is possible to write the following recursive formula for the error

$$\varepsilon_n = \mathrm{I}'_\mathrm{n} - \mathrm{I_n} = (1 - n\mathrm{I}'_{\mathrm{n}-1}) - (1 - n\mathrm{I}_{\mathrm{n}-1}) = -n(\mathrm{I}'_{\mathrm{n}-1} - \mathrm{I}_{\mathrm{n}-1}) = -n\varepsilon_{n-1}$$

and ε_n, with respect to the first error ε_1, is

$$\varepsilon_n = (-1)^{n-1} n! \, \varepsilon_1$$

As a consequence, even if ε_1 is small, the error ε_n grows up to infinity as a factorial.



(Absolute value of $\mathrm{I_n}$)

Step 10: Example of integral computation (stable formulation)

The second strategy is to develop an algorithm based on the following recursive formula:

- For $n = N$ we set:

  $\mathrm{I_N} = 0;$

- For $n > 0$ we rewrite the previous recursive formula $\mathrm{I_n} = 1 - nI_{n-1}$ in terms of $n - 1$ as follows:

$$I_{n-1} = \frac{1}{n}(1 - \mathrm{I_n})$$

The developed program starts from $N = 100$ and computes $\mathrm{I_1}$ as the last integral.

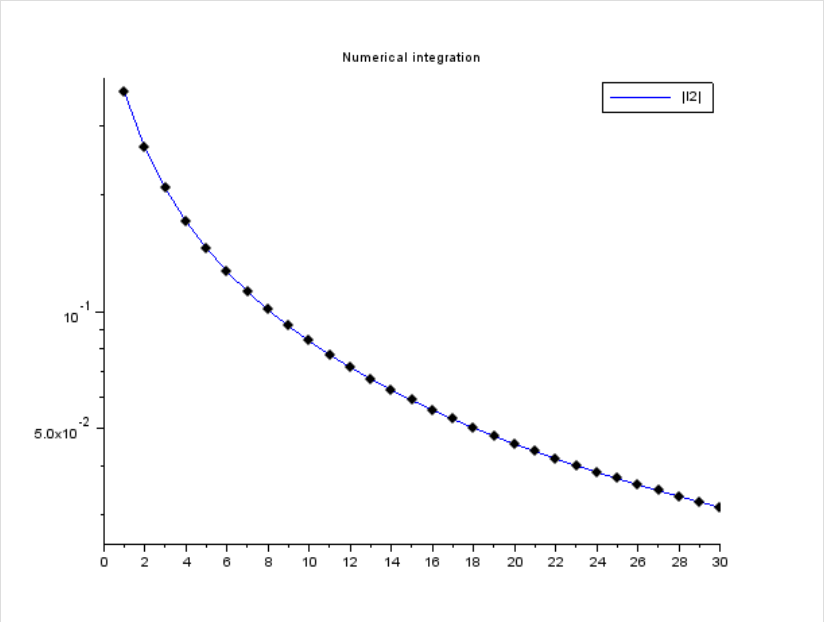
To perform the error analysis we denote by $\mathrm{I'_n} = \mathrm{I_n} + \varepsilon_n$ the approximate value of the integral at step n with respect to the exact value $\mathrm{I_n}$ and making an error ε_n. Hence, it is possible to write the following recursive formula for the error

$$\varepsilon_n = \mathrm{I'_n} - \mathrm{I_n} = (1 - n\mathrm{I'_{n-1}}) - (1 - n\mathrm{I_{n-1}}) = -n(\mathrm{I'_{n-1}} - \mathrm{I_{n-1}}) = -n\varepsilon_{n-1}$$

giving $\varepsilon_{n-1} = \frac{1}{-n}\varepsilon_n$. Hence ε_1 can be expressed in terms of ε_N as

$$\varepsilon_1 = \frac{(-1)^{N-1}}{N!}\varepsilon_N$$

As a consequence, even if ε_N is "big", the error ε_1 decreases to zero, since we have a factorial as denominator.



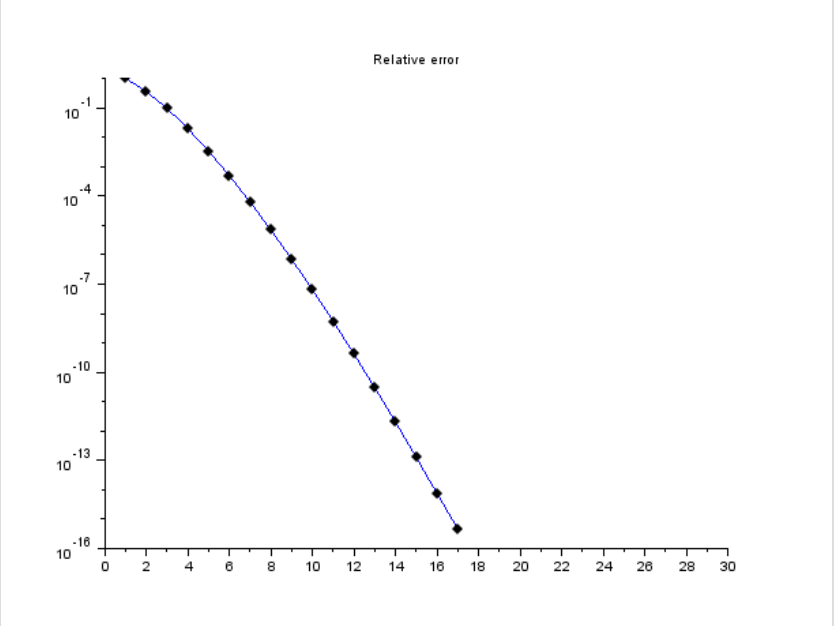
(Value of $\mathrm{I_n}$. The code for this plot is in intdemo.sce)

Step 11: Exercise #2

Considering the stable formulation, plot the relative error for I_1 starting from $I_N = 0$ for different values of N in the range $[1,30]$.

The relative error on ε_1 is computed as

$err_{rel} = \frac{|I_1^{exact} - I_1|}{|I_1^{exact}|}$ where I_1 depends on N and $I_1^{exact} = e^{-1}$.



(Relative error of the approximation of I_1 depending on N. Note that the values are visible only until N=17, after that limit the values are less then %eps and are not visible in a logarithmic scale)

Step 12: Concluding remarks and References

In this tutorial we have collected a series of numerical examples written in Scilab for the study of numerical stability.

1. Scilab Web Page: Available: www.scilab.org.
2. Openeering: www.openeering.com.
3. J. Higham, accuracy and Stability of Numerical Algorithms, SIAM
4. Atkinson, An Introduction to Numerical Analysis, Wiley

Step 13: Software content

To report a bugs or suggest improvements please contact Openeering team at the web site www.openeering.com.

Thank you for your attention,

Silvia Poles and Manolo Venturin

```
--------------
Main directory
--------------
stabline.sce        : Stability for line intersection
stabpoly.sce        : Stability for polynomial
intdemo.sce         : Integral computation example
ex1.sce             : Solution of exercise #1
ex2.sce             : Solution of exercise #2
license.txt         : The license file
```