Ordinary Differential Equations with Scilab
WATS Lectures
Provisional notes
Université de Saint-Louis
2004

G. Sallet

Université De Metz
INRIA Lorraine

2004

Table des matières

The content of these notes are the subject of the lectures given in august 2004, at the university Gaston Berger of Saint-Louis, in the context of the West African Training School (WATS) supported by ICPT and CIMPA.

1 Introduction

Ordinary differential equations (ODEs) are used throughout physics, engineering, mathematics, biology to describe how quantities change with time. The lectures given by Professors Lobry and Sari, last year, has introduced the basic concepts for ODEs. This lecture is concerned about solving ODEs numerically.
Scilab is used to solve the problems presented and also to make mathematical experiments. Scilab is very convenient problem solving environment (PSE) with quality solvers for ODEs. Scilab is a high-level programming language. Programs with Scilab are short, making practical to list complete programs. Developed at INRIA, Scilab has been developed for system control and signal processing applications. It is freely distributed in source code format.
Scilab is made of three distinct parts : an interpreter, libraries of functions (Scilab procedures) and libraries of Fortran and C routines. These routines (which, strictly speaking, do not belong to Scilab but are interactively called by the interpreter) are of independent interest and most of them are available through Netlib. A few of them have been slightly modified for better compatibility with Scilab's interpreter.
Scilab is similar to MATLAB, which is not freely distributed. It has many features similar to MATLAB. SCILAB has an inherent ability to handle matrices (basic matrix manipulation, concatenation, transpose, inverse etc.,) Scilab has an open programming environment where the creation of functions and libraries of functions is completely in the hands of the user.

The objective of this lecture is to give to the reader a tool, Scilab, to make mathematic experiments. We apply this to the study of ODE. This can be viewed as an example. Many others areas of mathematic can benefit of this technique.

This notes are intended to be used directly with a computer. You just need a computer on which Scilab is installed, and have some notions about the use of a computer. For example how to use a text editor, how to save a file.

Scilab can be downloaded freely from http ://www.Scilab.org. In this Web

site you will also find references and manuals. A very good introduction to Scilab is "Une introduction à Scilab" by Bruno Pinçon, which can be downloaded from this site.

We advise the reader to read this notes, and to check all the computations presented. We use freely the techniques of Scilab, we refer to the lectures by A.Sene associated to this course. All the routines are either given in the text or given in the appendix. All the routines have been executed and proved to be working. Excepted for some possible errors made on "cuting and pasting" the routines in this lectures are the original ones.

For the convenience of the reader, we recall the classical results for ODE's and take this opportunity to define the notations used throughout this lectures.

2 ODEs. Basic results

2.1 General Definitions

Definition 2.1 *:*
A first order differential is an equation

$$\dot{x} \;=\; f(t,x) \tag{1}$$

In this definition t represents time. The variable x is an unknown function from $\mathbb{R}$ with values in $\mathbb{R}^n$. Notation $\dot{x}$ is traditionally the derivative with respect to time t . The function f is an application, defined on an open set Ω of $\mathbb{R}\times\mathbb{R}^n$, with values in $\mathbb{R}^n$. The variable x is called the state of the system.

Definition 2.2 *:*
A solution of (1) is a derivable function x, defined on an interval I of $\mathbb{R}$

$$x : I \subset \mathbb{R} \longrightarrow \mathbb{R}^n$$

and which satisfies, on I

$$\frac{d}{dt}x(t) = f(t,x(t))$$

A solution is then a time parametrized curve, evolving in $\mathbb{R}^n$, such that the tangent at the point $x(t)$ to this curve, is equal to the vector $f(t,x(t))$.

2.2 Existence and uniqueness Theorems

From the title of this section you might imagine that this is just another example of mathematicians being fussy. But this is not : it is about whether you will be able to solve a problem at all and, if you can, how well. In this lecture we'll see a good many examples of physical problems that do not have solutions for certain values of parameters. We'll also see physical problems that have more than one solution. Clearly we'll have trouble computing a solution that does not exist, and if there is more than one solution then we'll have trouble computing the "right" one. Although there are mathematical results that guarantee a problem has a solution and only one, there is no substitute for an understanding of the phenomena being modeled.

Definition 2.3 (Lipschitz function) *:*
A function defined on an open set U of $\mathbb{R} \times \mathbb{R}^n$ with values in $\mathbb{R}^n$ is said to be Lipschitz with respect to x, uniformly relatively to t on U if there exists a constant L such that for any pair $(t, x) \in U$, $(t, y) \in U$ we have

$$\| f(t, x) - f(t, y) \| \leq L \| (x - y) \|$$

Definition 2.4 (Locally Lipschitz functions) *:*
A function defined on an open set Ω from $\mathbb{R} \times \mathbb{R}^n$ to $\mathbb{R}^n$ is said to be Lipschitz with respect to x, uniformly relatively to t sur Ω if for any pair $(t_0, x_0) \in \Omega$, there exists an open set $U \subset \Omega$ containing (t_0, x_0), a constant L_U such that f is Lipschitz with constant L_U in U.

Remark 2.1 *:*
Since all the norm in $\mathbb{R}^n$ are equivalent, it is straightforward to see that this definition does not depends of the norm. Only the constant L, depends on the norm.

Theorem 2.1 (Cauchy-Lipschitz) *:*
Let f a function defined on an open set Ω of $\mathbb{R} \times \mathbb{R}^n$, continuous on Ω, valued in $\mathbb{R}^n$ locally Lipschitz with respect to x, uniformly relatively to t on Ω .
Then for any point $(t_0, x_0$ of Ω there exists a solution of the ODE

$$\dot{x} = f(t, x)$$

defined on an open interval I of $\mathbb{R}$ containing t_0 and satisfying the initial condition

$$x(t_0) = x_0$$

Moreover if $y(t)$ defined on an interval, denotes another solution from the open interval J containing t_0 then

$$x(t) = y(t) \ \ sur \quad I \cap J$$

Definition 2.5 *:*
An equation composed of a differential equation with an initial condition

$$\begin{cases} \dot{x} & = & f(t,x) \\ x(t_0) & = & x_0 \end{cases} \tag{2}$$

is called a Cauchy problem

A theorem from Peano, gives the existence, but cannot ensures uniqueness

Theorem 2.2 (Peano 1890) *:*
If f is continuous any point (t_0, x_0) of Ω is contained in one solution of the Cauchy problem (2)

Definition 2.6 *A solution $x(t)$ of the Cauchy problem , defined on an open I, is said to be maximal if any other solution $y(t)$ defined on J,there exists at least $t \in I \cap J$, such that $x(t) = y(t)$ then we have $J \subset I$*

We can now give

Theorem 2.3 *:*
If f is locally lipschitz in x on Ω, then any point (t_0, x_0) is contained in an unique maximal solution

2.3 A sufficient condition for a function to be Lipschitz

If a function is sufficiently smooth, the function is Lipschitz. In other words if the RHS (right hand side) of the ODE is sufficiently regular we have existence and uniqueness of solution . More precisely we have

Proposition 2.1 *:*
If the function f of the ODE (1) is continuous with respect to (t, x) and is continuously differentiable with respect to x then f is locally Lipschitz.

2.4 A basic inequality

Definition 2.7 *A derivable function $z(t)$ is said to be an ε–approximate-solution of the ODE (1) if for any t where $z(t)$ is defined we have*

$$\| \dot{z}(t) - f(t, z(t)) \| \leq \varepsilon$$

Theorem 2.4 *:*
Let $z_1(t)$ et $z_2(t)$ two ε–approximate solutions of (1), where f is Lipschitz with constant L. If the solutions are respectively ε_1 et ε_2 approximate, we have the inequality

$$\| z_1(t) - z_2(t) \| \leq \| z_1(t_0) - z_2(t_0) \| \ e^{L|t-t_0|} \ + \ (\varepsilon_1 + \varepsilon_2) \frac{\left(e^{L|t-t_0|} - 1\right)}{L} \quad (3)$$

2.5 Continuity and initial conditions

Definition 2.8 *:*
If f is locally Lipschitz we have existence and uniqueness of solutions. We shall denote, when there is existence and uniqueness of solution, by

$$x(t, x_0, t_0)$$

the unique solution of

$$\begin{cases} \dot{x} = f(t, x) \\ x(t_0) = x_0 \end{cases}$$

Remark 2.2 *:*
The notation used here $x(t, x_0, t_0)$ is somewhat different of the notation used in often the literature, where usually the I.V. is written (t_0, x_0). We reverse the order deliberately, since the syntax of the solvers in Scilab are

```
--> ode(x0,t0,t,f)
```

Since this course is devoted to ODE and Scilab, we choose, for convenience, to respect the order of Scilab
The syntax in Matlab is

```
>> ode(@f, [t0,t],x0)
```

Proposition 2.2 *:*
With the hypothesis of the Cauchy-Lipschitz theorem the solution $x(t,\, x_0, t_0)$ *is continuous with respect to* x_0*. The solution is locally Lipschitz with respect to the initial starting point :*

$$\parallel x(t,x,t_0) \;-\; x(t,x_0,t_0) \parallel \;\leq e^{|t-t_0|} \parallel x - x_0 \parallel$$

2.6 Continuous dependence of parameters

We consider the following problem

$$\begin{cases} \dot{x} & = & f(t,x,\lambda) \\ x(t_0) & = & x_0 \end{cases}$$

where f is supposed continuous with respect to (t,x,λ) and Lipschitzian in x uniformly relatively to (t,λ).

Proposition 2.3 *With the preceding hypothesis the solution* $x(t,\, t_{0,}, x_0, \lambda)$ *is continuous with respect to* (t,λ)

2.7 Semi-continuity of the definition interval

We have seen in the Lipschitzian case that $x(t,\, x_0 t_0)$ is defined on an maximal open interval

$$t_0 \in \,]\alpha(x)\,,\, \omega(x)[$$

.

Theorem 2.5 *: Let* (t_0, x_0) *a point of the open set* Ω *and a compact interval* I *on which the solution is defined. Then there exists an open set* U *containing* x_0*such that for any* x *in* U *the solution* $x(t,x,t_0)$ *is defined on* I*.*

This implies that, for example, that $\omega(x)$ is lower semi-continuous .

2.8 Differentiability of solutions

Theorem 2.6 *We consider the system*

$$\left\{ \begin{array}{lcl} \dot{x} & = & f(t,x) \\ x(t_0) & = & x_0 \end{array} \right.$$

We suppose that f is continuous on Ω and such that the partial derivative $\frac{\partial f}{\partial x}(t,x)$ is continuous with respect to (t,x).
The the solution $x(t,\, t_0, x_0)$ is differentiable in t and satisfies the linear differential equation

$$\dot{\overbrace{\frac{\partial}{\partial x_0} x(t,x_0,t_0)}} = \frac{\partial f}{\partial x}(t, x(t,x_0,t_0)) \,.\, \frac{\partial}{\partial x_0} x(t,x_0,t_0)$$

2.9 Maximal solution

We have seen that to any point (t_0, x_0) is associated a maximal open set : $]\alpha(t_0,x_0)\ ,\ \omega(t_0,x_0)[$. What happens when $t \to \omega(t_0,x_0)$?

Theorem 2.7 *For any compact set K of Ω, containing the point (t_0,x_0),then $(t, x(t,\, t_0, x_0))$ does not stays in K when $t \to \omega(t_0,x_0)$*

Particularly when $\omega(t_0,x_0) < \infty$ we see that $x(t,\, t_0, x_0)$ cannot stay in any compact set.

3 Getting started

3.1 Standard form

The equation

$$\dot{x} = f(t,x)$$

is called the standard form for an ODE. It is not only convenient for the theory, it is also important in practice. In order to solve numerically an ODE problem, you must write it in a form acceptable to your code. The most common form accepted by ODE solvers is the standard form.
Scilab solvers accepts ODEs in the more general form

$$A(t,x)\dot{x} = f(t,y)$$

Where $A(t,x)$ is a non singular *mass matrix* and also implicit forms

$$g(t,x,\dot{x}) = 0$$

.

With either forms the ODEs must be formulated as a system of first-order equations. The classical way to do this, for differential equations of order n is to introduce new variables :

$$\begin{cases} x_1 = \dot{x} \\ x_2 = \ddot{x} \\ \cdots \\ x_{n-1} = x^{(n-1)} \end{cases} \tag{4}$$

Example 3.1 *:*
We consider the classical differential equation of the mathematical pendulum :

$$\begin{cases} \ddot{x} & = & -sin(x) \\ x(0) & = & a \\ \dot{x}(0) & = & b \end{cases} \tag{5}$$

This ODE is equivalent to

$$\begin{cases} \dot{x_1} = x_2 \\ x_2 = -sin(x_1) \\ x_1(0) = a; x_2(0) = b \end{cases} \tag{6}$$

3.2 Why to solve numerically ODEs ?

The ODEs, for which you can get an analytical expression for the solution, are the exception rather that the rule. Indeed the ODE encountered in practice have no analytical solutions. Even simple differential equations can have complicated solutions.
The solution of the pendulum ODE is given by mean of Jacobi elliptic function SN and CN. Now the solution of the ODE of the pendulum with friction, cannot be expressed, by known analytical functions.
Let consider some simple ODEs on $\mathbb{R}^n$

3.2.1 $\dot{x} = x^2 + t$

The term in x^2 prevents to use the method of variation of constants.Given in a computer algebra system (Mathematica, Maple, Mupad,...), this ODE has for solution

```
DSolve[x'[t] == x[t]^2 + t, x[t], t]
```

Which gives

$$\frac{N}{D}$$

Where

$$N = -C\, J_{-\frac{1}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right) + t^{\frac{3}{2}}\, J_{-\frac{2}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right) - C\, J_{-\frac{4}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right) + C\, J_{\frac{2}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right)$$

and

$$D = 2t\, J_{\frac{1}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right) + C\, J_{-\frac{1}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right)$$

Where the functions $J_\alpha(x)$ are the Bessel function. The Bessel functions Bessel $J_n(t)$ and Bessel $Y_n(t)$ are linearly independent solutions to the differential equation $t^2\ddot{x} + t\dot{x} + (t^2 - n^2)x = 0$. $J_n(t)$ is often called the Bessel function of the first kind, or simply the Bessel function.

3.2.2 $\dot{x} = x^2 + t^2$

this ODE has for solution proposed

```
DSolve[x'[t] == x[t]^2 + t^2, x[t], t]
```

Which gives

$$\frac{N}{D}$$

Where

$$N = -C\, J_{-\frac{1}{4}}\left(\frac{t^2}{2}\right) - 2t^2\, J_{-\frac{3}{4}}\left(\frac{t^2}{2}\right) - C\, J_{-\frac{5}{4}}\left(\frac{t^2}{2}\right) + C\, J_{\frac{3}{4}}\left(\frac{t^2}{2}\right)$$

and

$$D = 2t\, J_{\frac{1}{4}}\left(\frac{t^2}{2}\right) + C\, J_{-\frac{1}{4}}\left(\frac{t^2}{2}\right)$$

3.2.3 $\dot{x} = x^3 + t$

This time Mathematica has no answer

```
In [ 3] := DSolve[x'[t] == x[t]^3 + t, x[t], t]

Out[3]: = DSolve[x'[t] == x[t]^3 + t, x[t], t]
```

4 Theory

For a Cauchy problem , with existence and uniqueness

$$\begin{cases} \dot{x} = f(t, x) \\ x(t_0) = x_0 \end{cases} \tag{7}$$

A discrete variable method start with the given value x_0 and produce an approximation x_1 of the solution $x(t_1, x_0, t_0)$ evaluated at time t_1. This is described as advancing the integration a step size $h_0 = t_1 - t_0$. The processus is then repeated producing approximations x_j of the solution evaluated on a mesh

$$t_0 < t_1 < \cdots < t_i < \cdots < t_f$$

Where t_f is the final step of integration.
Usually the contemporary good quality codes, choose their steps to control the error, this error is set by the user or have default values.
Moreover some codes are supplemented with methods for approximating the solution between mesh points. The mesh points are chosen by the solver, to satisfy the tolerance requirements, and by polynomial interpolation a function is generated that approximate the solution everywhere on the interval of integration. This a called a continuous output. Generally the continuous approximation has the same error tolerance that the discrete method. It is

important to know how a code produces answers at specific points, so as to select an appropriate method for solving the problem with the required accuracy.

4.1 Generalities

Definition 4.1 *:*
We define the local error, at time (t_{n+1}) *of the mesh, using the definition (2.8), by*

$$e_n = x_{n+1} - x(t_{n+1}, x_n, t_n)$$

Or the size of the error by

$$le_n = \|x_{n+1} - x(t_{n+1}, x_n, t_n)\|$$

The method gives an approximation x_n at time t_n. The next step, start from (t_n, x_n) to produce x_{n+1}. The local error measure the error produced by the step and ignore the possible error accumulated already in x_n. A solver try to control the local error. This controls the global error

$$\| x_{n+1} - x(t_{n+1}, x_0, t_0)\|$$

Only indirectly, since the propagation of error can be upper bounded by

$$\| x(t_{n+1}, x_0, t_0) - x_{n+1}\| \ \leq \ \cdots$$

$$\cdots \leq \|x(t_{n+1}, x_n, t_n\ ) - x_{n+1}\| \ + \ \|x(t_{n+1}, x_0, t_0) - x(t_{n+1}, x_n, t_n\ )\|$$

The first term in the right hand side of the inequality is the local error, controlled by the solver, the second is given by the difference of two solutions evaluated at same time, but with different I.V. The concept of stability of the solutions of an ODE is then essential : two solutions starting from nearby Initial values, must stay close. The basic inequality (3) is of interest for this issue. This inequality tells us that is $L \mid t - t_0 \mid$ is small, the ODE has some kind of stability. The Lipschitz constant on the integration interval play a critical role. A big L would implies small step size to obtain some accuracy on the result. If the ODE is unstable the numerical problem can be ill-posed.

We also speak of well conditioned problem. We shall give two examples of what can happen. Before we give a definition

Definition 4.2 (convergent method) *:*
A discrete method is said to be convergent if on any interval $[t_0, t_f]$ *for any solution* $x(t, x_0, t_0)$*, of the Cauchy problem*

$$\begin{cases} \dot{x} = f(t, x) \\ x(t_0) = x_0 \end{cases}$$

If we denote by n *the number of steps, i.e.* $t_f = t_n$*,* $\tilde{x}_0$ *the first approximation of the method to the starting point* x_0*, then*

$$\|x(t_n) - x(t_f)\| \to 0$$

When $\max_i h_i \to 0$*, and* $\tilde{x}_0 \to x_0$*.*

It should be equivalent to introduce $\max_i \|x(t_i) - x(t_i)\|$. Our definition consider the last step, the alternative is to consider the error maximum for each step. Since this definition is for any integration interval, there are equivalent. We give two example of unstable ODE problems.

Example 4.1 *:*
We consider the Cauchy-Problem

$$\begin{cases} \dot{x_1} = x_2 \\ \dot{x}_2 = 100\, x_1 \\ x_1(0) = 1\ ,\ x_2(0) = -10 \end{cases} \tag{8}$$

The analytic solution is $x(t) = e^{-10t}$. The I.V. has been chosen for. We now try to get numerical solutions, by various solver implemented in SCILAB. Some commands will be explained later on. In particular, for the default method in SCILAB, we must augment the number of steps allowed. This is done by ODEOPTIONS. We shall see this in the corresponding section

```
-->;exec("/Users/sallet/Documents/Scilab/ch4ex1.sci");

-->x0=[0;0];

-->t0=0;
```

```
-->%ODEOPTIONS=[1,0,0,%inf,0,2,3000,12,5,0,-1,-1];

-->ode(x0,t0,3,ch4ex1)
 ans  =

!   0.0004145 !
!   0.0041445 !

-->ode('rk',x0,t0,3,ch4ex1)
 ans  =

!   0.0024417 !
!   0.0244166 !

-->ode('rkf',x0,t0,3,ch4ex1)
 ans  =

! - 0.0000452 !
! - 0.0004887 !

-->ode('adams',x0,t0,3,ch4ex1)
 ans  =

! - 0.0045630 !
! - 0.0456302 !

-->exp(-30)
 ans  =

    9.358E-14
```

All the results obtained are terrible. With LSODA, RK, RKF, RK,ADAMS. With RKF and ADAMS the value is even negative!! All the codes are of the highest quality and all fails to give a sound result. The reason is not in the

code, but in the ODE. The system is a linear system

$$\dot{x} = A\,x$$

with

$$A = \begin{bmatrix} 0 & 1 \\ 100 & 0 \end{bmatrix}$$

The eigenvalues of A are 1 and -10. As seen, in last year lectures, the solutions are linear combination of e^t and e^{-10t}. The picture is a saddle node (or mountain pass). The solution starting at $(1, -10)$ is effectively e^{-10t}. This one of the two solutions going to the equilibrium (the pass of the saddle) , but any other solution goes toward infinity. The ODE is unstable. Then when a solver gives an approximation which is not on the trajectory, the trajectories are going apart. The problem is ill conditioned. Moreover, it is simple to check that the Lipschitz constant is $L = 100$ for the euclidean norm (make in Scilab norm(A)...). The problem is doubly ill conditioned

Example 4.2 *:*
We consider the Cauchy-Problem

$$\begin{cases} \dot{x_1} = x_2 \\ \dot{x_2} = t\,x_1 \\ x_1(0) = \frac{3^{-\frac{2}{3}}}{\Gamma(\frac{2}{3})} \\ x_2(0) = -\frac{3^{-\frac{1}{3}}}{\Gamma(\frac{1}{3})} \end{cases} \tag{9}$$

This simple equation has for solution the special function Airy $Ai(t)$ With the help of Mathematica :

```
DSolve[{x'[t] == y[t], y'[t] == t x[t]}, {x[t], y[t]}, t]
```

which gives

$$\begin{cases} x(t) = C_1 Ai(t) + C_2 Bi(t) \\ y(t) = C_1 Ai'(t) + C_2 Bi'(t) \end{cases}$$

The function Ai et Bi are two independent functions solutions of the general equation $\ddot{x} = tx$. The Airy functions often appear as the solutions to boundary value problems in electromagnetic theory and quantum mechanics.

The Airy functions are related to modified Bessel functions with $\frac{1}{3}$ orders. It is known that $Ai(t)$ tends to zero for large positive t, while $Bi(t)$ increases unboundedly.
We check that the I.V are such that $Ai(t)$ is solution of the Cauchy-Problem.
Now the situation is similar to the preceding example. Simply, this example require references to special function. It is not so immediate as in the preceding example.
This equation is a linear nonautonomous equation $\dot{x} = A(t).x$, with

$$A(t) = \begin{bmatrix} 0 & 1 \\ t & 0 \end{bmatrix}$$

The norm associated to the euclidean norm is t (same reason as for the preceding example). Then the Lipschitz constant on $[0, 1]$ is $L = 1$. which gives an upper bound from the inequality (3) of e. This is quite reasonable. On the interval $[0, 10]$, the Lipschitz constant is 10, we obtain an upper bound of e^{100}. We can anticipate numerical problems.
We have the formulas (see Abramowitz and Stegun)

$$Ai(t) = \frac{1}{\pi}\sqrt{\frac{t}{3}}K_{\frac{1}{3}}\left(2/3t^{\frac{3}{2}}\right)$$

and

$$Bi(t) = \sqrt{\frac{t}{3}}\left[I_{-\frac{1}{3}}\left(2/3t^{\frac{3}{2}}\right) + I_{\frac{1}{3}}\left(\frac{2}{3}t^{\frac{3}{2}}\right)\right]$$

Where $I_n(t)$ and $K_n(t)$ are respectively the first and second kind modified Bessel functions. The Bessel K and I are built–in , in Scilab (Type help bessel in the window command)). So we can now look at the solutions :
We first look at the solution for $t_f = 1$, integrating on $[0, 1]$. In view of the upper bound for the Airy ODE on the error, the results should be correct.

```
-->deff('xdot=airy(t,x)','xdot=[x(2);t*x(1)]')
--> t0=0;
-->x0(1)=1/((3^(2/3))*gamma(2/3));
-->x0(2)=1/((3^(1/3))*gamma(1/3));
-->X1=ode('rk',x0,t0,1,airy)
 X1  =
```

```
!   0.1352924 !
! - 0.1591474 !

-->X2=ode(x0,t0,1,airy)
 X2  =

!   0.1352974 !
! - 0.1591226 !
--X3=ode('adams',x0,t0,1,airy)
 X3  =

!   0.1352983 !
! - 0.1591163 !
-trueval=(1/%pi)*sqrt(1/3)*besselk(1/3,2/3)
 trueval  =

    0.1352924
```

The results are satisfactory. The best result is given by the ' 'rk" method. Now we shall look at the result for an integration interval $[0, 10]$

```
-->X1=ode('rk',x0,t0,10,airy)
 X1  =

   1.0E+08 *

!   2.6306454 !
!   8.2516987 !

X2=ode(x0,t0,10,airy)
 X1  =

   1.0E+08 *

!   2.6281018 !
!   8.2437119 !

--X3=ode('adams',x0,t0,10,airy)
```

```
 X3  =

   1.0E+08 *

!   2.628452  !
!   8.2448118 !

-->trueval=(1/%pi)*sqrt(10/3)*besselk(1/3,(2/3)*10^(3/2))
 trueval  =

    1.105E-10
```

This time the result is terrible! The three solvers agree on 3 digits, around 2.6210^8, but the real value is $1.105\ 10^{-10}$. The failure was predictable. The analysis is the same of for the preceding example.

4.2 Backward error analysis

A classical view for the error of a numerical method is called backward error analysis. a method for solving an ODE with I.V. , an IVP problem approximate the solutions $x(t_n, x_0, t_0)$ of the Cauchy Problem

$$\begin{cases} \dot{x} & = & f(t,x) \\ x(t_0) & = & x_0 \end{cases}$$

We can choose a function $z(t)$ which interpolates the numerical solution. For example we can choose a polynomial function which goes through the points (t_i, x_i), with a derivative at this point $f(t_i, x_i)$. This can be done by a Hermite interpolating polynomial of degree 3. The function z is then with a continuous derivative. We can now look at

$$r(t) = \dot{z}(t) - f(t, z(t))$$

In other words the numerical solution z is solution of modified problem

$$\dot{x} = f(t,x) + r(t)$$

Backward error analysis consider the size of the perturbation. Well conditioned problem will give small perturbations.

4.3 One–step methods

One-steps methods are methods that use only information gathered at the current mesh point t_j. They are expressed under the form

$$x_{n+1} = x_n + h_n \, \Phi(t_n, x_n, h_n)$$

Where the function Φ depends on the mesh point x_n at time t_n and the next time step $h_n = t_{n+1} - t_n$.
In the sequel we shall restrict, for simplicity of the exposition, to one-step methods with constant step size. The result we shall give, are also true for variable step size, but the proof are more complicated.

Example 4.3 (Improved Euler Method) *;*
If we choose

$$\Phi(t_n, x_n, h_n) = f(t_n, x_n)$$

We have the extremely famous **Euler method**.

Example 4.4 (Euler Method) *;*
If we choose

$$\Phi(t_n, x_n, h_n) = \frac{1}{2} \left[f(t_n, x_n) + f(t_n + h_n, x_n + h_n \, f(t_n, x_n)) \right]$$

This method is known as the **Improved Euler method** *or also as* **Heun method**. Φ *depends on* (t_n, x_n, h_n).

4.3.1 Analysis of one-step methods

We shall analyze the error and propagation of error.

Definition 4.3 (Order of a method) *:*
A method is said of order p, if there exists a constant C such that for any point of the approximation mesh we have

$$le_n \ \leq \ C \, h^{p+1}$$

That means that the local error at any approximating point is $\mathcal{O}(h^{p+1})$.
For a one-step method

$$le_n = \| \, x(t_{n+1}, x_n, t_n) - x_n - \Phi(t_n, x_n, h)\|$$

In practice, the computation of x_{n+1} is corrupted by roundoff errors, in other words the method restart with $\tilde{x}_{n+1}$, in place of x_{n+1}.
In view of this errors we shall establish a lemma

Lemma 4.1 (propagation of errors) *:*
We consider a one-step method, with constant step. We suppose that the function $\Phi(t, x, h)$ is Lipschitzian of constant Λ relatively to x, uniformly with respect to h and t. We consider two sequences x_n and $\tilde{x}_n$ defined by

$$x_{n+1} = x_n + h\Phi(t_n, x_n, h) \quad \tilde{x}_{n+1} = \tilde{x}_n + h\Phi(t_n, \tilde{x}_n, h) + \varepsilon_n$$

We suppose that $\mid \varepsilon_n \mid \leq \varepsilon$
Then we have the inequality

$$\|\tilde{x}_{n+1} - x_{n+1}\| \leq e^{\Lambda|t_{n+1}-t_0|} \left(\| \, \tilde{x}_0 - x_0\| + (n+1)\varepsilon\right)$$

Proof
We have, using that Φ is Lipschitz of constant Λ :

$$\|\tilde{x}_{n+1} - x_{n+1}\| \leq (1 + h\Lambda)\|\tilde{x}_n - x_n\| + \varepsilon$$

A simple induction gives

$$\|\tilde{x}_{n+1} - x_{n+1}\| \leq (1 + h\Lambda)^{n+1}\|\tilde{x}_0 - x_0\| + \varepsilon\frac{e^{(n+1)h\Lambda} - 1}{h\Lambda}$$

Now using $(1 + x) \leq e^x$, and $t_{n+1} - t0 = (n + 1)h$, setting for shortness $T = t_{n+1} - t0$, we have :

$$\|\tilde{x}_{n+1} - x_{n+1}\| \leq e^{\Lambda T}\left(\|\tilde{x}_0 - x_0\| + (n+1)\varepsilon\frac{1 - e^{-\Lambda T}}{\Lambda T}\right)$$

Remarking that $\frac{1-e^{-x}}{x} \leq 1$, gives the result.
Now let consider any solution of the Cauchy problem. For example $x(t, \tilde{x}_0, t_0)$.
Set $z(t) = x(t, \tilde{x}_0, t_0)$. we have

$$z(t_{n+1}) = z(t_n) + h\Phi(t_n, z(t_n), h) + e_n$$

Where e_n is evidently

$$e_n = z(t_{n+1}) - z(t_n) - h\Phi(t_n, z(t_n), h)$$

By definition $le_n = \|e_n\|$.
We see that for any solution of the Cauchy problem we can define $\tilde{x}_n = x(t_n, \tilde{x}_0, t_0)$, since this sequence satisfies the condition of the lemma.
With this lemma we can give an upper bound for the global error. Since we can set $\tilde{x}_n = x(t_n, x_0, t_0)$ (we choose directly the good I.V.). Without roundoff error, we know that

$$\|e_n\| = le_n \ \leq \ C\, h^{p+1}$$

Then we see, using the lemma, that the final error $x(t_f, x_0, t_0) - x_{n+1}$ is upper bounded by $e^{\Lambda T}(n+1)Ch^{p+1}$, but since by definition $T = (n+1)h$, $t_{n+1} - t_0 = T$, $t_f = t_{n+1}$, we obtain

$$\|x(t_f, x_0, t_0) - x_{n+1}\| \ \leq \ e^{\Lambda T}(n+1)CTh^p$$

The accumulation of local error, gives an error of order p at the end of the integration interval. This justifies the name of order p methods. We have also proved that any method of order $p \leq 1$ is convergent. This is evidently theoritical, since with a computer, the sizes of steps are necessarily finite. But this is the minimum we can ask for a method.
If we add the rounding error to the theoretical error e_n, if we suppose that the rounding are upper bounded by ε (in fact, it is the relative error, but the principle is the same) we have

$$\tilde{x}_{n+1} = \tilde{x}_n + h\,(\Phi(t_n, \tilde{x}_n, h) + \alpha_n) + \beta_n$$

$$\tilde{x}_{n+1} = \tilde{x}_n + h\Phi(t_n, \tilde{x}_n, h) + h\alpha_n + \beta_n$$

We have $\varepsilon_n = h\alpha_n + \beta_n + e_n$, where α_n is the roundoff error made in computing Φ and β_n the roundoff error on the computation of $\tilde{x}_{n+1}$. If we suppose that we know upper bounds for α_n and β_n, for example M_1 and M_2, taking in consideration the theoretical error, using the lemma, a simple computation shows that the error $E(h)$ has an expression of the kind

$$E(h) = e^{\Lambda T}\left(\|\tilde{x}_0 - x_0\| + C\,T\,h^p + TM_1 + \frac{M_2 T}{h}\right)$$

So we can write

$$E(h) = K_1 + e^{\Lambda T} T \left(C\, h^p + \frac{M_2}{h} \right)$$

This shows that the error has a minimum for a step size of

$$\left(\frac{M_2}{pC} \right)^{\frac{1}{p+1}}$$

Which gives an optimal number of steps for an order p method

$$N_{opt} = T \left(\frac{pC}{M_2} \right)^{\frac{1}{p+1}} \tag{10}$$

In other words finite precision arithmetic ruins accuracy when too much steps are taken. Beyond a certain point the accuracy diminishes. We shall verify this fact experimentally.
To be correct the error is relative i.e. in computer the result $\bar{x}$ of the computation of a quantity x satisfies

$$\frac{\|\bar{x} - x\|}{\|x\|} \leq \mathbf{u}$$

Where for IEEE arithmetic $\mathbf{u} = 1.2\ 10^{-16}$ Since we are integrating on compact intervals, the solutions involved $x(t)$ are bounded and since Φ is continuous , $\Phi(t, x, h)$ is also bounded. This means that we must have an idea of the size of $x(t)$ and the corresponding Φ. We shall meet this kind of problem in the sequel, when we examine absolute error tolerance and relative error tolerance

4.3.2 Conditions for a one-step method to be of order p

We shall give necessary and sufficient condition for a method to be of order $\leq p$. Since the local error is

$$le_n = \|x_{n+1} - x(t_{n+1}, x_n, t_n)\|$$

Calling

$$e_n = x(t_{n+1}, x_n, t_n) - x_n - h\, \Phi(t_n, x_n, h)$$

If we suppose that Φ is of class $\mathcal{C}^p$, the Taylor formula gives

$$\Phi(t,x,h) = \sum_{i=0}^{p} \frac{h^i}{i!} \frac{\partial^i \Phi}{\partial h^i}(t,x,0) \quad + \circ(h^p)$$

If we suppose f of class $\mathcal{C}^p$,

$$x(t_{n+1}, x_n, t_n) - x_n = x(t_{n+1}, x_n, t_n) - x(t_n, x_n, t_n)$$

Once again by Taylor formula, denoting $x(t) = x(t, x_n, t_n)$

$$x(t_{n+1}, x_n, t_n) - x_n = \sum_{j=1}^{p+1} \frac{h^j}{j!} \frac{d^j x}{dt^j}(t_n, x_n) \quad + \circ(h^{p+1})$$

Now since $x(t)$ is solution of the differential equation we have

$$\frac{dx}{dt}(t_n, x_n) = f(t_n, x_n)$$

$$\frac{d^2x}{dt^2}(t_n, x_n) = \frac{\partial f}{\partial t}(t_n, x_n) \; + \; \frac{\partial f}{\partial x}(t_n, x_n)\, f(t_n, x_n)$$

To continue, we denote by

$$f^{[n]}(t, x(t)) = \frac{d^j x}{dt^j}\left[f(t, x(t))\right]$$

Then

$$f^{[2]}(t,x) = \frac{\partial^2 f}{\partial t^2}(t,x) + 2\frac{\partial^2 f}{\partial t\, \partial x}(t,x)\, f(t,x) + \frac{\partial^2 f}{\partial x^2}(t,x)\, (f(t,x), f(t,x))$$

The computation becomes rapidly involved, and special techniques, using graph theory, must be developped. But the computation can be made (at least theoretically). Then

$$x(t_{n+1}, x_n, t_n) - x_n = \sum_{j=1}^{p+1} \frac{h^j}{j!} f^{[j-1]}(t_n, x_n) \quad + \circ(h^{p+1})$$

Hence we deduce immediately (beware at the shift of indexes)

$$e_n = \sum_{k=1}^{p+1} \left[\frac{h^k}{k!} f^{[k-1]}(t_n, x_n) - \frac{h^k}{(k-1)!} \frac{\partial^{k-1} \Phi}{\partial h^{k-1}}(t_n, x_n, 0) \right] \quad + \circ(h^{p+1})$$

We have proven

Proposition 4.1 (order of a method) *:*
We consider a method of class $\mathcal{C}^p$. A method is of order at least $p+1$ (beware at the shift of indexes) iff Φ satisfies, for $k = 1 : p+1$

$$\frac{1}{k} f^{[k-1]}(t,x) = \frac{\partial^{k-1}\Phi}{\partial h^{k-1}}(t,x,0)$$

Corollary 4.1 (order 1 method) *:*
A method is of order at least 1, iff

$$f(t,x) = \Phi(t,x,0)$$

The Euler method is then of order 1.

Corollary 4.2 (order 2 method) *:*
A method is of order at least 2, iff

$$f(t,x) = \Phi(t,x,0)$$

and

$$\frac{1}{2}\left[\frac{\partial f}{\partial t}(t,x) \; + \; \frac{\partial f}{\partial x}(t,x)\, f(t,x)\right] \; = \; \frac{\partial \Phi}{\partial h}(t,x,0)$$

Check that Heun is an order 2, method.

4.3.3 Runge-Kutta method

The Runge-Kutta methods are one-steps methods. The principle of RK methods is to collect information around the last approximation, to define the next step. Two RK methods are implemented in Scilab : “rk” and “rkf” .
To describe a RK method we define Butcher Arrays.

Definition 4.4 (Butcher Array) *:*
A Butcher array of size $(k+1) \times (k+1)$ is an array

$$\begin{array}{c|ccccc} c_1 & 0 & & & & \\ c_2 & a_{2,1} & 0 & & & \\ c_3 & a_{3,1} & a_{3,2} & 0 & & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \\ c_k & a_{k,1} & a_{k,2} & \cdots & a_{k,k-1} & 0 \\ \hline & b_1 & b_2 & \cdots & b_{k-1} & b_k \end{array}$$

The c_i coefficients satisfying

$$c_i = \sum_{j=1}^{i-1} a_{i,j}$$

A Butcher array is composed of $(k-1)^2 + k$ datas.
We can now describe a k-stages Runge-Kutta formula

Definition 4.5 (k-stages RK formula) *:*
A k-stages RK formula is defined by a $k+1$ Butcher array. If a (t_n, x_n) approximation has be obtained, the next step (t_{n+1}, x_{n+1}) is given by the following algorithm :
First we define k intermediate stages, by finite induction , which gives points $x_{n,i}$ and slopes s_i :

$$x_{n,1} = x_n \quad s_1 = f(t_n, x_{n,1})$$

$$x_{n,2} = x_n + h\, a_{2,1}\, s_1 \quad s_2 = f(t_n + c_2\, h\, ,\, x_{n,2})$$

$$x_{n,3} = x_n + \, h\, (a_{3,1}\, s_1 \, + \, a_{3,2} s_2) \quad s_3 = f(t_n + c_3\, h\, ,\, x_{n,3})$$

$\ldots$

$$x_{n,k} = x_n + \, h \sum_{i=1}^{k-1} a_{k,i} s_i \quad s_k = f(t_n + c_k\, h\, ,\, x_{n,k})$$

The next step is then defined by

$$x_{n+1} = x_n + h \sum_{i=1}^{k} b_i\, s_i$$

The intermediates points $x_{n,i}$ are associated with times $t_n + c_i\, h$.
We shall give a quite complete set of examples, since this RK formulas are used by high quality codes, and can be found in the routines.

Example 4.5 (Euler RK1) *:*
The Euler method is a RK formula with associated Butcher array

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$

Example 4.6 (Heun RK2) *:*
The Heun method (or improved Euler) is a RK formula with associated Butcher array

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Example 4.7 (Heun RK2) *:*
The Midpoint method, named also modified Euler formula, is a RK formula with associated Butcher array

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$$

Example 4.8 (Classical RK4) *:*
The classical RK4 formula with associated Butcher array is

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$$

Example 4.9 (Bogacki-Shampine Pair, BS(2,3)) *:*
It is the first example of a Embedded RK formulas. We shall come on this matter later.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{3}{4}$	0	$\frac{3}{4}$		
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

Example 4.10 (Dormand Prince Pair DOPRI5) *:*
It is the second example of embedded RK formulas.

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$-\frac{49}{176}$	$-\frac{5103}{18656}$		
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

Example 4.11 (an ode23 formula) *:*
This example was the embedded RK formulas used in MATLAB till version 4, for the solver ODE23.

0				
$\frac{1}{4}$	$\frac{1}{4}$			
$\frac{27}{40}$	$-\frac{189}{800}$	$\frac{729}{800}$		
1	$\frac{214}{891}$	$\frac{1}{33}$	$\frac{650}{891}$	
	$\frac{41}{162}$	0	$\frac{800}{1053}$	$-\frac{1}{78}$

Example 4.12 (RKF Runge-Kutta Fehlberg 45) *:*
This formula is the Runge-Kutta Fehlberg coefficients. They are used in Scilab "RKF". It is still an "interlocking" RK formula.

0					
$\frac{1}{4}$	$\frac{1}{4}$				
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$			
$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$	$\frac{7296}{2917}$		
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$-\frac{49}{176}$	$-\frac{5103}{18656}$
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$	$-\frac{1}{5}$

4.3.4 Order of RK formulas

We shall derive conditions for which RK formulas are of order p. We have said that for a RK formula the coefficient c_i are imposed. We shall justify this assumption.
Any ODE can be rendered autonomous, or more exactly replaced by an equivalent autonomous system. The trick is to built a "clock". Let the classical system

$$\left\{ \begin{array}{lcl} \dot{x} & = & f(t,x) \\ x(t_0) & = & x_0 \end{array} \right. \tag{11}$$

This system is clearly equivalent to the autonomous system

$$\left\{ \begin{array}{lcl} \dot{y} = 1 & & \\ \dot{z} & = & f(y,z) \\ z(t_0) & = & x_0 \\ y(t_0) = t_0 & & \end{array} \right. \tag{12}$$

Actually some codes require the equations to be presented in autonomous form.
It is expected that a solver will have the same behavior with the two systems.

We shall compare the steps taken by a RK formula for the two systems. Let before denote by $\bar{S}_i$ the slopes for the second system in $\mathbb{R} \times \mathbb{R}^n$, necessarily of form

$$\bar{S}_i = \begin{vmatrix} 1 \\ \bar{s}_i \end{vmatrix}$$

And by $\bar{W}_{n,i}$ the intermediate points obtained. Since the first coordinate of the autonomous system is a clock, starting at time t_0, since the sequence of the original system is given by the intermediate points $x_{n,i}$ evaluated at time $t_n + c_i h$, the code should gives a sequence, for the intermediates points corresponding to the autonomous system

$$\bar{W}_{n,i} = \begin{vmatrix} t_n + c_i h \\ x_{n,i} \end{vmatrix}$$

we have, using the notations of RK formulas

$$s_1 = f(t_n, x_n) \qquad \bar{s}_1 = \begin{vmatrix} 1 \\ f(t_n, x_n) \end{vmatrix}$$

$$x_{n,2} = x_n + h a_{2,1} s_1 \qquad \bar{W}_{n,2} = \begin{vmatrix} t_n + h a_{2,1} \\ x_{n,2} \end{vmatrix}$$

$$s_2 = f(t_n + c_2\, h, x_{n,2}) \qquad \bar{S}_2 = \begin{vmatrix} 1 \\ \bar{s}_2 = f(t_n + h a_{2,1}, x_{n,2}) \end{vmatrix}$$

We must have $c_2 = a_{2,1}$. If this is satisfied then $\bar{s}_2 = s_2$, hence for the next step

$$x_{n,3} = x_n + \, h \, (a_{3,1}\, s_1 \, + \, a_{3,2} s_2) \qquad \bar{W}_{n,3} = \begin{vmatrix} t_n + h \, (a_{3,1} \, + \, a_{3,2}) \\ x_{n,3} \end{vmatrix}$$

Then we must have $c_2 = a_{3,1} \, + \, a_{3,2}$, we obtain by induction that the c_i are equal to the sum of the $a_{i,j}$ on the same row.

To derive condition for RK method we use the corallary (4.1). For a RK method the function Phi is given by

$$\Phi(t, x, h) = \sum_{i=1}^{k} b_i \, s_i$$

Where it is clear that it is the s_i which depends on (t, x, h).

RK formulas of order 1 From corollary (4.1) we must have $f(t,x) = \Phi(t,x,0)$. When $h = 0$ from the formulas, all the intermediates points are given by $x_{n,1} = x_n$, and $s_i = f(t_n, x_n)$. Hence

$$\Phi(t,x,0) = f(t,x) \sum_{i=1}^{k} b_i = f(t,x)$$

A RK formula is of order at least 1, iff

$$\sum_{i=1}^{k} b_i \;=\; 1$$

.

RK formulas of order 2 :
From corollary (4.2), if it of order 1 and if

$$\frac{1}{2}\left[\frac{\partial f}{\partial t}(t,x) \;+\; \frac{\partial f}{\partial x}(t,x)\, f(t,x)\right] \;=\; \frac{\partial \Phi}{\partial h}(t,x,0)$$

We must evaluate $\frac{\partial \Phi}{\partial h}(t,x,h)$. We have

$$\sum_{i=1}^{k} b_i \frac{\partial s_i}{\partial h}$$

From the formulas

$$\frac{\partial s_i}{\partial h} = c_i \frac{\partial f}{\partial t}(t_i + c_i h, x_{n,i}) + \frac{\partial f}{\partial x}\left[\sum_{k=1}^{i-1} a_{i,k} s_k + hA\right]$$

Where the expression A is given by

$$\sum_{k=1}^{i-1} a_{i,k} \frac{\partial s_k}{\partial h}$$

.
When $h = 0$ we have already seen that $x_{n,i} = x_n$ and $s_i = f(t_n, x_n)$, then

$$\frac{\partial s_i}{\partial h}(t,x,0) = c_i \frac{\partial f}{\partial t}(t,x)) + \frac{\partial f}{\partial x}\left[\sum_{k=1}^{i-1} a_{i,k} f(t,x)\right]$$

Withe the hypothesis $\sum_{k=1}^{i-1} a_{i,k} = c_i$, using the fact that $\frac{\partial f}{\partial x}$ is a linear application we get

$$\frac{\partial s_i}{\partial h}(t,x,0) = c_i \left[\frac{\partial f}{\partial t}(t,x)) + \frac{\partial f}{\partial x}(t,x) f(t,x)\right] = c_i f^{[1]}(t,x)$$

Finally the condition reduces to

$$\frac{1}{2} f^{[1]}(t,x) \;=\; f^{[1]}(t,x) \sum_{i=1}^{k} b_i c_i$$

We have proved that a RK is at least of order 2 if

$$\sum_{i=1}^{k} b_i \;=\; 1 \qquad \text{and} \qquad \sum_{i=1}^{k} b_i c_i = \frac{1}{2}$$

The computation can be continued, but appropriate techniques must be used. See Bucher or Haire and Wanner. The condition obtained can be expressed as matrix operations. We introduce, from the Butcher array, the matrix A, of size $k \times k$. The coefficients of A for $i > j$ are the $a_{i,j}$ of the array. The other coefficient are 0. We introduce the length k column vector C and the length k vector B, in the same manner. We use freely the Scilab notation. Particularly Scilab can be used for testing the order of a RK formula. So we express the relation as a Scilab test. We shall give condition till order 5. RK formulas of order 7 and 8 exists. Finding RK coefficients is also an art. Some formulas of the same order are more efficient.

Relations for Order of RK formulas :
Each test add to the preceding.
Before, it must be verified that the c_i satisfy the condition $\sum a_{i,j} = c_i$:

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

The test must output %T.
Order 1

```
Sum(B)==1
```

Order 2

```
B*C==1/2
```

Order 3

```
B*C.^2==1/3        B*A*C==1/6
```

Order 4

```
B*C.^3==1/4           B*A*C.^ 2==1/12

B*(C.*(A*C))==1/8      B*A^ 2*C==1/24
```

Order 5

```
B*C.^4==1/5           B*( (C.^2) .* (A*C))==1/10

B*(C.*(A*C.^2))==1/15      B*( C.*((A^2)* C)==1/30

B*((A*C).*(A*C))==1/20        B*A*C.*^3==1/20

(B*A) *(C.*(A*C))==1/40     B*A^2*C.^2==1/60

B*A^3C== 1/120
```

Exercise
Check, on the examples, the order of the RK formulas. For the Embedded RK formulas, you will find out that, for example for the BS(2,3) pair that if you only take the 4×4 Butcher array of the BS(2,3) you have an order 3 RK formula. The complete array gives an order 2 formula. This explain the line and the double line of the array. Since the computation cost of a code is measured by the number of evaluation of f, the BS(2,3) give the second order formula for free. The interest of such formulas for step size control will be explained in the next sections.

4.3.5 RK formulas are Lipschitz

If we prove this, then from our preceding results, the RK formulas are convergent and at least of order 1.
We must evaluate

$$\Phi(t,x,h) - \Phi(t,y,h) = \sum_{i=1}^{k} b_i \left(s_i(x) - s_i(y)\right)$$

We denote by $\alpha = max|a_{i,j}|$, $\beta = max|b_i|$, by L the Lipschitz constant of f,
With the notation of RK formula, a simple induction shows that (for the intermediate points x_i and y_i) we have

$$\|x_i - y_i\| \ \leq \ (1 + h\alpha L)^{i-1} \ \|x - y\|$$

and

$$\|s_i(x) - s_i(y)\| \ \leq \ L(1 + h\alpha L)^{i-1} \ \|x - y\|$$

Hence

$$\|\Phi(t,x,h) - \Phi(t,y,h)\| \ \leq \ \beta \frac{(1 + h\alpha L)^k - 1}{h\alpha} \ \|x - y\|$$

This prove that the RK formula are Lipschitz methods.

4.3.6 Local error estimation and control of stepsize

The local error of a method of order p is given par

$$e_n = x(t_{n+1}, x_n, t_n) - x_{n+1}$$

Suppose that we have a formula of order $p+1$ to compute an estimate x^*_{n+1}, then we have

$$est_n = x^*_{n+1} - x_{n+1} = [x(t_{n+1}, x_n, t_n) - x_{n+1}] - \left[x(t_{n+1}, x_n, t_n) - x^*_{n+1}\right]$$

Then

$$\|est_n\| = le_n + \mathcal{O}(h^{p+2}) = \mathcal{O}(h^{p+1}) = Ch^{p+1}$$

Where C is depending on (t_n, x_n). Since le_n is $\mathcal{O}(h^{p+1})$, the difference est_n give a computable estimate of the error. Since the most expensive part of taking a step is the evaluation of the slopes, embedded RK formulas are particularly interesting since they give an estimation of the error for free.

Modern codes use this kind of error estimation. If the error is beyond a tolerance *tol* (given by the user, or is a default value used by the code), the step is rejected and the code try to reduce the stepsize.
If we have taken a step σh, the local error would have been

$$le_n^\sigma = C(\sigma h)^{p+1} = \sigma^{p+1}\|est\|$$

, We use here $\|est_n\|$ for le_n, then the step size σh passing the test satisfies

$$\sigma < \left(\frac{tol}{\|est\|}\right)^{\frac{1}{p+1}}$$

The code will use for new step $h\left(\frac{tol}{\|est\|}\right)^{\frac{1}{p+1}}$.
This is the way that most popular codes select the step size. But there is practical details. This is only an estimation. Generally how much the step size is decreased or increased is limited. Moreover a failed step is expensive, the codes use, for safety, a fraction of the predicted step size, usually 0.8 or 0.9. Besides a maximum step allowed, can be used, to prevent too big steps. The maximal step increase is usually chosen between 1.5 and 5. It is also used that after a step rejection to limit the increase to 1.
In the case of embedded RK formulas, two approximation are computed. If the step is accepted, with which formula does the code will advance ? Advancing with the higher order result is called **local extrapolation**. This is for example the case of the BS(2,3) pair. The 4th line gives of BS(2,3) a 3 order RK formula and an estimation say x_{n+1}. To compute the error $f(t_{n+1}, x_{n+1})$ is computed. If the step is accepted, if the code advance with x_{n+1}, the slope is already computed. The next step is free (at least for evaluation of f). This kind of formula are called FSAL (first same as last). The pairs DOPRI5 and RKF are also a FSAL pair. (check it !).

4.4 Experimentation

The objective of this section is to experiment and verify some of the theoretical results obtained. The power of Scilab is used. For the need of experiments we shall code some of the methods described before. This is for heuristic reasons. We give here an advice to the reader : do not rewrite codes that have been well written before and have proved to be sure. The solver of Scilab

are high quality codes, don't reinvent the wheel. However it is important to know how codes are working. This was the reason of the preceding chapter.

4.4.1 coding the formulas

We shall code three RK formulas : Euler, Heun, RK4 (classical). Since all the formulas given are RK formulas we shall indicates one general method. We suppose that the Butcher array is given by the " triplette " (A, B, C) of size respectively $k \times k$, $1 \times k$ and $k \times 1$. With theses data you can check the order of the RK formula.
When this done we suppress the first row of A (only 0).

```
A(1,:)=[ ];
```

The computation of the slopes is straighforward. We can write a k-loop or simply write everything. In pseudo-code, for the main loop (computing x_{i+1}) , we obtain :
Assuming that the RHS is coded in odefile with inputs (t, x), that k is the number of stages of the RK formulas, that x_i has already been computed :

```
n=length(x0) // dimension of the ODE
s=zeros ( n,k)
//  k the number of stages, preallocation of the
// matrix of slopes

A1=A';   //technical reasons

// computing x(i+1)
// inside the slope  lightening the notation.
x=x(i);
t=t(i);
sl=feval ( t ,x, odefile); s(:,1)=sl(:);
//making sure sl is a column vector
sl=feval(t+h*C(2),x +h*A1(:,1)); s(:,2)=sl(:);
.... write the other stages till k-1 and the last
sl=feval (t+h*C(k), x+h*A(:,k)); s(:,k)=sl(:);

// compute the new point
```

```
x= x+ h*s*B';
x(i+1)=x;
```

This the core of the code, care must be taken to details. How is stored $x(i)$ column ? row ?. Since the utility of a solver is also to plot solution, the mesh points should be stored as a column vector, then $x(i)$ should be the i-th row of $N \times n$ matrix, where N is the number of mesh points. Once you have write the code, this is not difficult to code all the RK formulas.
Do it !
the routine can also be coded more naively, for example RK4, here is a Scilab file, named rk4.sci :

```
function [T,X]=rk4(fct,t0,x0,tf,N)
//Integrates with RK4 classical  method
//  fct user supplied function : the RHS of the ODE
//
//  t0 initial time
//
//   x0 initial condition
//
//   tf final time  [t0,tf] is the integration interval
//
//   N number of steps

x0=x0(:) // make sure x0 column !
n=length(x0)
h=(tf-t0)/(N-1)
T=linspace(t0,tf,[,N])
T=T(:)
X=zeros(N,n) // preallocation
X(1,:)=x0'

//main loop

for i=1:N-1
// compute the slopes
s1=feval(T(i),X(i,:),fct)
s1=s1(:)
```

```
x1=X(i,:)+h*(s1/2)'
s2=feval(T(i)+h/2,x1',fct)
s2=s2(:)
x2=X(i,:)+h*(s2/2)'
s3=feval(T(i)+h/2,x2',fct)
s3=s3(:)
x3=X(i,:)+h*s3'
s4=feval(T(i)+h,x3',fct)
s4=s4(:)
X(i+1,:)=X(i,:)+h*(s1'+2*s2'+2*s3'+s4')/6
end
```

A WARNING
When you create a Scilab file, this file has the syntax

```
function   [out1,out2, ...] =name_of_function (in1,in2, ...)
```

Where output : out1, out2, ...are the wanted quantities computed by the file, and the in1, in2, ...are the inputs required by the file.

ALWAYS SAVE THE FILE UNDER THE NAME OF THE GIVEN FUNCTION ! That is save under

```
name_of_function.sci.
```

If you are saving under another name Scilab get confused, and you are in trouble.

4.4.2 Testing the methods

We shall use a test function, for example $\dot{x} = x + t$ with I.V. $x(0) = 1$. The exact solution is $x(t) = -1 - t - e^t$.
We write the file fctest1.sci

```
function xdot=fctest1(t,x)
//test fonction
//
xdot=x+t;
```

And finally write a script for testing the three methods

```
//compare the different Runge Kutta methods
// and their respective error
//
//initialization
//
;getf("/Users/sallet/Documents/Scilab/euler.sci");
;getf("/Users/sallet/Documents/Scilab/rk2.sci");
;getf("/Users/sallet/Documents/Scilab/rk4.sci");
;getf("/Users/sallet/Documents/Scilab/fctest1.sci");


Nstepvect=(100:100:1000);
n=length(Nstepvect);
eul_vect=zeros(n,1);
rk2_vect=zeros(n,1);
rk4_vect=zeros(n,1);

for i=1:n
[s,xeul]=euler(fctest1,0,1,3,Nstepvect(i));
// sol for euler
[s,xrk2]=rk2(fctest1,0,1,3,Nstepvect(i));
// sol for Heun  RK2
[s,xrk4]=rk4(fctest1,0,1,3,Nstepvect(i));
// sol for RK4
z=-1-s+2*exp(s); // exact solution for the points s
eulerror=max(abs(z-xeul)); // biggest error for Eul
rk2error=max(abs(z-xrk2)); // biggest error for RK2
rk4error=max(abs(z-xrk4)); // biggest error for RK4
//
//
eul_vect(i)=eulerror;
rk2_vect(i)=rk2error;
rk4_vect(i)=rk4error;
//
end
//
//plot
xbasc()
```

```
plot2d('ll',Nstepvect,[eul_vect,rk2_vect,rk4_vect])
xgrid(2)
```

Now we call this script from the Scilab command window :

```
;getf("/Users/sallet/Documents/Scilab/comparRK.sci");
```

We get the following picture

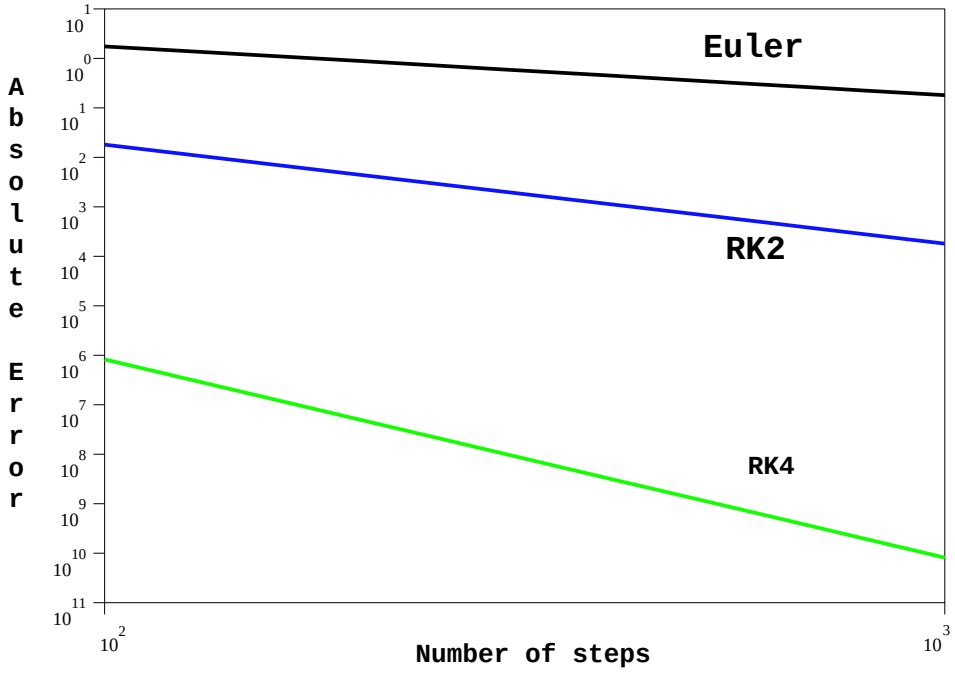

FIG. 1 – Comparison of RK Formulas

Clearly measuring the slopes show that the methods are respectively of order 1, 2 and 4, as expected.

4.4.3 Testing roundoff errors

We want to experiment the effects of roundoff, that is to say the effects of finite precision arithmetic. We choose for example the RK2 improved Euler method, and we test the method with the function $\dot{x} = x$ and the I.V. $x_0 = 1$. We look at $x(1) = e$.

Since we look only at the terminal point, and since we shall have a great number of steps, we choose to minimize the number of variables used. Scilab stores usual variables in a stack. The size of this stack depends on the amount of free memory. Then we use for this experiment a modified code of RK2, minimizing the number of variables. Compare with the original code given RK2 (same as RK4) before.

```
function sol=rk2mini(fct,t0,x0,tf,N)
// Integrates with RK2 method, improver Euler,
// with minimum code, minimum allocation memory
// for experiment computing the solution only
// at final time
//
//  fct user supplied function : the RHS of the ODE
//
//  t0 initial time
//
//   x0 initial condition
//
//   tf final time  [t0,tf] is the integration intervall
//
//   N number of steps

x0=x0(:)
n=length(x0)
h=(tf-t0)/N
x=x0;
t=t0;
k=1;

//main loop

while k< N
s1=feval(t,x,fct) // compute the slope
s1=s1(:)
x1=x+h*s1   // Euler point
s2=feval(t+h,x1,fct) // slope at Euler point
s2=s2(:)
```

```
x=x+h*(s1'+s2')/2
t=t+h
k=k+1
end

sol=x
```

Since we work in finite precision, we must be aware that $(t_f - t_0) \neq N * h$. In fact $N * h$ is equal to the length of the integration interval at precision machine. The reader is invited to check this.
We obtain the following results :

```
-->result
 result  =

!   10.           0.0051428        0.0011891!
!   100.          0.0000459        0.0000169!
!   1000.         4.536E-07        1.669E-07!
!   10000.        4.531E-09        1.667E-09!
!   100000.       4.524E-11        1.664E-11!
!   1000000.      2.491E-13        9.165E-14!
!   10000000.     1.315E-13        4.852E-14!
!   30000000.     3.060E-13        1.126E-13!
```

Where the first column is the number of steps, the second is the absolute error $|e - sol|$ and the third column is the relative error $|\frac{e-sol}{e}|$.
If we plot, for example, the absolute error versus the number of steps, in a loglog plot, by the command

```
-->plot2d(result1(:,1),result1(:,3),logflag='ll',style=-4)

-->plot2d(result1(:,1),result1(:,3),logflag='ll',style=1)
```

We obtain the figure

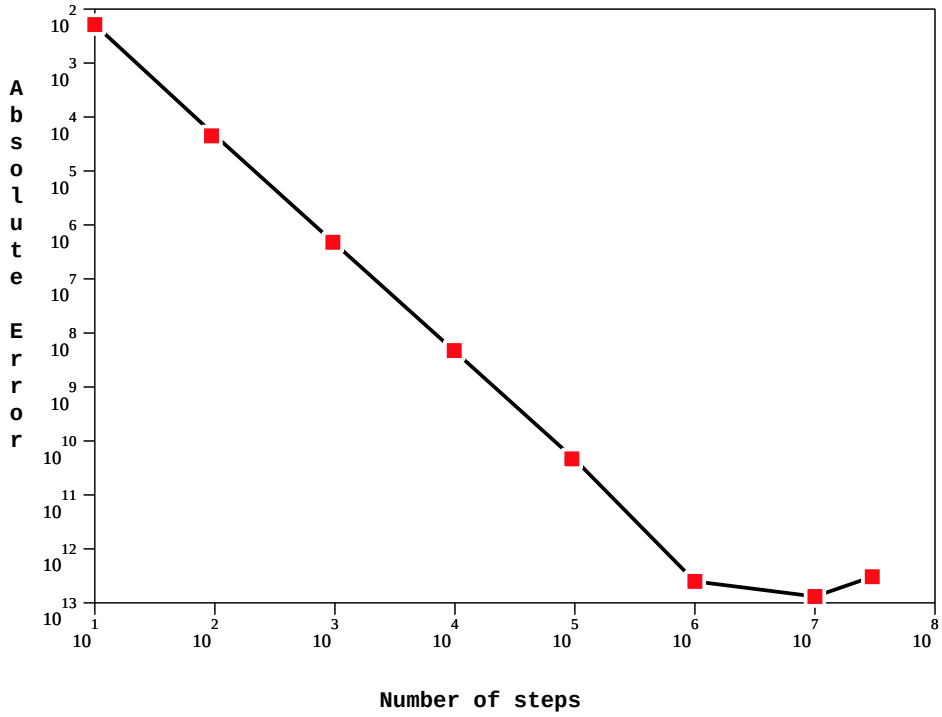

FIG. 2 – Effects of finite precision machine on accuracy

The reader is invited to make his own experiments. We draw the reader's attention to the fact that the computation can be lengthy. We use a double precision computer (as it is the case in this lecture, since Scilab use double precision) the effects appears for an order 2 one-step method around 10^7 steps.
For our test problem, the RK2 method is simply

$$x_{n+1} = x_n(1 + h + \frac{h^2}{2})$$

The local error is equivalent to the first neglected factor of the Taylor development $\frac{h^3}{6}x_n$, then on $[0, 1]$, we can estimate $le_n \leq \frac{e}{6}h^3$. Then in the formula (10) we can use $C = e/6$, which gives $M_2 \approx 10^{21}$.
In this experiment we have use the IEEE arithmetic precision of the computer. This implies that, if we want to see the phenomena, we must to use a large number of steps. This has been the case.

We can simulate simple precision or a even a less stringent precision. A famous anecdote is about C.B. Moler, a famous numerician, and creator and founder ofMATLAB, which was accustomed to describe the precision as "half–precision" and "full precision" rather than single and double precision. Here is a Scilab function which chop any number to the nearest number with t significant decimal digits

```
function c=chop10(x,t)
// chop10 rounds the elements of a matrix
// chop10(X) is the matrix obtained
// in rounding the elements of X
// to   t   significant  decimal digits. The rounding is toward
//the nearest number with t decimal digits.
// In other words the aritmetic is with unit
// roundoff 10^(-t)
//

y=abs(x)+ (x==0);
e=floor(log10(y)+1);
c=(10.^(e-t)).*round((10.^(t-e)).*x);
```

There a little trick, here, in this code. Note the line

```
y=abs(x)+ (x==0);
```

Which is intended to encompass the case $x = 0$.
Now we can rewrite the RK2 code, with chop10. This time, we must take the effect of the arithmetic into account, since $N * h \neq (t_f - t_0)$. At the end of the main loop, we add a step, to take care of this, to obtain a value at t_f. Note that we have included the function chop as a sub–function of RK2. Scilab permits this.

```
function sol=rk2hp(fct,t0,x0,tf,N)
// RK2 code with half precision (H.P.)
x0=x0(:)
n=length(x0)
h=(tf-t0)/N
h=chop10(h,8) // H.P.
x=x0;
```

```
t=t0;
k=1;

//main loop

while k< N +1

k=k+1
s1=feval(t,x,fct)
s1=s1(:)
s1=chop10(s1,8); // H.P.
x1= chop10(x+h*s1,8)   // Euler point
s2=feval(t+h,x1,fct) // slope at Euler point
s2=s2(:)
s2=chop10(s2,8) //H.P.
x=x+h*(s1+s2)/2
x=chop10(x,8) // H.P.
t=t+h
end

if t<tf then

h=tf-t;
s1=feval(t,x,fct);
s1=s1(:);
x1=x+h*s1;
s2=feval(t+h,x1,fct);
s2=s2(:);
x=x+h*(s1+s2)/2;
t=t+h;
end

sol=x
////////////////////////////
function c=chop10(x,t)
y=abs(x)+ (x==0);
e=floor(log10(y)+1);
c=(10.^(e-t)).*round((10.^(t-e)).*x);
```

Now we write a script for getting the results :

```
// script for getting the effects of roundoff
// in single precision
//note : the function chop10
// has been included in rk2hp
;getf("/Users/sallet/Documents/Scilab/rk2hp.sci");
;getf("/Users/sallet/Documents/Scilab/ode1.sci");
N=100:100:3000;
//
results=zeros(30,3);
results(:,1)=N';
//
for i=1:30,

sol=rk2hp(ode1,0,1,1,N(i));
results(i,2)=abs(%e-sol);
results(i,3)=abs(%e-sol)/%e;
end
```

We obtain

```
!   100.     0.0000456    0.0000168 !
!   200.     0.0000104    0.0000038 !
!   300.     0.0000050    0.0000018 !
!   400.     0.0000015    5.623E-07 !
!   500.     0.0000024    8.934E-07 !
!   600.     1.285E-07    4.726E-08 !
!   700.     0.0000018    6.527E-07 !
!   800.     3.285E-07    1.208E-07 !
!   900.     0.0000010    3.683E-07 !
!   1000.    9.285E-07    3.416E-07 !
!   1100.    5.285E-07    1.944E-07 !
!   1200.    0.0000035    0.0000013 !
!   1300.    0.0000039    0.0000014 !
!   1400.    9.121E-07    3.356E-07 !
```

```
!   1500.     0.0000023    8.566E-07 !
!   1600.     0.0000020    7.462E-07 !
!   1700.     0.0000012    4.380E-07 !
!   1800.     0.0000013    4.678E-07 !
!   1900.     0.0000016    5.991E-07 !
!   2000.     8.715E-07    3.206E-07 !
!   2100.     0.0000011    4.151E-07 !
!   2200.     4.987E-07    1.835E-07 !
!   2300.     0.0000030    0.0000011 !
!   2400.     0.0000011    4.151E-07 !
!   2500.     0.0000058    0.0000021 !
!   2600.     0.0000046    0.0000017 !
!   2700.     0.0000020    7.452E-07 !
!   2800.     0.0000028    0.0000010 !
!   2900.     0.0000044    0.0000016 !
!   3000.     0.0000030    0.0000011 !
```

This result gives the following plot for the absolute error (the relative error is given by the third column)

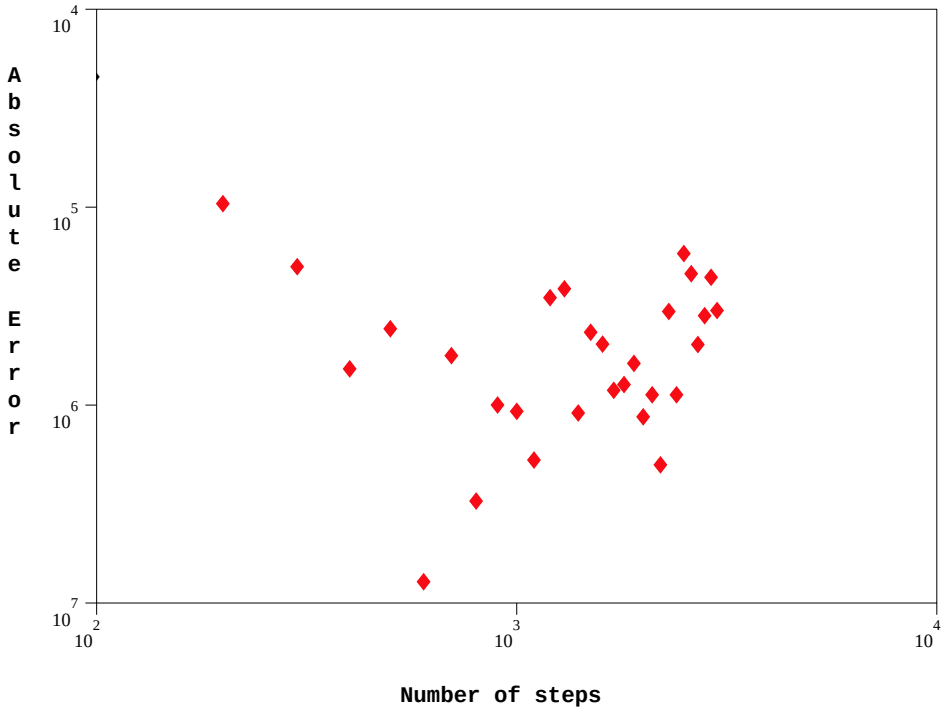

FIG. 3 – Effects of half precision on accuracy.

We observe that for a number of step under 1000 the slope of the error is 2 as predicted by the theory, but rapidly the situation deteriorates with roundoff errors.

4.5 Methods with memory

Methods with memory exploit the fact that when the step (t_n, x_n) is reached, we have at our disposition previously computed values, at mesh points, x_i and $f(t_i, x_i)$. An advantage of methods with memory is their high order of accuracy with just few evaluation of f. We recall that, in the evaluation of the method, it is considered that the highest cost in computing is the evaluation of f.
Methods with memory distinguishes between explicit and implicit methods. Implicit methods are used for a class of ODE named **stiff**. A number of theoretical question are still open. Many popular codes use variable order methods with memory. Practice shows that this codes are efficient. But understanding

the variation of order is an open question. Only recently theoretical justification have been presented (Shampine, Zhang 1990, Shampine 2002). To quote Shampine :

> our theoretical understanding of modern Adams and BDF codes that vary their order leaves much to be desired.

In this section we just give an idea of how works methods with memory. In modern codes the step is varying and the order of the method also. To be simple we shall consider (as for one-step methods) the case of constant step size. This is for pedagogical reasons. In theory changing the step size is ignored, but in practice it cannot. Varying the step size is important for the control of the local error. It is considered that changing step step size is equivalent to restarting . But starting a method with memory is a delicate thing. Usually in the early codes the starting procedure use a one-step method with an appropriate order. Another problem is the problem of stability of these methods, this is related to the step size. Methods with memory are more complex to analyze mathematically. However considering constant step size is still useful. General purpose solvers tend to work with constant step size, at least during a while. Special problems, for example, problems arising from semi-discretization in PDE are solved with constant step size.

4.5.1 Linear MultistepsMethods LMM

When the step size is constant the Adams and the BDF methods are included in a large class of formulas called **Linear Multipstep Methods**. These methods are defined by

Definition 4.6 (LMM methods) *:*
A LMM method of k-steps defines a rule to compute x_{n+1}, when k preceding steps $(x_n, x_{n-1}, \cdots, x_{n-k+1})$ are known . This rule is given by

$$\sum_{i=0}^{k} \alpha_i \, x_{n+1-i} = h \, \sum_{i=0}^{k} \beta_i \, f(t_{n+1-i}, x_{n+1-i}) \tag{13}$$

It is always supposed that $\alpha_0 \neq 0$, which gives x_{n+1}
If $\beta_0 = 0$ the method is explicit.
Otherwise, since x_{n+1} appears on the two sides of the equation ,the method is implicit. This means that x_{n+1} must be computed from the formula (13).

Three class of methods can be defined from LMMs. The Adams-Basforth explicit methods, the Adams-Moulton methods, and the Backward Differentiation Methods (BDF). The later methods are implicit .
The relation (13) can be rewritten
When $\beta_0 = 0$

$$x_{n+1} = -\sum_{i=1}^{k} \frac{\alpha_i}{\alpha_0} x_{n+1-i} + h \sum_{i=1}^{k} \frac{\beta_i}{\alpha_0} f(t_{n+1-i}, x_{n+1-i})$$

And when $\beta_0 \neq 0$

$$x_{n+1} = h\frac{\beta_0}{\alpha_0} f(t_{n+1}, x_{n+1}) - \sum_{i=1}^{k} \frac{\alpha_i}{\alpha_0} x_{n+1-i} + h \sum_{i=1}^{k} \frac{\beta_i}{\alpha_0} f(t_{n+1-i}, x_{n+1-i})$$

4.5.2 Adams-Basforth methods (AB)

The family of AB methods are described by the numerical scheme ABk

$$x_{n+1} = x_n + h \sum_{i=1}^{k} \beta^*_{k,i} f(t_{n+1-i}, x_{n+1-i}) \tag{14}$$

This class of method is quite simple and requires only one evaluation of f at each step, unlike RK methods.
The mathematical principle under Adams formulas is based on a general paradigm in numerical analysis : When you don't know a function approximate it with an interpolating polynomial, and use the polynomial in place of the function. We know approximations of f at (t_i, x_i), for k preceding steps. To lighten the notation we use

$$f(t_i, x_i) = f_i$$

The classical Cauchy problem

$$\begin{cases} \dot{x} & = & f(t,x) \\ x(t_n) & = & x_n \end{cases}$$

For passing from x_n to x_{n+1} this is equivalent to the equation

$$x_{n+1} = x_n + \int_{x_n}^{x_{n+1}} f(s, x(s))\, ds$$

There exists a unique $k-1$ degree polynomial $P_k(t)$ approximating f at the k-points (t_i, x_i). This polynomial can be expressed by

$$P_k(t) = \sum_{i=1}^{k} L_i(t)\, f_{n+1-i}$$

Where L_i the classical fundamental Lagrangian interpolating polynomal.
Replacing f by P_k under the integral, remembering that $h = x_{n+1} - x_n = x_{i+1} - x_i$, gives the formula (14). The coefficients $\beta^*_{k,i}$ are the results of this computation. For constant step size, it is not necessary to make the explicit computation. We can construct a table of Adams-Basforth coefficients.
We give the idea to construct the array , and compute the two first lines.
If we consider the polynomial function 1, the approximation of degree 0 is exact, and the formula must be exact. Then consider $\dot{x} = 1$, with $x(1) = 1$, the solution is $x(t) = t$, then integrate from 1 to $1 + h$, with AB1 formula, gives

$$1 + h = 1 + h\beta^*_{1,1}\, 1$$

Evidently $\beta^*_{1,1} = 1$ and AB1 is the forward Euler formula.
Considering AB2, with $\dot{x} = 2t$, $x(1) = 1$, with the 2 steps at 1 and $1 + h$, we get for the evaluation at $1 + 2h$, with the formula

$$x_{n+1} = x_n + h(\beta^*_{2,1} f_n + \beta^*_{2,2} f_{n-1})$$

The result

$$(1 + 2h)^2 = (1 + h)^2 + h\left(\beta^*_{2,1} 2(1 + h) + \beta^*_{2,2} 2\right)$$

This is a polynomial equation in h, by identification of the coefficients (this formula must be satisfied for any h), we obtain $1 + 2\beta^*_{2,1} = 4$ with $1 + \beta^*_{2,1} + \beta^*_{2,2} = 2$ which gives

$$\beta^*_{2,1} = \frac{3}{2} \quad \text{and} \quad \beta^*_{2,2} = -\frac{1}{2}$$

The reader is invited to establish the following tableau of Adams-Basfortth coefficients

k	$\beta^*_{k,1}$	$\beta^*_{k,2}$	$\beta^*_{k,3}$	$\beta^*_{k,4}$	$\beta^*_{k,5}$	$\beta^*_{k,6}$
1	1					
2	$\frac{3}{2}$	$-\frac{1}{2}$				
3	$\frac{23}{12}$	$-\frac{16}{12}$	$\frac{5}{12}$			
4	$\frac{55}{24}$	$-\frac{59}{24}$	$\frac{37}{24}$	$-\frac{9}{24}$		
5	$\frac{1901}{720}$	$-\frac{2774}{720}$	$\frac{2616}{720}$	$-\frac{1274}{720}$	$\frac{251}{720}$	
6	$\frac{4277}{1440}$	$-\frac{7923}{1440}$	$\frac{9982}{1440}$	$-\frac{7298}{1440}$	$\frac{2877}{1440}$	$-\frac{475}{1440}$

When the mesh spacing is not constant, the h_n appears explicitly in the formula and everything is more complicated. In particular it is necessary to compute the coefficient at each step. Techniques has been devised for doing this task efficiently.
It can be proved, as a consequence of the approximation of the integral, that

Proposition 4.2 (order of AB) *:*
The method ABk is of order k.

4.5.3 Adams-Moulton methods

The family of AM methods are described by the numerical scheme AMk

$$x_{n+1} = x_n \; + \; h \sum_{i=0}^{k-1} \beta_{k,i} \, f(t_{n+1-i}, x_{n+1-i}) \tag{15}$$

The AMk formula use $k-1$ preceding steps (because one step is at x_{n+1}, then one step less that the corresponding ABk method
The principle is the same as for AdamsBasforth but in this case the interpolation polynomial use k points, including (t_{n+1}, x_{n+1}). The k interpolating values are then

$$(f_{n+1}, f_n, \cdots, f_{n+2-k})$$

Accepting the value (implicit) f_{n+1}, exactly the same reasoning as for AB, gives an analogous formula, with f_{n+1} entering in the computation. This is an implicit method. The coefficient of the AM family can be constructed in the same manner as for the AB family, and we get a tableau of AM coefficients :

k	$\beta_{k,0}$	$\beta_{k,1}$	$\beta_{k,2}$	$\beta_{k,3}$	$\beta_{k,4}$	$\beta_{k,5}$
1	1					
2	$\frac{1}{2}$	$\frac{1}{2}$				
3	$\frac{5}{12}$	$\frac{8}{12}$	$-\frac{1}{12}$			
4	$\frac{9}{24}$	$\frac{19}{24}$	$-\frac{5}{24}$	$\frac{1}{24}$		
5	$\frac{251}{720}$	$\frac{646}{720}$	$-\frac{264}{720}$	$\frac{106}{720}$	$-\frac{19}{720}$	
6	$\frac{475}{1440}$	$\frac{1427}{1440}$	$-\frac{798}{1440}$	$\frac{482}{1440}$	$-\frac{173}{1440}$	$\frac{27}{1440}$

The method AM1, is known as Forward Euler formula

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1})$$

The method AM2 is known as the trapezoidal rule (explain why !), in the context of PDEs it is also called Crank-Nicholson method

$$x_{n+1} = x_n + \frac{h}{2}[f(t_n, x_n) + f(t_{n+1}, x_{n+1})$$

It can be proved, it is a consequence of the approximation of the integral, that

Proposition 4.3 (order of AM) *:*
The method AMk is of order k.

It can be shown that the AM formula is considerably more accurate that the AB formula of the same order, for moderately large order k.

4.5.4 BDF methods

The principle behind BDF is the same. We use also a polynomial interpolation but for a different task. Now we interpolate the solution itself, including the searched point x_{n+1} at the mesh point t_{n+1}. Using the $k+1$ values $x_{n+1}, x_n, \cdots, x_{n+1-k}$ at mesh points $t_{n+1}, t_n, \cdots, t_{n+1-k}$ we obtain a unique polynomial $P_k(t)$ of degree k. A way of approximate a derivative is to differentiate an interpolating polynomial. Using this, we collocate the ODE at t_{n+1} i.e.

$$P_k'(t_{n+1}) = f(t_{n+1}, P_k(t_{n+1})\,) = f(t_{n+1}, x_{n+1})$$

This relation is an implicit formula for x_{n+1}. If we write

$$P_k(t) = \sum_{i=0}^{k} L_i(t)\, x_{n+1-i}$$

The preceding relation becomes

$$\sum_{i=0}^{k} L_i'(t_{n+1})\, x_{n+1-i} = f(t_{n+1}, x_{n+1})$$

If we recall that

$$L_i(t) = \frac{\prod_{j\neq i}(t - x_{n+1-j})}{\prod_{j\neq i}(x_{n+1-i} - x_{n+1-j})}$$

It is then clear that for each index i, we can write

$$L_i(t) = \frac{1}{h} M_i(t)$$

Hence the final formula, with constant step for BDF is

$$\sum_{i=0}^{k} \alpha_i\, x_{n+1-i} = h f(t_{n+1}, x_{n+1}) \tag{16}$$

The BDF formulas are then LMMs methods.
It is easy to see that

Proposition 4.4 (order of BDF) *:*
The method BDFk is of order k.

For constant step an array of BDF coefficient can be computed with the same principle as for the building of the coefficient of Adams methods. There however a difference, the Adams formulas are based on interpolation of the derivative of the solution, the BDFs are based on the interpolation of solution values themselves.
let look at at example. Considering the ODE $\dot{x} = 3t^2$ with $x(1) = 1$, should gives the coefficients of BDF3.
Looking at the mesh $(1 + h, 1, 1 - h, 1 - 2h)$ give the relation

$$\alpha_0(1+h)^3 + \alpha_1 + \alpha_2(1-h)^3 + \alpha_3(1_2h)^3 = 3h(1+h)^2$$

This relation gives

$$\begin{cases} \alpha_0 - \alpha_2 - 8\alpha_3 = 3 \\ \alpha_0 + \alpha_2 + 4\alpha_3 = 2 \\ \alpha_0 - \alpha_2 - 2\alpha_3 = 1 \\ \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3 = 0 \end{cases}$$

From which we obtain

$$\alpha_0 = \frac{11}{6} \quad \alpha_1 = -3 \quad \alpha_2 = \frac{3}{2} \quad \alpha_3 = -\frac{1}{3}$$

The array for BDF is

k	$\alpha_{k,0}$	$\alpha_{k,1}$	$\alpha_{k,2}$	$\alpha_{k,3}$	$\alpha_{k,4}$	$\alpha_{k,5}$	
1	1	-1					
2	$\frac{3}{2}$	-2	$\frac{1}{2}$				
3	$\frac{11}{6}$	-3	$\frac{3}{2}$	$-\frac{1}{3}$			
4	$\frac{25}{12}$	-4	3	$-\frac{4}{3}$	$\frac{1}{4}$		
5	$\frac{137}{60}$	-5	5	$-\frac{10}{3}$	$\frac{5}{4}$	$-\frac{1}{5}$	
6	$\frac{147}{60}$	-6	$\frac{15}{12}$	$-\frac{20}{3}$	$\frac{15}{4}$	$-\frac{6}{5}$	$\frac{1}{6}$

The name Backward difference comes from the fact that expressed in Backward Differences the interpolating polynomial has a simple form. The Backward differences of a function are defined inductively, $\nabla^{n+1} f$

$$\nabla f(t) = f(t) - f(t-h)$$

$$\nabla^{n+1} f(t) = \nabla(\nabla^n f(t))$$

If we set $x_{n+1} = x(t_{n+1})$ for interpolation, the polynomial is

$$P_k(t) = x_{n+1} + \sum_{i=1}^{k} \frac{(t-t_{n+1}) \cdots (t-t_{n+2-i})}{h^i i!} \nabla^i x_{n+1}$$

Then the BDF formula takes the simple form

$$\sum_{i=1}^{k} \frac{1}{i} \nabla^i x_{n+1} = h f(t_{n+1}, x_{n+1})$$

4.5.5 Implicit methods and PECE

The reader at this point can legitimately ask the question : why using implicit methods, since an implicit method requires to solve an equation, and then add some computation overhead ?
The answer is multiple. A first answer is in stability. We shall compare two methods on a simple stable equation :

$$\begin{cases} \dot{x} = \alpha x \\ x(0) = 1 \end{cases}$$

With $\alpha < 0$.
The simple method RK1=AB1 gives rise to the sequence

$$x_{n+1} = (1 + \alpha h)^{n+1}$$

This sequence converge iff $\mid 1 + \alpha h \mid < 1$, which implies $\mid h \mid < \frac{2}{|\alpha|}$
On contrary the forward Euler method AM1=BDF1 gives

$$x_{n+1} = \frac{1}{1 - \alpha h} x_n$$

This sequence is always convergent for any $h > 0$. If we accept complex values for α the stability region is all the left half of the complex plane. Check that this is also true for the AM2 method. We shall study in more details this concept in the next section.
Stability restricts the step size of Backward Euler to $\mid h \mid < \frac{2}{|\alpha|}$. When α takes great values, the forward Euler becomes interesting despite the expense of solving an implicit equation.
The step size is generally reduced to get the accuracy desired or to keep the computation stable. The popular implicit methods used in modern codes, are much more stable than the corresponding explicit methods. If stability reduces sufficiently the step size for an explicit method, forcing to take a great number of steps, an implicit method can be more cheap in computation time and efficiency.
In stiff problems, with large Lipschitz constants, a highly stable implicit method is a good choice.
A second answer is in comparing AB and AM methods. AM are much more accurate than the corresponding AB method. The AM methods permits bigger step size and in fact compensate for the cost of solving an implicit equation.
We shall takes a brief look on the implicit equation. All the methods considered here are all LMMs and, when implicit, can be written

$$x_{n+1} = h\frac{\beta_0}{\alpha_0} f(t_{n+1}, x_{n+1}) - \sum_{i=1}^{k} \frac{\alpha_i}{\alpha_0} x_{n+1-i} + h \sum_{i=1}^{k} \frac{\beta_i}{\alpha_0} f(t_{n+1-i}, x_{n+1-i})$$

Or in a shorter form

$$x_{n+1} = hC_1F(x_{n+1}) + hC_2 + C_3$$

The function $\Phi(x) = hC_1F(x) + fC_2 + C_3$ is clearly Lipschitz with constant $\mid h \mid C_1L$, where L is the Lipschitz constant of f the RHS of the ODE. Then by a classical fixed point theorem, the implicit equation has a unique solution if $\mid h \mid C_1L < 1$ or equivalently

$$\mid h \mid < \frac{1}{C_1L}$$

The unique solution is the limit of the iterated sequence $z_{n+1} = \Phi(z_n)$. The convergence rate is of order of h

If h is small enough a few iterations will suffice. For example it can be required that the estimated $\mid h \mid C_1 L$ is less that 0.1 , or that three iterations are enough .... Since x_{n+1} is not expected to be the exact solution there is no point in computing it more accurately than necessary. We get an estimation of the solution of the implicit equation, this estimate is called a prediction. The evaluation of f is called an evaluation. This brings us to another type of methods. Namely the predictor corrector methods.
Another methods are **predictor corrector** methods or **P.E.C.E.** methods. The idea is to obtain explicitly a first approximation (prediction) px_{n+1} of x_{n+1}. Then we can predict pf_{n+1} i.e. $f(t_{n+1}, px_{n+1})$. A substitution of f_{n+1} by pf_{n+1} in an implicit formula , give a new value for x_{n+1} said corrected value, with this corrected value f_{n+1} can be evaluated ...

$$\left\{\begin{array}{ll} \text{Prediction} & px_{n+1} = explicit \;\; formula \\ \text{Evaluation} & pf_{n+1} = f(t_{n+1}, px_{n+1}) \\ \text{Correction} & x_{n+1} = implicit(pf_{n+1}) \\ \text{Evaluaton} & f_{n+1} = f(t_{n+1}, x_{n+1}) \end{array}\right. \tag{17}$$

Let apply this scheme to an example with AB1 and AM2 methods (look at the corresponding arrays)

$$\left\{\begin{array}{ll} \text{Prediction AB1} & px_{n+1} = x_n + h\, f(t_n, x_n) \\ \text{Evaluation} & pf_{n+1} = f(t_{n+1}, px_{n+1}) \\ \text{Correction AM2} & x_{n+1} = x_n + \frac{1}{2}[f_n + pf_{n+1}] \\ \text{Evaluaton} & f_{n+1} = f(t_{n+1}, x_{n+1}) \end{array}\right. \tag{18}$$

We rediscover in another disguise the Heun method RK2.
It is simple to look at the local error for a PECE method. The reader is invited to do so. Check that the influence of the predictor method is less than the corrector method. Then for a k order predictor is wise to choose a $k+1$ corrector method. Prove that for a order p^* predictor method and an order p corrector method , the PECE associated method is of order $\min(p^*+1, p)$.
A PECE is an explicit method. As we have seen AB1-AM2 is RK2 method. Check the stability of this method and verify that this method has a finite stability region.

4.5.6 Stability region of a method

:
The classic differential equation

$$\begin{cases} \dot{x} = \lambda x \\ x(0) = 1 \\ \Re(\lambda) < 0 \end{cases} \tag{19}$$

Is called Dahlquist test equation.
When applied to this test equation a discrete method gives a sequence defined by induction. To be more precise, since all the method encountered in this notes can be put in LMM form, we look at this expression for a LMM method. using the definition (13) of a LMM formula we obtain, when applied to the Dahlquist equation (19), the recurrence formula :

$$\sum_{i=0}^{k} (\alpha_i - \beta_i h\lambda) x_{n+1-i} = 0$$

We set $\mu = h\lambda$
It is well known that to this linear recurrent sequence is associated a polynomial

$$(\alpha_0 - \beta_0\mu)\zeta^{k+1} + \cdots + (\alpha_i - \beta_i \mu)\zeta^{k-i+1} + \cdots + (\alpha_k - \beta_k \mu) = 0$$

Since this formula is linear in μ the right hand side can be written

$$R(\mu) = \rho(\zeta) - \mu\,\sigma(\zeta)$$

The polynomial ρ, with coefficient α_i is known as the first characteristic polynomial of the method, The polynomial σ, with coefficient β_i is known as the second characteristic polynomial of the method.
From recurrence formulas, it is well known that a recurrence formula is convergent if the simple root of the polynomial $R(\mu)$ are contained in the closed complex disk U of radius 1, and the multiple roots are in the open disk.

Definition 4.7 (Stability Domain) *:*
The set S

$$S = \{\mu \in \mathbb{C} \mid roots(R(\mu) \subset U\}$$

is called the stability region of the LMM method.
If $S \subset \mathbb{C}^-$ the method is said to be A-stable.

characterization of the stability domain We have

$$\mu = \frac{\rho(\zeta)}{\sigma(\zeta)}$$

The boundary of S is then the image of the unit circle by the function $H(\zeta) = \frac{\rho(\zeta)}{\sigma(\zeta)}$. The stability region, whenever it is not empty, must lie on the left of this boundary. With Scilab it is straightforward to plot the stability region and obtain the classical pictures of the book (HW, S ; B)
For example for Adams-Basforth formulas

$$H(\zeta) = \frac{\zeta^{k+1} - \zeta^k}{\beta^*_{k,1}\,\zeta^k + \cdots + \beta^*_{k,k}} = \frac{\zeta - 1}{\beta^*_{k,1} + \cdots + \beta^*_{k,k}\,\zeta^{-k}}$$

For BDF formulas

$$\mu = H(\zeta) = \frac{\rho(\zeta)}{\zeta^{n+1}}$$

In Scilab, using the vectorization properties

```
-->z=exp(%i*%pi*linspace(0,200,100);
-->r=z-1;
-->-->s2=(3-(1)./z)/2;
-->w2=(r./s2)';
-->plot2d(real(w2),imag(w2),1)
-->s2=(23-(16)./z+(5)./z.^2)/12;
-->w3=(r./s2)';
-->plot(ream(w3),imag(w3),2)
-->s4=(55-(59)./z+(37)./z^2-(9)./z.^3)/24;
-->w4=(r./s4)';
-->plot(real(w4),imag(w4),3)
-->s5=(1901-(2774)./z+(2616)./z.^2-(1274)./z.^3+...
-->(251)./z.^4)/720;
 -->w5=(r./s5)';
 -->plot2d(real(w5),imag(w5),4)
```

Here is **A WARNING**
Pay attention at the code. We write

```
(16)./z
```

And not

```
16./z
```

The first code is clear, and Scilab interpret like $ones(z)$ ./ z . The second is interpreted like the real 1. follow by a slash and the matrix z. In Scilab $x = A\,/B$ is the solution of $x * B = A$. Then Scilab compute the solution of the matrix equation $x * z = 1$..
The two codes gives very different results. We are looking for the element wise operation ./, this is the reason of the parenthesis. We could also have written, using space

```
1  ./  z
```

However we prefer to use parenthesis as a warning for the reader.
We obtain the following pictures

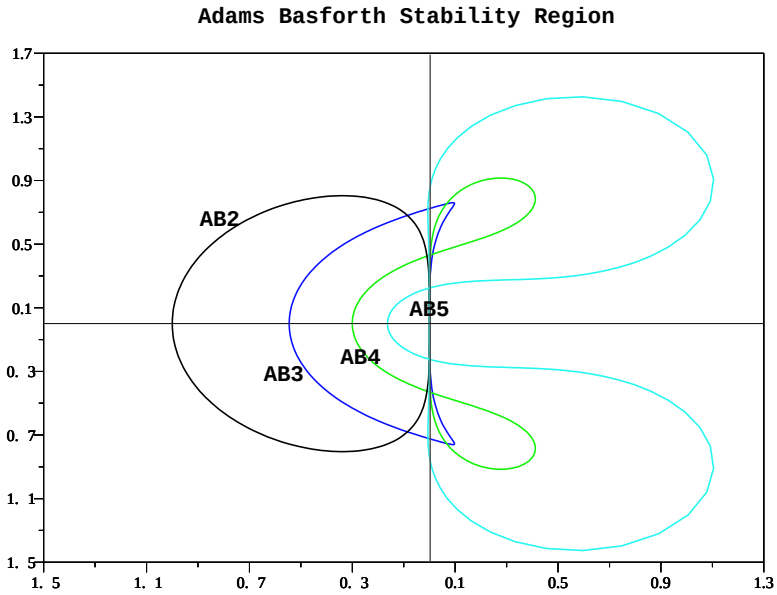

FIG. 4 – Adams-Basforth region of stability

The reader is invited to obtain the following figures for Adams-Moulton and BDF formulas

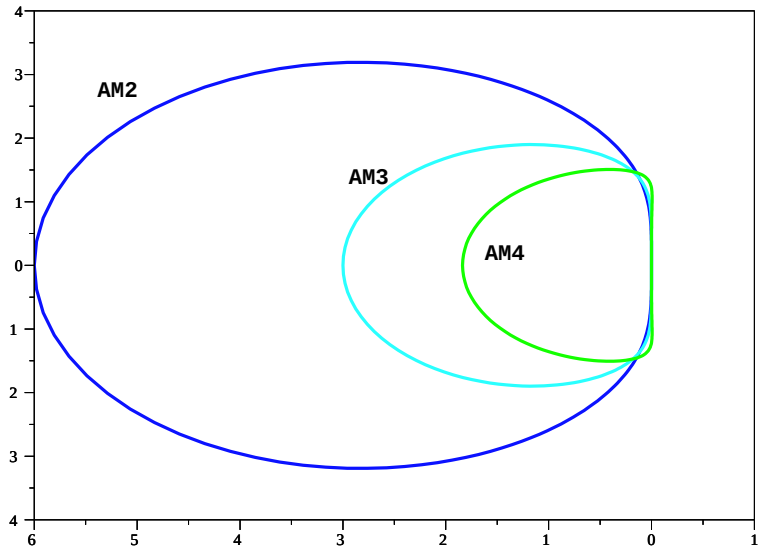

FIG. 5 – Adams-Moulton region of stability

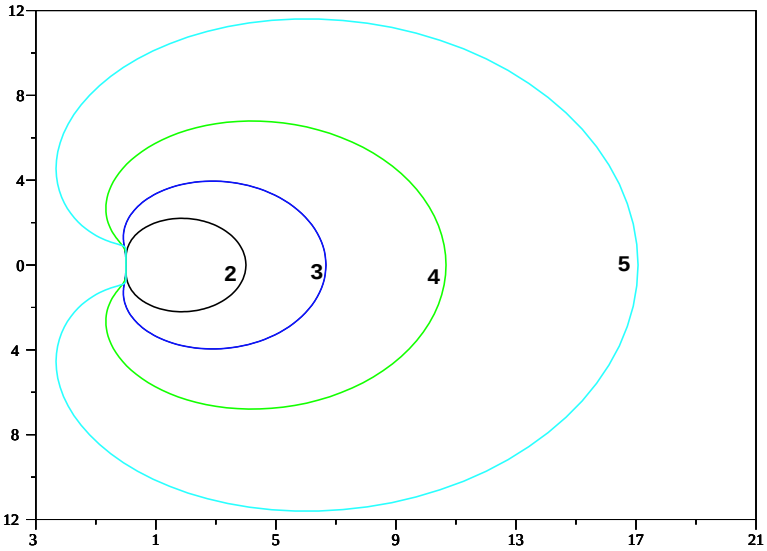

FIG. 6 – BDF region of stability, on the left of the boundaries

It can be shown that for RK methods of order p the polynomial $R(\zeta)$ is

$$R(\zeta) = 1 + \zeta + \frac{\zeta}{2} + \cdots + \frac{\zeta^k}{k!}$$

.

4.5.7 Implicit methods, stiff equations, implementation

In this lectures we have often spoken of **stiff equations**. Stiffness is a complicated idea that is difficult to define. We have seen that for solving ODE numerically the equation must be stable. This for example implies that the Jacobian of the RHS must be a stable matrix. But with the Jacobian can be associated the Lipschitz constant, related with the norm of the Jacobian. Roughly a stiff problem is the combination of stability, with big eigenvalues (negative real part) of the Jacobian , implying a big norm, hence a big Lipschitz constant. This imply that the process described by the ODE contains components operating on different time scale.
For a stiff problem the constant appearing in the upper bound hL , with L the Lipschitz constant , the step size giving the desired accuracy, since L is big, must be chosen very small for assuring stability. A working definition for stiffness could be

> Equations where certains implicit methods, in particular BDF, perform better, usually tremendously better, than explicit ones

To implement an implicit method, we have to solve some implicit equation in x

$$x = hC_1F(x) + hC_2 + C_3$$

Where $F(x) = f(t_{n+1}, x)$.
We have proposed a first solution by iteration procedure. If the norm of the Jacobian of f is big, this must imply for convergence a small step size h. Another method is Newton's method, which define a sequence converging to the solution. We set

$$F(x) = x - (hC_1F(x) + hC_2 + C_3)$$

The sequence is defined by induction, $x^{(0)}$ is the initial guess, and

$$x^{(k+1)} = x^{(k)} - \left[\frac{\partial F}{\partial x}(x^{(k)}\right]^{-1} F(x^{(k)})$$

With

$$\left[\frac{\partial F}{\partial x}(x)\right] = I - hC_1\frac{\partial f(t_{n+1},x)}{\partial x}$$

Each Newton iteration requires solving a linear system with a matrix related to the Jacobian of the system. For any step size, Newton's method converge if the initial guess is good enough.
To implement Newton's method, or some other variant, the matrix of partial derivatives of the RHS is required. The user is encouraged to provide directly the Jacobian. If the Jacobian is not provided, the solver must compute numerically the Jacobian with all the accompanying disadvantages.

5 SCILAB solvers

In the simplest use of Scilab to solve ODE, all you must do is to tell to Scilab what Cauchy problem is to be solved. That is, you must provide a function that evaluate $f(t,x)$, the initial conditions (x_0,t_0) and the values where you want the solution to be evaluated.

5.1 Using Scilab Solvers, Elementary use

The syntax is

```
x=ode(x0,t0,T,function);
```

The third input is a vector T. The components of T are the times at which you want the solution computed.
The 4th input ' 'function" is a function which compute the RHS of the ODE, $f(t,x)$.

A WARNING : THE FUNCTION MUST ALWAYS HAVE IN ITS INPUT THE EXPRESSION T , even if the ODE is autonomous.

That is, if given in a Scilab file, the code of the right hand side must begin as

```
function  xdot =RHS(t,x)
// the following lines are the lines of code that compute
// f(t,x)
```

The time t must appears in the first line.
This file must be saved as RHS.sci and called from the window command line by a getf ("../RHS.sci ")
Or as an alternative you can code as an inline function, directly from the window command :

```
-->deff('xdot=RHS(t,x)','xdot=t^2*x')

-->RHS
 RHS  =

[xdot]=RHS(t,x)
```

You can check that the function RHS exists, in typing RHS, and Scilab answer give the input and the output.
If you are interested in plotting the solution, you must get a mesh of points on the integration interval

```
-->T=linspace(0,1,100);

-->x=ode(1,0,T',RHS);

-->xbasc()

-->plot2d(T',x')
```

you obtain

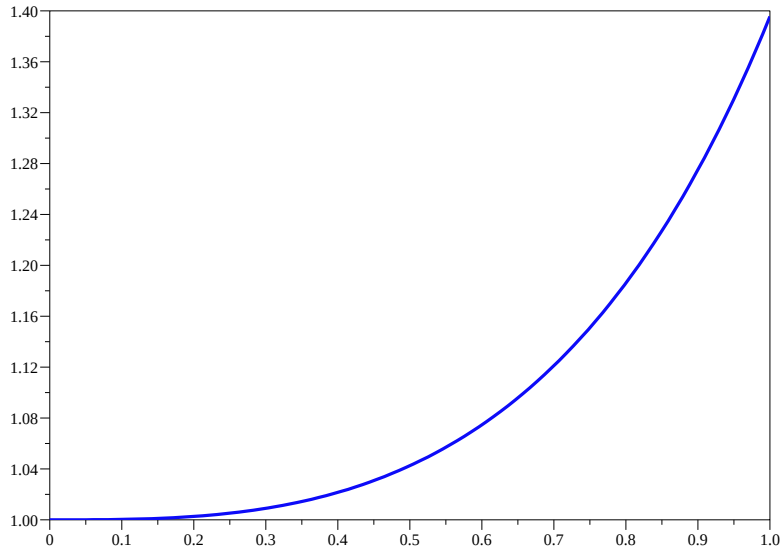

FIG. 7 – Plot of a solution

When the function is simple, it can be entered directly from the command line. But the solvers of Scilab are intended to be used with large and complicated systems. In this case the equation is provided as Scilab function.
We shall give another example

Example 5.1 (Euler rigid body equations) *:*
We consider the following system

$$\begin{cases} \dot{x}_1 = x_2\, x_3 \\ \dot{x}_2 = -x_3\, x_1 \\ \dot{x}_3 = -0.51\; x_1\, x_2 \end{cases}$$

With I.V.

$$[x_1(0); x_2(0); x_3(0)] = [0; 1; 1]$$

This example illustrates the solution of a standard test problem proposed by Krogh for solvers intended for nonstiff problems. It is studied in the book of Shampine and Gordon. This problem has a known solution in term of Jacobi special function. We give some preliminaries.
We define the so-called incomplete elliptic integral of the first kind (Abramowitz and Stegun) :

$$F(\Phi, m) = \int_0^{\Phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}} = \int_0^{\sin(\Phi)} \frac{dt}{\sqrt{(1-t^2)(1-mt^2}}$$

Since F is clearly a monotone increasing function of Φ, it admits an inverse function, called the amplitude.

$$u = F(\Phi, m) \quad \Phi = am(u, m)$$

Definition 5.1 (Jacobi ellliptic functions) *:*
The Jacobi elliptic functions are given respectively by (see Abramowitz and Stegun)

$$\begin{cases} sn(u,m) = \sin(\Phi) = \sin(am(u,m)) \\ cn(u,m) = \cos(\Phi) = \cos(am(u,m)) \\ dn(u,m) = \sqrt{1 - sn^2(u,m)} \end{cases}$$

It is now a simple exercise to check that

$$\begin{cases} \frac{d}{dt} sn(u,m) = cn(u,m)\, dn(u,m) \\ \\ \frac{d}{dt} cn(u,m) = -sn(u,m)\, dn(u,m) \\ \\ \frac{d}{dt} dn(u,m) = -m\, sn(u,m)\, cn(u,m) \end{cases}$$

With $sn(0,m) = 0$, $cn(0,m) = dn(0,m) = 1$.
We immediately see that the solution of the Euler equations, with the I.V. given, are

$$\begin{cases} x_1(t) = sn(t,m) \\ x_2(t) = cn(t,m) \\ x_3(t) = dn(t,m) \end{cases}$$

With $m = 0.51$. We readily see that the solutions are periodic with period $4\,F(\frac{\pi}{2}, m)$, which is known as the complete elliptic integral of the first kind $4\,K(m) = 4\,F(\frac{\pi}{2}, m)$.

In Scilab the function $sn(u,m)$ is built-in and called by

```
%sn(u,m)
```

. The inverse of the Jacobi sinus is

```
%asn(u,m)
```

Then the amplitude is given by $am(u,m) = asin(\%sn(u,m)$, and the incomplete elliptic integral is given by $F(\Phi,m) = \%asn(sin(\Phi))$. The complete integral is given by

```
%k(m)
```

With this preliminaries we can make some experiments. First we get the figure of Gordon-Shampine book :
Coding the rigid body equations

```
function xdot=rigidEuler(t,x)
xdot=[x(2)*x(3);-x(3)*x(1);-0.51*x(1)*x(2)]
```

Computing and plotting the solution :

```
-->;getf("/Users/sallet/Documents/Scilab/rigidEuler.sci");

--> x0=[0,1,1]'; t0=0; T=linspace(0,12,1200);

--> m=0.51;K=% k(m);

-->sol=ode(x0,t0,T,rigidEuler);

-->plot2d(T',sol')
```

Note the transpose for plotting. The solver gives the solution in n rows, n the dimension of the system, and N columns the number of time points asked by the user. The function plot use column vectors.
We obtain the figure

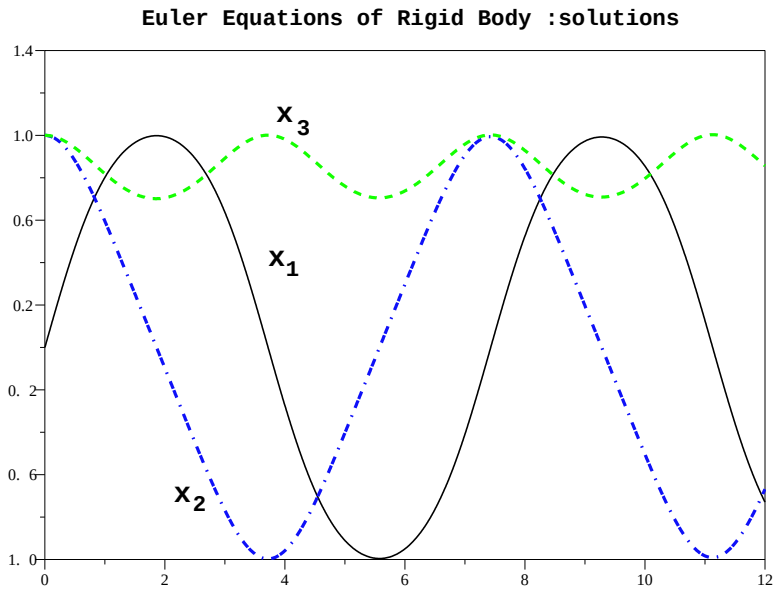

FIG. 8 – Solution of the movement of a rigid body

We can now appreciate the quality of the solution.
The theoretical solution is

$$(sn(t,m), cn(t,m), dn(t,m))$$

As we have seen we have access in Scilab to sn, and then to $dn(t,m) = \sqrt{1 - m\, sn(t,m)^2}$. We have to use an addition formula to obtain cn from sn. We have

$$sn(u+v) = \frac{sn(u)\, cn(v)\, dn(v) + sn(v)\, cn(u)\, dn(u)}{1 - m\, sn(u)^2\, sn(v)^2}$$

We obtain from this relation (analogous to $\sin(x + \pi/2)$)

$$cn(x) = sn(x + K)\, dn(x)$$

Using the vectorized properties of Scilab (look at the operations $.*$ and $.)$

```
-->theorsol=[\%sn(T,m); ...
```

```
-->% sn(T+K,m).*sqrt(1-m*(% sn(T,m)).^2);...
-->sqrt(1-m*%sn(T,m).^2)];
-->norm(abs(theorsol-sol))/norm(theorsol)
 ans  =

    0.0244019
```

We obtain an absolute error of 2% for all the interval. Don't forget that we integrates on $[0, 12]$. If we only look after a time interval of $1s$, the result are better :

```
-->(%sn(1,m)-sol(1,101))/%sn(1,m)
 ans  =

    0.0011839
```

And at the end we have also

```
--> norm(theorsol(:,$)-sol(:,$))/norm(theorsol)
 ans  =

    0.0015027
```

Which is quite correct.
We shall see how to improve these results from the accuracy point of view.

5.2 More on Scilab solvers

The syntax ode is an interface to various solvers, in particular to ODEPACK. The library ODEPACK can be found in Netlib. They are variations of Hindsmarch's LSODE. LSODE implements Adams-Moulton formulas for non stiff problems and BDF's formulas. When the user says nothing, the solver selects between stiff and non stiff methods. LSODE is a successor of the code GEAR (Hindsmarch), which is itself a revised version of the seminal code DIFSUB of Gear (1971) based on BDF's formulas.
When the user does no precise the solver used, ODE call LSODA solver of the ODEPACK package. Lsodar is the livermore solver for ordinary differential equations, with automatic method switching for stiff and nonstiff problems, and with root-finding. This version has been modified by scilab group on Feb 97 following Dr Hindmarsh direction.

5.2.1 Syntax

```
x=ode('method',x0,t0,T,rtol,atol, function,Jacobian)
```

Method :
The user can choose between

1. 'adams'
2. 'Stiff'
3. 'RKF'
4. 'RK'.

The method 'Adams' use Adams-Moulton formulas. The maximum order for the formulas is, by default, 12.
The method 'Stiff' uses a BDF method. The maximum order is 5
The method 'RKF' uses the program of Shampine and Watts (1976) based on the RKF embedded formula of Fehlberg of example (4.12)
An adaptative Runge-Kutta formula of order 4

Tolerance : rtol, atol :
Tolerance for the error can be prescribed by the user. At each step the solver produce an approximation $\bar{x}$ at mesh points t and an estimation of the error est is evaluated. If the user provide a n-vector $rtol$ of the dimension of the system, and another n-vector atol, then the solver will try to satisfy

$$est(i) \leq rtol(i) \mid \bar{x}(i) \mid + atol(i)$$

For any index $i = 1 : n$.
This inequality defines a mixed error control. If $atol = 0$ then it corresponds to a pure relative error control, if $rtol = 0$ i corresponds to a pure absolute error control.
When $\bar{x}(i)$ is big enough, the absolute error test can lead to impossible requirements : The inequality

$$\mid \bar{x}(i) - x(i) \mid \leq est(i) \leq atol(i)$$

gives, if $x(i)$ is big enough

$$\frac{\mid \bar{x}(i) - x(i) \mid}{\mid x(i) \mid} \leq \frac{atol(i)}{\mid x(i) \mid} \leq eps$$

This tolerance ask for a relative error smaller than the precision machine, which is impossible.
In a similar way, absolute error control can lead also to troubles, if $x(i)$ becomes small, for example, if $\mid x(i) \mid \leq atol(i)$, with a pure absolute control any number smaller than $atol(i)$ will pass the test of error control. The norm of $x(i)$ can be small but theses values can have influence on the global solution of the ODE, or they can later grow and then must be taken in account. The scale of the problem must be reflected in the tolerance. We shall give examples later, for example section(5.4.3).
If $rtol$ and $atol$ are not given as vectors, but as scalars, Scilab interprets $rtol$ as $rtol = rtol * ones(1, n)$, and similarly for $atol$.
Default values for $rtol$ and $atol$ are respectively

$$atol = 10^{-7} \quad \text{and} \quad rtol = 10^{-5}$$

Jacobian :
We have seen why the Jacobian is useful when the problems are stiff in section (4.5.7).
When the problems are stiff, it is better to give the Jacobian of the RHS of the ODE. You must give Jacobian as a Scilab function, and the syntax must include t and the state x in the beginning of the code for the Jacobian. This is exactly the same rule as for the RHS of the ODE. For a Scilab file function :

```
function  J=jacobian (t,x)
```

The same rule must be respected for an "inline " function.
When the problem is stiff you are encouraged to provide the Jacobian. If this is not feasible the solver will compute internally by finite differences, i.e. the solver compute the Jacobian numerically. It is now easy to understand why providing Jacobian is interesting for the computation overhead and also for the accuracy.
Before illustrating with examples the preceding notions we describe how to change options for the ODE solvers in Scilab.

5.3 Options for ODE

Some options are defined by default. A vector must be created to modify these options If created this vector is of length 12. At the beginning of the

session this vector does not exist.
The ODE function checks if this variable exists and in this case it uses it. If this variable does not exist ODE uses default values. For using default values you have to clear this variable.
To create it you must execute the command line displayed by odeoptions.

```
--> % ODEOPTIONS=[itask,tcrit,h0,hmax,hmin,jactyp,...
--> mxstep,maxordn,maxords,ixpr,ml,mu]
```

A Warning : the syntax is exactly as written, particularly you must write with capital letters. Scilab makes the distinction. If you write % odeoptions , ODE will use the default values !
The default values are

```
[1,0,0,%inf,0,2,500,12,5,0,-1,-1];
```

The meaning of the elements is described below.

5.3.1 itask

Default value 1
The values of "itask" are 1 to 5

- itask =1 : normal computation at specified times
- itask=2 : computation at mesh points (given in first row of output of ode)
- itask=3 : one step at one internal mesh point and return
- itask= 4 : normal computation without overshooting tcrit
- itask=5 : one step, without passing tcrit, and return

5.3.2 tcrit

Default value 0
tcrit assumes itask equals 4 or 5, described above.

5.3.3 h0

Default value 0
`h0` first step tried
It Can be useful to modify `h0`, for event location and root finding.

5.3.4 hmax

Default value ∞ (% inf in Scilab)
`hmax` max step size
It Can be useful to modify `hmax`, for event location and root finding.

5.3.5 hmin

Default value 0
`hmin` min step size
It Can be useful to modify `hmin`, for event location and root finding.

5.3.6 jactype

Default value 2

- jactype= 0 : functional iterations, no jacobian used (”adams” or ”stiff” only)
- jactype= 1 : user-supplied full jacobian
- jactype= 2 : internally generated full jacobian
- jactype= 3 : internally generated diagonal jacobian (”adams” or ”stiff” only)
- jactype= 4 : user-supplied banded jacobian (see ml and mu below)
- jactype = 5 : internally generated banded jacobian (see ml and mu below)

5.3.7 mxstep

Default value 500
`mxstep` maximum number of steps allowed.

5.3.8 maxordn

`maxordn` maximum non-stiff order allowed, at most 12

5.3.9 maxords

`maxords` maximum stiff order allowed, at most 5

5.3.10 ixpr

Default value 0
`ixpr` print level, 0 or 1

5.3.11 ml,mu

`ml,mu` .If jactype equals 4 or 5, `ml` and `mu` are the lower and upper half-bandwidths of the banded jacobian : the band is the lower band of the i,j's with i-ml ¡= j and the upper band with j -mu ¡=i. (mu as for upper, ml as for lower ...)

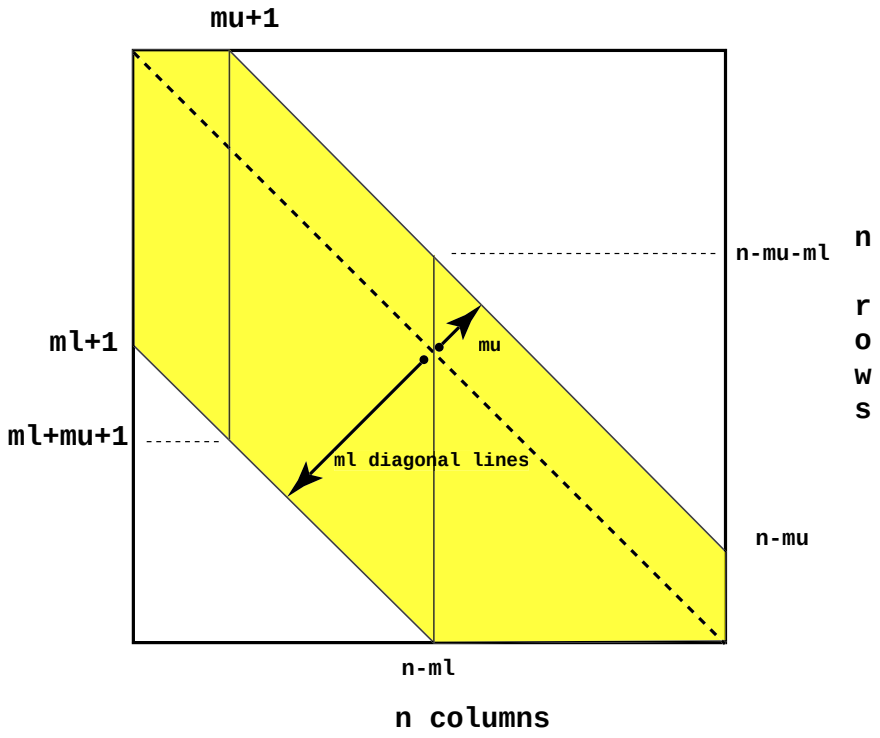

FIG. 9 – A banded Jacobian

If jactype equals 4 the jacobian function must return a matrix J which is ml+mu+1 x ny (where ny=dim of x in xdot=f(t,x)) , such that column 1 of J is made of mu zeros followed by df1/dx1, df2/dx1, df3/dx1, ... (1+ml

possibly non-zero entries) , column 2 is made of mu-1 zeros followed by df1/dx2, df2/dx2, etc
This is summarized by the following sketch :

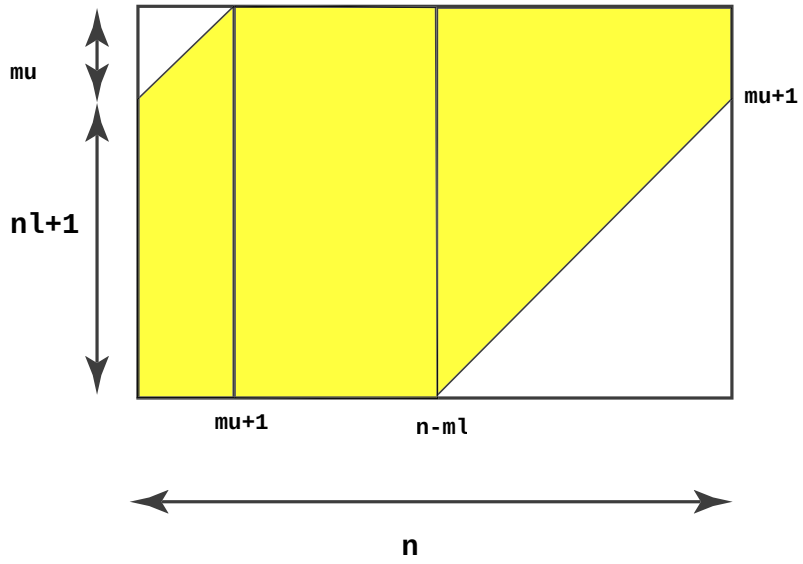

FIG. 10 – The matrix given to Scilab fo banded Jacobian

To be more precise and justify the preceding sketch, let a banded (mu,ml) Jacobian , with entries $J(i,j)$

$$J(i,j) = \left(\frac{\partial f_i}{\partial x_j}\right)$$

This express that the function f_i depends only of the variables

$$(x_{i-ml}, x_{i-ml+1}, \cdots, x_{i+mu})$$

Or differently that the variable x_j occurs only in the functions

$$(f_{j-ml}, f_{j-ml+1}, \cdots, f_{j+mu})$$

When the Jacobian is not banded if Jactype=1, you store the Jacobian in a $n \times n$ matrix J as defined before.

When jactype=2, and you have given values in $\%ODEOPTIONS$, to ml and mu , with $0 \leq ml, mu \leq n-1$, Scilab is waiting to store the Jacobian in a $ml + mu + 1 \times n$ matrix. In the column j of this matrix is loaded

$$[f_{j-ml}; , f_{j-ml+1}; \cdots ; f_{j+ml-1}; f_{j+ml}]$$

This equivalent to say that $\frac{\partial f_i}{\partial x_j}$ is loaded in $J(i-j+mu+1, j)$. Since we have $-mu \leq i-j \leq ml$, the number of line is then between 1 and $ml + mu + 1$. You can check that this give the preceding picture.
We shall see an example of this option, in the example Brusselator in section (5.4.4).

5.4 Experimentation

There exists test problems in the litterature. For example Hull [1972] et al, Krogh, Enright et al [1975] Hairer &Wanner . We shall use some of these test problems.

5.4.1 Two body problem

We use the problem D5 of Hull

$$\begin{cases} \ddot{x} = -\frac{x}{(x^2+y^2)^{3/2}} & \\ \ddot{y} = -\frac{y}{(x^2+y^2)^{3/2}} & \\ x(0) = 1-e & \dot{x} = 0 \\ y(0) = 0 & \dot{y} = \sqrt{\frac{1+e}{1-e}} \end{cases} \tag{20}$$

The solution is an ellipse of eccentricity e. The solution is

$$x(t) = \cos(u) - e \; y(t) = \sqrt{1-e^2}\sin(u)$$

Where u is the solution of the Kepler's equation $u - e\sin(u) = t$.
We shall integrates this equation for $e = 0.9$ and find the solution for $tf = 20$. We must solve the Kepler's equation, for $tf = 20$, i.e. solve

$$u - tf - e\sin(u) = 0$$

It is clear that this equation has always a unique solution, we shall find this solution uf with Scilab, using $fsolve$

```
  -->tf=20;e=0.9;
  -->deff('v=kepler(u)','v=u-tf-e*sin(u)')

-->uf=fsolve(1,kepler)
 uf  =

    20.8267099361762185

--> solex=[=cos(uf)-e;...
-->-sin(uf)/(1-e*cos(uf));...
-->sqrt(1-e^2)*sin(uf);...
-->sqrt(1-e^2)*cos(uf)/(1-e*cos(uf))]
 solex =

! - 1.29526625098757586 !
! - 0.67753909247075539 !
!   0.40039389637923184 !
! - 0.12708381542786892 !
```

This result corresponds to the solution in Shampine. Here is some computations

```
   -->t0=0;x0=[0.1;0;0;sqrt(19)];tf=20;

   -->solstand=ode(x0,t0,t,D5);
lsoda--  at t (=r1), mxstep (=i1) steps
needed before reaching tout
      where i1 is :        500
      where r1 is :   0.1256768952473E+02
```

Scilab tells you that the number of steps is too small. We shall then modify this. To have access to % ODEOPTIONS, we type

```
  -->%ODEOPTIONS=[1,0,0,% inf,0,2,20000,12,5,0,-1,-1];

  -->solstand=ode(x0,t0,tf,D5)
 solstand  =

! - 1.29517035385191548 !
! - 0.67761734310378363 !
!   0.40040490187681727 !
! - 0.12706223352197193 !

-->rtol=1d-4;atol=1d-4;
--solad2=ode('adams',x0,t0,tf,rtol,atol,D5);

-->norm(solex1-solad2)/norm(solex1)
 ans  =

    0.12383663388400930

-->rtol=1d-12;atol=1d-14;

-->solad14=ode('adams',x0,t0,tf,rtol,atol,D5);

-->norm(solex1-solad14)/norm(solex1)
 ans  =

    0.00000000085528126

    // comparing the solutions

    -->solex
 solex  =

! - 1.29526625098757586 !
! - 0.67753909247075539 !
!   0.40039389637923184 !
! - 0.12708381542786892 !

-->solad2
```

```
 solad2  =

! - 1.420064880055532   !
! - 0.56322053593253107 !
!   0.33076856194984439 !
! - 0.17162136959584925 !

-->solad14
 solad14  =

! - 1.29526624993082473 !
! - 0.67753909321197181 !
!   0.40039389630222816 !
! - 0.12708381528611454 !
```

A tolerance of 10^{-4} gives a result with one correct digit ! the default tolerance 4 significant digits, a tolerance of 10^{-14} gives 7 correct digits. This problem illustrates the dangers of too crude tolerances.

5.4.2 Roberston Problem : stiffness

The following system is a very popular example in numerical studies of stiff problem. It describes the concentration of three products in a chemical reaction :

$$\begin{cases} \dot{x_1} = -k_1 x_1 + k_2\, x_2\, x_3 \\ \dot{x_2} = k_1 x_1 - k_2\, x_2\, x_3 - k_3 x_2^2 \\ \dot{x_3} = k_3 x_2^2 \end{cases} \tag{21}$$

The coefficients are

$$\begin{aligned} k_1 &= 0.04 \\ k_2 &= 10^4 \\ k_3 &= 3.10^7 \end{aligned}$$

With the initial value

$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

The Jacobian of the system is

$$J = \begin{bmatrix} -k_1 & k_2 x_3 & k_2 x_2 \\ k_1 & -k_2 x_3 - 2k_3 x_2 & -k_2 x_2 \\ 0 & 2k_3 x_2 & 0 \end{bmatrix}$$

It is straightforward to see that the eigenvalues of J are 0 and two others negatives values

$$\frac{-k_1 - 2\,k_2\,x_2 - k_2\,x_3 \pm \sqrt{-8\,k_2\,x_2\,\left(k_1 + k_3\,x_2\right) + \left(k_1 + k_2\,\left(2\,x_2 + x_3\right)\right)^2}}{2}$$

Check that for this system, with the corresponding initial value, $x_1+x_2+x_3 = 1$. Hint $= \dot{x}_1 + \dot{x}_2 + \dot{x}_3 = 0$. The equations are redondant.
We have one equilibrium $x_1 = x_2 = 0$ and $x_3 = 1$. At the I.V. the eigenvalues are

$$(0, 0, -k_1)$$

At the equilibrium the eigenvalues are

$$(0, 0, -(k_1 + k_2))$$

We remark that the system let invariant the positive orthant. This is appropriate, since concentration cannot be negative. To study the system it is sufficient to study the two first equation. On this orthant we have a Lyapunov function $V(x_1, x_2) = x_1 + x_2$. It is clear that

$$\dot{V} = \dot{x}_1 + \dot{x}_2 = -k_3 x_2^2 \leq 0$$

By LaSalle principle, the greatest invariant set contained in $\dot{V} = 0$, i.e. $x_2 = 0$ is reduced to $x_1 = 0$ (look at the second equation). Hence the equilibrium $x_1 = x_2 = 0$ is a globally asymptotically stable equilibrium (in the positive orthant). This means that all the trajectories goes toward $(0, 0, 1)$. Near this equilibrium the system is stiff, since $-(k_1 + k_3) = -10^4 - 0.04$
We can now look at the numerical solutions
Before we define a file Robertsoneqn.sci, and a file JacRober the Jacobian of the RHS.

```
  function xdot=JacRober(t,x)
xdot=[-.04 , 1d4*x(3), 1d4*x(2);..
.04,-1d4*x(3)-3.d7*x(2),-1.d4*x(2);..
0,6D7*x(2),0]
```

Then we compute, solve and plot, in a first time, naively :

```
 ;getf("/Users/sallet/Documents/Scilab/Robertsoneqn.sci");
-->;getf("/Users/sallet/Documents/Scilab/JacRober.sci");
 -->T=logspace(-6,11,10000);
 --> t0=0;x0=[1;0;0];
--> sol1=ode(x0,t0,T,Robertsoneqn);
-->xbasc()
-->plot2d(T',(sol1)')
```

We obtain

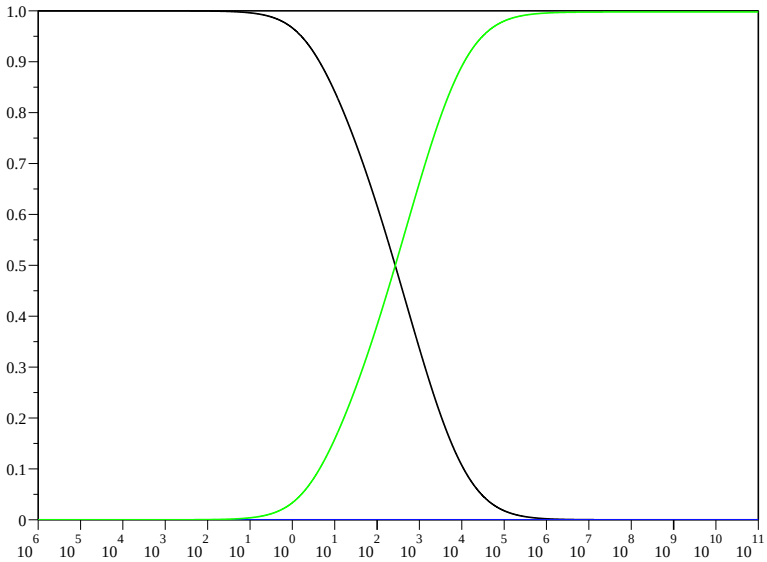

FIG. 11 – Robertson chemical reaction

It seems that a curve has disappeared. More attention shows that the second curve is on the x–axis. To see what happens we multiply $x_2(t)$ by 10^4

We have specified in the plot some options. $strf =' xyz'$means no captions (x=0), with a rectangle defined (y=1), axes are drawn, y axis on the left (z=1). The option *rect* precise the frame by a length 4 column vector

$$[xmin; ymin; xmax; ymax]$$

finally $logflag =$ 'ln' says that the x-axis is logarithmic and the y-axis is normal.

```
-->xbasc()
-->plot2d(T',(diag([1,1d4,1])*sol1)',strf='011',
rect=[1d-6;-.1;1d11;1.1],logflag='ln')
```

To obtain the classical figure.

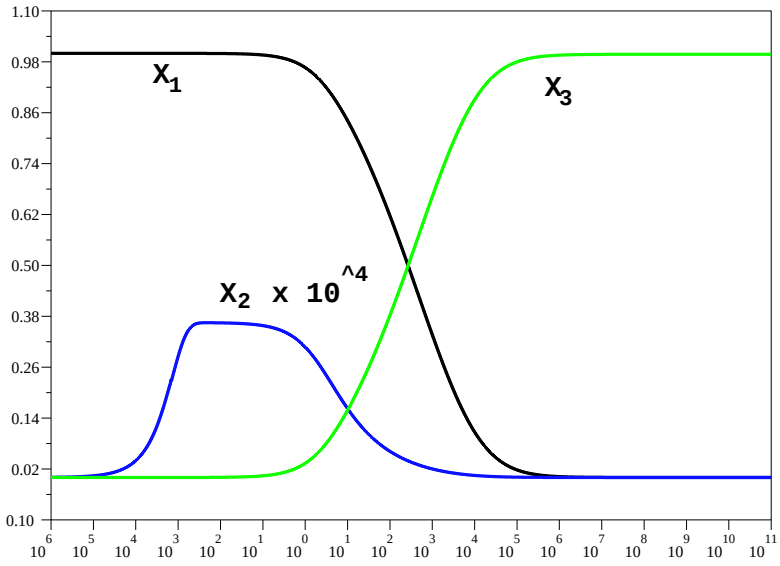

FIG. 12 – Robertson chemical reaction

Since the component x_2 becomes very small we can augment the requirement on atol for x_2 and also use the Jacobian.
If we have defined verb %ODEOPTIONS" we must only change jacstyle. See on the first example, two body, for `%ODEOPTIONS`. We have

```
 -->rtol=1d-4; atol=[1d-4;1d-6;1d-4];

-->%ODEOPTIONS(6)=1

-->solJac=ode(x0,t0,T,rtol,atol,Robertsoneqn,JacRober);

-->solJac(:,$)
 ans  =

!   0.00000001989864378 !
!   0.00000000000008137 !
!   0.97813884204576329 !

-->sol1(:,$)
 ans  =

!   0.00000002076404250 !
!   0.00000000000008323 !
!   0.99773865098969228 !
```

5.4.3 A problem in Ozone Kinetics model

We borrow this example from the book Kahaner-Moler-Nash. The system corresponds to the modeling of the amount of Ozone in the atmosphere. This example will illustrate the need of some mathematical analysis, with the necessity of some exploratory computations for adjusting the tolerances. One reaction mechanism is

$$O + O_2 \xrightarrow{k_1} O_3$$
$$O + O_3 \xrightarrow{k_2} 2\,O_2$$
$$O_2 \xrightarrow{k_3(t)} 2\,O$$
$$O_3 \xrightarrow{k_4(t)} O + O_2$$

The concentration of O_2, molecular oxygen, is many orders of magnitude larger than the other concentrations and therefore is assumed to be constant $O_2 = 3.5\,10^{16} cm^{-3}$
The equations are

$$\begin{cases} \dot{x_1} = -k_1 x_1\, O_2 - k_2\, x_1\, x_2 + 2k_3(t)\, O_2 + k_4(t)\, x_2 \\ \\ \dot{x_2} = k_1 x_1\, O_2 - k_2 x_1\, x_2 - k_4(t) x_2 \end{cases} \tag{22}$$

The different variables represent x_1 the concentration of free oxygen, x_2 the concentration of ozone. The rate of kinetics constants are known

$$k_1 = 1.63\ 10^{-16} \qquad k_2 = 4.66\ 10^{-16}$$

The two other reaction rates vary twice a day and are modeled by

$$k_i(t) = \begin{cases} e^{-c_i/\sin(\omega\, t)} & \text{if} \sin(\omega\, t) > 0 \\ \\ 0 & \text{if} \sin(\omega\, t) \leq 0 \end{cases}$$

With

$$\omega = \frac{\pi}{43200} s^{-1} \qquad c_3 = 22.62 \qquad c_4 = 7.601$$

The time is measured in seconds ($43200s = 12h = 1/2$ day)
The rates k_3 and k_4 govern the production of free singlet oxygen, they rise rapidly at dawn, reach a peak at noon, decrease to zero at sunset. These constants are zero at night. These functions are $\mathcal{C}^\infty$.
The I.V are

$$x_1(0) = 10^6\ cm^{-3} \qquad x_2 = 10^{12}\ cm^{-3}$$

A mathematical analysis shows that the system is "well-posed", i.e. the positive orthant is invariant by the system, namely any solution starting in the positive orthant cannot leaves this orthant. The system is difficult for the solvers. They are rapid change in the solutions at daylight and at night the ODE reduces to

$$\begin{cases} \dot{x_1} = -k_1 x_1\, O_2 - k_2\, x_1\, x_2 \\ \\ \dot{x_2} = k_1 x_1\, O_2 - k_2 x_1\, x_2 \end{cases}$$

Moreover the reaction rates are special. We shall plot the reaction rates k_3 and k_4. We have to code these functions and the ODE.

```
function xdot=ozone1(t,x)

k1=1.63d-16; k2=4.66d-16; O2=3.7d16;

xdot=[-k1*x(1)*O2-k2*x(1)*x(2)+2*k3(t)*O2+k4(t)*x(2);..
k1*x(1)*O2-k2*x(1)*x(2)-k4(t)*x(2)];

//////////////
function w=k3(t)
c3=22.62; omega=%pi/43200;

w= exp(-c3./(sin(omega*t)+...
((sin(omega*t)==0)))).*...(sin(omega*t)~=0)

//////
function J=jacozone1(t,x)
k1=1.63d-16; k2=4.66d-16; O2=3.7d16;

J=[-k1*O2-k2*x(2),-k2*x(1)+k4(t);k1*O2-k2*x(2),-k2*x(1)-k4(t)]

////////////
function v=k4(t)
c4=7.601; omega=%pi/43200;

v= exp(-c4./(sin(omega*t)+((sin(omega*t)==0)))).*...
(sin(omega*t)~=0)
```

Some remarks, here, are necessary. First we have coded in ozone1.sci some sub-functions of the main program. This is important when several functions must be supplied. We have as sub-function, the Jacobian, jacozone1 and the depending of time functions $k_3(t)$ and $k_4(t)$. These function are used by the main program "ozone".
We draw the reader's attention to the code of theses two functions. A natural, but clumsy , way to code "case functions" as k_3 is to use a **if** condition. There is a principle in vectorial language as Scilab to vectorize the code. Moreover using loops is time consuming. A motto is

Time is too short to short to spend writing loop

> A rule- of-thumb is that the execution time of a MATLAB (SCILAB) function is proportional to the number of statements executed. No matter what those statements actually do (C. Moler)

We use test functions.

```
sin(omega*t)==0
```

The value is $\%T$ if the test succeed , $\%F$ otherwise. These are boolean variables True or False. These variables can be converted in numerics where %T is replaced by 1 and % F replaced by 0. But if boolean variables are used in some arithmetic operation the conversion is automatic. Example :

```
-->(1-1==0)*2
 ans  =

    2.

-->(1-1==0)+2
 ans  =

    3.

-->(1-1==0)/2
           !--error     4
undefined variable : %b_r_s


-->2/(1-1==0)
           !--error     4
undefined variable : %s_r_b

 -->2/bool2s(1-1==0)
 ans  =

    2.
```

The operations $*$ and $+$ convert automatically, / need to use "bool2s". If you are not being sure, use bool2s.
We have in the code, the value of the function which is multiplied by

```
.*(sin(omega*t)~=0)
```

Note the "dot" operation to vectorize. This imply that our function is 0 when $\sin(\omega\, t) \leq 0$.
We add a term, in the denominator of the exponential :

```
+(   (sin(omega*t)==0)   )
```

To prevent to divide by zero, which shall give a error message an interrupt the program. In this manner we obtain the result. Moreover the function is vectorized, as we shall see :

```
-->;getf("/Users/sallet/Documents/Scilab/ozone1.sci");
-->T=linspace(2,43200,1000);

-->plot2d(T',[1d5*k3(T)',k4(T)'])
```

We have multiplied k_3 by 10^5 since k_3 varies from 0 to 15. 10^{-11} and k_4 varies from 0 to 5. 10^{-4}. Otherwise the graph of k_3 would have been squashed on the x-axis.

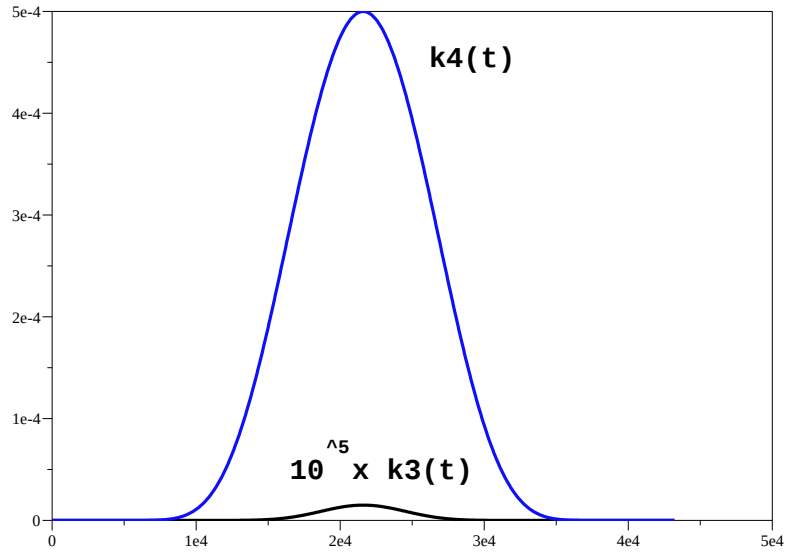

FIG. 13 – reaction rates

It seems that these functions are zero for some values of t in the interval. We check it , with the help of the function **find**

```
-->I2=find(k4(T)==0)
 I2  =

!   1.    2.    3.    4.    997.    998.    999.    1000. !

-->I1=find(k3(T)==0)
 I1  =


         column  1 to 11

!   1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  991. !

         column 12 to 19

! 992.  993.  994.  995.  996.  997.  998.  999. !

         column 20

!   1000. !
```

We have obtain the indexes for which the function are zero, we deduce the time

```
-->[T(4),T(5)]
 ans  =

!   131.72372    174.96496 !

-->k4([T(4),T(5)])
 ans  =

!   0.    3.568-260 !
```

```
-->k3(T(11))
 ans  =

    9.669-312
```

To understand what happens here, we must go back to the computer arithmetic. Scilab is conforming to the IEEE standard. To be under the complete IEEE arithmetic you must type

```
ieee(2)
```

This means that invalid operations give NaN (Not a Number). An Overflow give %inf. The range is $10^{\pm 308}$. Or in other words the overflow threshold is $1.797e308$ and the underflow around $4.941d-324$. There is no symmetry. The reader is advised to cautious, when working around this boundaries. The unit roundoff is %eps :

```
-->ieee(2)

-->log(0)
 ans  =

  -Inf

-->1/0
 ans  =

   Inf

-->0/0
 ans  =

   Nan

-->1.7977d308
 ans  =
```

```
   Inf

-->1.79765d308
 ans  =

    1.798+308

-->5d-324
 ans  =

    4.941-324

-->5d-325
 ans  =

    0.


-->4.8d-324
 ans  =

    4.941-324

-->4d-324
 ans  =

    4.941-324
```

In other words under 10^{-324} , for the computer everything is zero. With exponential function some care is necessary , as we have seen with the function k_i.
We make the following experience

```
-->T=linspace(1,500,5000);

-->Ind=find(k3(T));

-->min(Ind);
```

```
-->T(min(Ind))
 ans  =

    417.54871

-->k3(T(min(Ind)))
 ans  =

    4.941-324

  -->T=418.5:.01:420;

-->plot2d('nl',T',k3(T)')

T=linspace(42781,42789,100);

-Ind=max(find(k3(T)));          ;

-->T(Ind)
 ans  =

    42782.455
```

We are playing with the limits of underflow machine! Notice the shattering effects in the next figure.

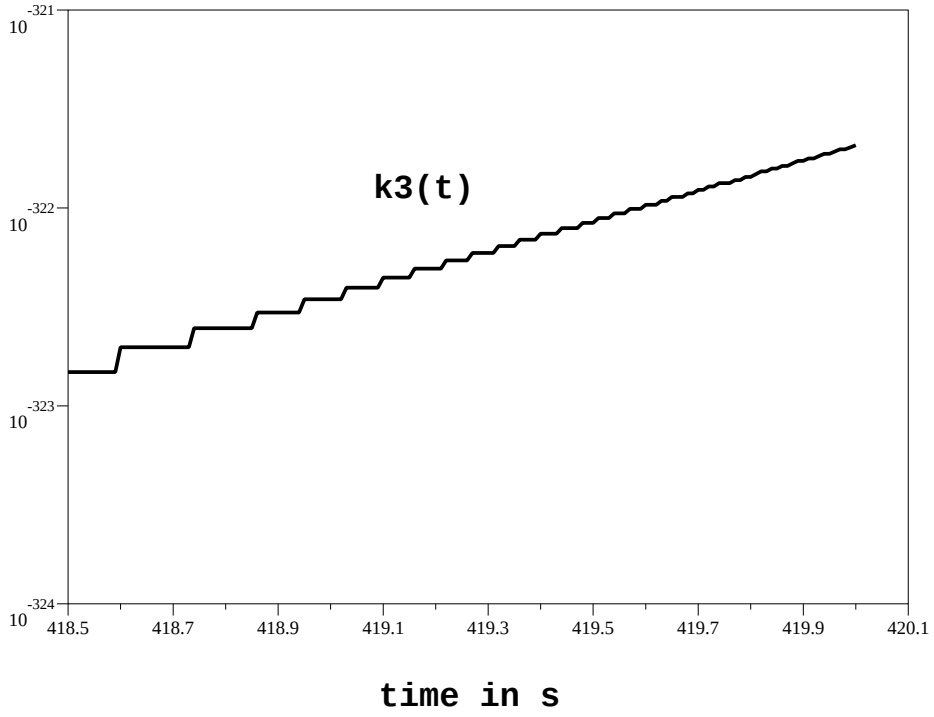

FIG. 14 – Effects of IEEE arithmetic

We know, that for the computer , the functions k_3 is zero on the intervals $[0, 418]$ and $[42782, 43200]$ and this periodically on period 43200. In the same manner we find out that k_4 is zero on the intervals $[0, 141]$ and $[43059, 43200]$ and this periodically on period 43200.
We shall first integrate in $[0, 141]$ with a simplified function ozone2.sci . Simply we have erased k_1, k_2 and modified the Jacobian :

```
function xdot=ozone2(t,x)

k1=1.63d-16; k2=4.66d-16; O2=3.7d16;

xdot=[-k1*x(1)*O2-k2*x(1)*x(2);k1*x(1)*O2-k2*x(1)*x(2)];

//////////////

function J=jacozone2(t,x)
```

```
k1=1.63d-16; k2=4.66d-16; O2=3.7d16;

J=[-k1*O2-k2*x(2),-k2*x(1);k1*O2-k2*x(2),-k2*x(1)]
```

We can now integrate. We choose to see the evolution, second by second, (look at the choice of T)

```
-->%ODEOPTIONS=[1,0,0,%inf,0,2,20000,12,5,0,-1,-1];

--> ;getf("/Users/sallet/Documents/Scilab/ozone2.sci");

-->t0=0; x0=[1d6;1d12];

-->T1=linspace(1,141,141);

-->sol1=ode(x0,t0,T1,ozone2);

-->plot2d(T1',sol1(1,:)')
```

Since the two components are on different scale we plot two separate figures. We obtain the two plots

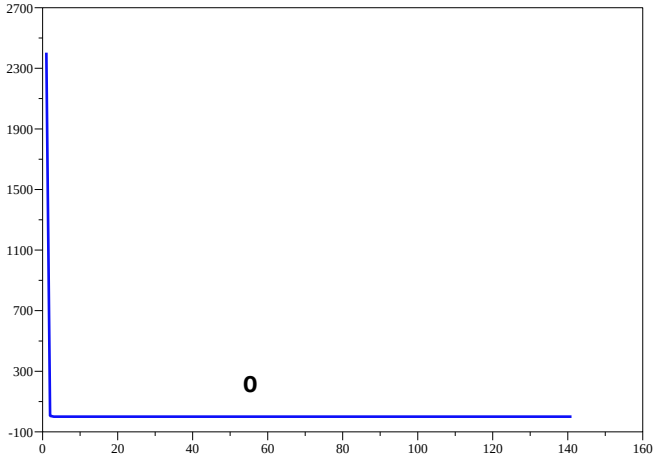

FIG. 15 – concentration of free oxygen

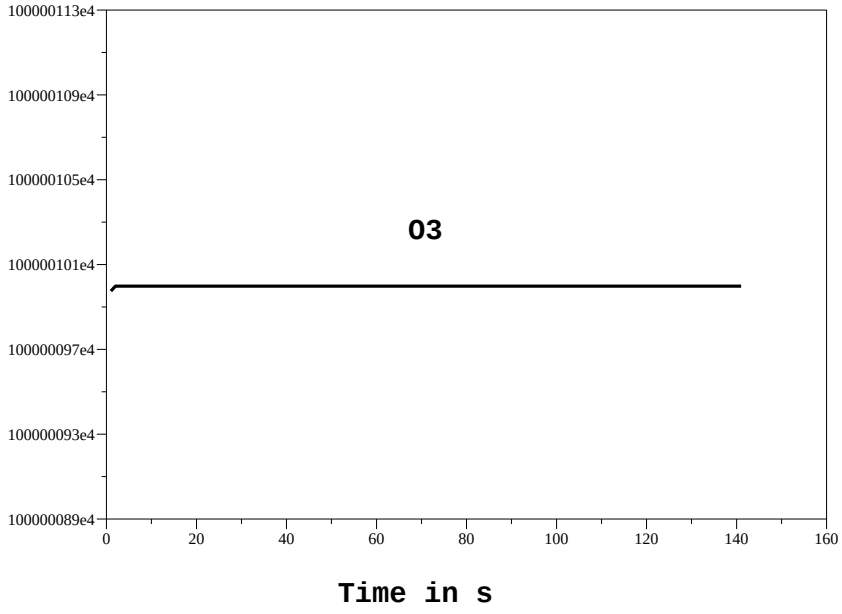

FIG. 16 – concentration of Ozone

We see on the figure that the concentration is negative! We check this

```
-->ind=find(sol1(1,:)<0);
 -->min(ind)
 ans  =

    7.

-->T(7)
 ans  =

    7.

-->sol1(1,6)
 ans  =

    0.0000000001981

-->sol1(1,7)
```

```
 ans  =

  - 0.0000000000283
```

We also see that the ozone rapidly reach a constant value, 14 significant digits remain constant :

```
-->format('v',16)

-->sol1(2,10)
 ans  =

    1000000999845.5

-->sol1(2,$)
 ans  =

    1000000999845.5
```

After some experimentation, we choose *atol* and *rtol*. For the first component x_1, what is important is the absolute error, since x_1 is rapidly very small. See the discussion on error and tolerance absolute and relative in paragraph (5.2.1). The components are scaled very differently as we can see. There exists a rule-of-thumb, to begin :

> Let m the number of significant digits required for solution component x_i, set $rtol(i) = 10^{m+1}$ and $atol(i)$ at the value at which $|x_i|$ is essentially significant.

Beginning with this and after some trial and error, we come with

```
-->rtol=[1d-2;1d-13];

-->atol=[1d-150,1d-2];

-->sol11=ode(x0,t0,T1,rtol,atol,ozone2);
```

```
-->ind=find(sol11(1,:)<0);

-->min(ind)
 ans  =

    60.

-->max(ind)
 ans  =

    77.
```

It means that we have only 17 values which are negative.
We can try with a stiff only method.

```
-->%ODEOPTIONS(6)=1;

-->sol11=ode(x0,t0,T1,rtol,atol,ozone2,jacozone2);

--ind=find(sol11(1,:)<0);

--min(ind)
 ans  =

    60.

--max(ind)
 ans  =

    77.
```

The result is equivalent. A try with a pure adams method gives poorer results. We shall now take the last results of the integration as new start, and use on the interval $[141, 418]$ a function ozone3.sci which use only the function $k_4(t)$. This is immediate and we omit it.

```
-->x0=sol11(:,$)
 x0  =
```

```
!   1.914250067-152 !
!   1000000999845.5 !

-->to=T1($)
 to  =

    141.

-->;getf("/Users/sallet/Documents/Scilab/ozone3.sci");

-->T2=linspace(141,418,277);

-->sol2=ode(x0,t0,T2,rtol,atol,ozone3,jacozone3);

-->xbasc()

-->plot2d(T2',sol2(1,:)')

-->xset('window',1)

-->xbasc()

-->plot2d(T2',sol2(2,:)')
-->min(sol2(1,:))
 ans  =

  - 3.483813747-152
```

The first component has still negative values, but they are in the absolute tolerance! We obtain the figure for the concentration of free oxygen. The figure for the ozone is the same and the concentration stays constant.

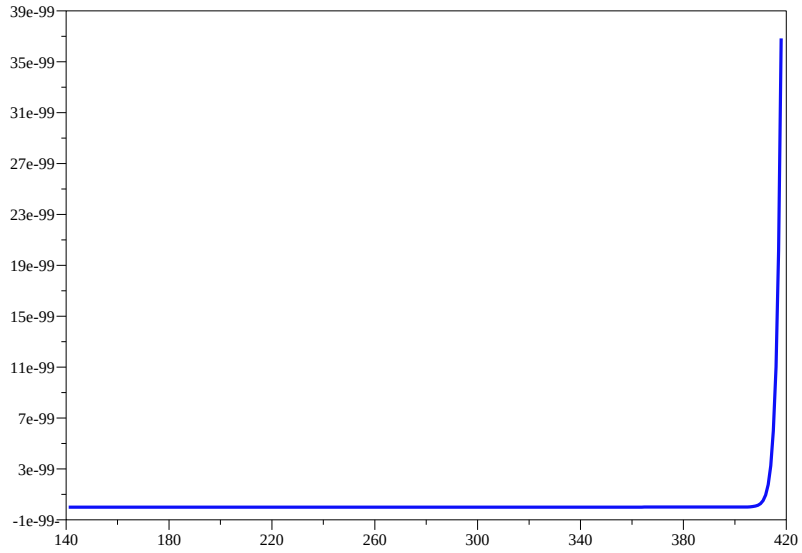

FIG. 17 – concentration of free oxygen on $[141, 418]$

We can now integrate on the interval $[418, 43059]$, with the two functions k_i, using ozone1.sci the complete program. We make an exploratory computation with default tolerance $atol = 10^{-7}$ and $rtol = 10^{-5}$.

```
t0=T2($);x0=sol2(:,$);
-->;getf("/Users/sallet/Documents/Scilab/ozone1.sci");
-->rtol=1d-7;atol=1d-9;
-->T3=linspace(418,42782,42364);

-->sol3=ode(x0,t0,T3,rtol,atol,ozone1);

-->xset('window',2)

-->plot2d(T3',sol3(1,:)')

-->xset('window',3)

-->plot2d(T3',sol3(2,:)')

-->min(sol3(1,:))
```

```
 ans  =

  - 1.814E-11

-->sol3(1,$)
 ans  =

    1.866E-13
```

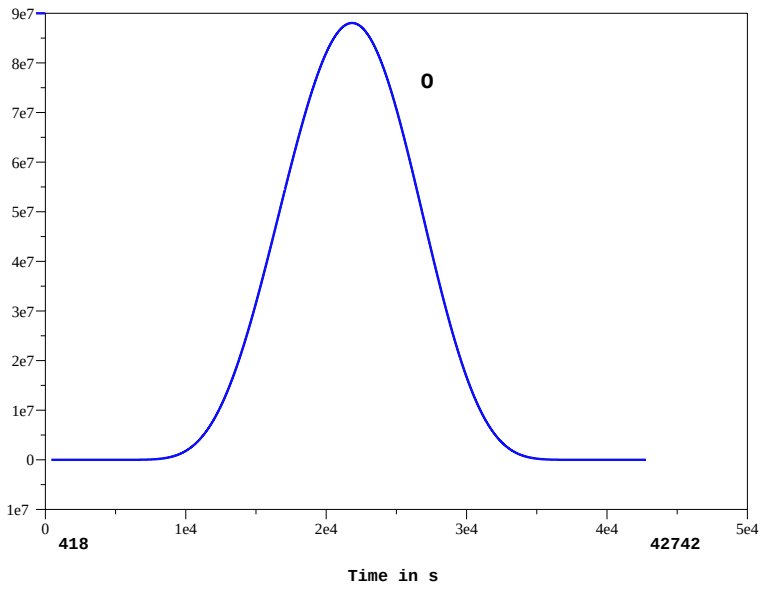

FIG. 18 – Free oxygen on day light

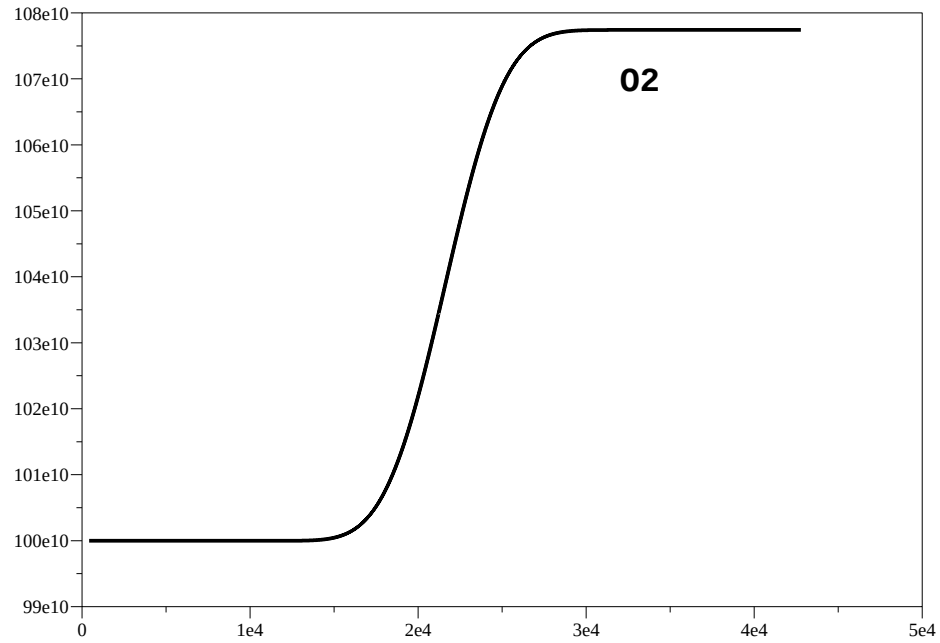

FIG. 19 – Molecular oxygen, day light

On day light the free oxygen, starting at very low concentration ($3.6\ 10^{-98}$) goes to a peak $8.8\ 10^{7}$ at noon ($21429s$), and then decreases to $1.86\ 10^{-13}$. We check that the concentration of O has still negative values, but under the value of $atol$. We observe that the concentration of molecular oxygen change from a plateau $1.000\ 10^{12}$ to another plateau $1.077\ 10^{12}$. Our tolerance are satisfactory.

We complete now the simulation, with two other runs, on the third interval, corresponding to $[42782, 43059]$ with the function ozone3 , and on the interval $[43059, 43341]$, the remaining of the day and the beginning of the next one, with ozone2, since the k_i are zero. The remaining of the simulation is obtained in a periodic way. You should write a script to obtain for example a simulation on 10 days.

For reference for the reader, we give the figure corresponding for the two intervals for the concentration of free oxygen at the beginning of the day, and the two intervals at the end and at the beginning of the next day. If we plot, and pasted all the results obtained, together, these plots will disappear, in reason of their scale. We shall get something like the figure (18).

We use the following script :

```
-->xbasc()

-->[([0,0.0,0.5,0.5])

-->plot2d(T1',sol11(1,:)')

-->xsetech([0.5,0.0,0.5,0.5])

-->plot2d(T2',sol2(1,:)')

-->xsetech([0,0.5,0.5,0.5])

-->plot2d(T4',sol4(1,:)')

-->xsetech([0.5,0.5,0.5,0.5])

-->plot2d(T5',sol5(1,:)')
```

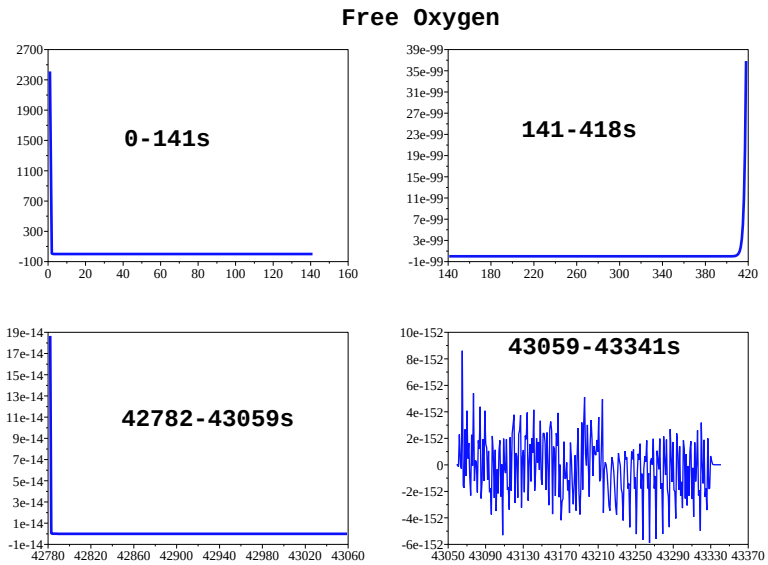

FIG. 20 – Molecular oxygen at low levels

The chattering effect in the graph of interval $[43059, 43341]$ is simply a scale zoom effect. The same thing arrive in the interval $[0, 141]$ but the I.V. for this

interval is 10^6, and we begin the plot at time 1 giving a concentration of 2410, whereas the I.V. at the beginning of the interval for the concentration of free oxygen is 3.8 10^{-156}. The chattering effect in interval $[0, 141]$ is squashed by the scale. You can check by yourself, that this happen also in the first interval, with the command

```
-->xbasc()

-->plot2d(T1(70:141)',sol11(1,70:141)')
```

The remaining of the simulation should give something approximatively periodic, since the I.V. are almost the same, comparing the starting for the second intervals ($k_4 \neq 0$ and $k_3 = 0$) :

```
-->sol5(:,$)
 ans  =

!   2.982-155 !
!   1.077E+12 !

-->sol11(:,$)
 ans  =

!   1.924-152 !
!   1.000E+12 !
```

Do it ! Obtain a simulation for 10 days.

5.4.4 Large systems and Method of lines

Till now we have just solve ODE with small dimension. But the solver of Scilab are able to handle large systems. We shall here consider an example in dimensions 20 and 40. But depending on your computer, problems of larger size can be considered.
A source of large systems of ODE is the discretization of PDE. We shall consider some examples and particularly the example of the Brusselator with diffusion (HW).

The brusselator with diffusion is the system of PDE :

$$\begin{cases} \frac{\partial u}{\partial t} = A + u^2 v - (B+1)u + \alpha \frac{\partial^2 u}{\partial x^2} \\ \frac{\partial v}{\partial t} = -Bu - u^2 v + \alpha \frac{\partial^2 v}{\partial x^2} \end{cases} \tag{23}$$

With $0 \leq x \leq 1$
With the boundaries conditions

$$u(0,t) = u(1,t) = 1 \qquad v(0,t) = v(1,t) = 3$$

$$u(x,0) = 1 + \sin(2\pi x) \qquad v(x,0) = 3$$

When a PDE is discretized in space (semi-discretization) the result is an ODE.
For example, let consider the variable coefficient wave equation

$$u_t + c(x)u_x = 0$$

For $x \in [0, 2\pi]$. We consider the grid $0 = x_0 < x_1 < x_2 < \cdots < x_N < x_{N+1} = 2\pi$

Some functions $v_i(t)$ are used to approximate the function $u(x_i, t)$. Another approximation is given of the spatial derivatives of $u(t, x)$.
For example we can use a first order difference approximation on a grid of spacing Δx

$$\frac{\partial u(t, x_i)}{\partial x} \approx \frac{v_i(t) - v_{i-1}(t)}{\Delta x}$$

A more accurate approximation can be a central difference

$$\frac{\partial u(t, x_i)}{\partial x} \approx \frac{v_{i+1}(t) - v_{i-1}(t)}{2\Delta x}$$

Of course these formulas must be adapted at the boundary of the grid.
If one use the spectral methods,

$$\frac{\partial u(t, x_i)}{\partial x} \approx (Dv)_i$$

Where D is a differentiation matrices applied to the function v, and $(Dv)_i$ is the ith component.
Consider the Brusselator with diffusion (23). We use a grid of N points

$$x_i = \frac{i}{N+1} \qquad i = 1 : N \qquad \Delta x = \frac{1}{N+1}$$

We approximate the second spatial derivatives by finite difference to obtain a system of ODE of dimension N.

$$\begin{cases} \dot{u}_i = A + u_i^2 v_i - (B+1)u_i + \frac{\alpha}{(\Delta x)^2}\,(u_{i-1} - 2u_i + u_{i+1}) \\ \dot{v}_i = Bu_i - u_i^2 v_i + \frac{\alpha}{(\Delta x)^2}\,(v_{i-1} - 2v_i + v_{i+1}) \\ u_0(t) = u_{N+1}(t) = 1; v_0(t) = v_{N+1}(t) = 3 \\ u_i(0) = 1 + 2\sin(2\pi x_i) \\ v_i(0) = 3 \end{cases} \tag{24}$$

We shall code this ODE. We ordered the components and the equations in the following order :

$$(u_1, v_1, u_2, v_2, \cdots, u_N, v_N)$$

That is we set $x_{2i-1} = u_i$ and $x_{2i} = v_i$ for $i = 1 : N$. If we adopt the convention $x_{-1} = u_0(t) = x_{2N+1} = u_N(t) = 1$ and $x_0(t) = v_0(t) = x_{2N+2}(t) = v_N(t) = 3$ to incorporate the boundary conditions, the equation (24) is equivalent to

$$\begin{cases} \dot{x}_{2i-1} = A + x_{2i-1}^2\, x_{2i} - (B+1)x_{2i} + \frac{\alpha}{(\Delta x)^2}\,(x_{2i-3} - 2x_{2i-1} + x_{2i+1}) \\ \dot{x}_{2i} = Bx_{2i-1} - x_{2i-1}^2\, x_{2i} + \frac{\alpha}{(\Delta x)^2}\,(x_{2i-2} - 2x_{2i} + x_{2i+2}) \\ x_{2i-1}(0) = 1 + 2\sin(2\pi \frac{i}{N+1}) \\ x_{2i}(0) = 3 \end{cases} \tag{25}$$

It is straightforward, from the original equation to check that each RHS of the system depends at most only of the 2 preceding and the 2 following coordinates. This proves that the Jacobian is banded with mu=2 and ml=2. If we follow the syntax of the section (5.3.11) the matrix J is, setting, for shortening the formulas , $D = (B+1) + 2c$
For the 4 first columns

$$\begin{bmatrix} 0 & 0 & c & c & \cdots \\ 0 & x_1^2 & 0 & x_3^2 & \cdots \\ 2x_1x_2 - D & -x_1^2 - 2c & 2x_3x_4 - D & -x_3^2 - 2c & \cdots \\ B - 2x_1x_2 & 0 & B - 2x_1x_4 & 0 & \cdots \\ c & c & c & c & \cdots \end{bmatrix}$$

And for the remaining columns till the two last columns.

$$\begin{bmatrix} \cdots & c & c & \cdots & c & c \\ \cdots & x_{2i-1}^2 & 0 & \cdots & 0 & x_{2N-1}^2 \\ \cdots & 2x_{2i-1}x_{2i} - D & -x_{2i-1}^2 - 2c & \cdots & 2x_{2N-1}x_{2N} - D & -x_{2N-1}^2 - 2c \\ \cdots & B - 2x_{2i+1}x_{2i} & 0 & \cdots & B - 2x_{2N+1}x_{2N} & 0 \\ \cdots & c & c & \cdots & 0 & 0 \end{bmatrix}$$

The code, using the vectorization properties of Scilab, results from these formulas. We introduce a temporary vector to take care of the boundaries conditions. The Jacobian is coded as a sub-function of the main program.

```
function xdot=brusseldiff(t,x)
//constants
A=1;B=3;alpha=1/50;
//
// dimension of the problem
N=20;
c=alpha*(N+1)^2;

// A temporary vector for taking in account
// boundary conditions
x=x(:);
y=[1;3;x;1;3];
xdot=zeros(2*N,1)

i=1:2:2*N-1

xdot(i)=A+y(i+3).*y(i+2).^2-(B+1)*y(i+2)+...
c*(y(i)-2*y(i+2)+y(i+4))

xdot(i+1)=B*y(i+2)-y(i+3).*y(i+2).^2+...
c*(y(i+1)-2*y(i+3)+y(i+5))

/////////////////
function J=Jacbruss(t,x)

//banded Jacobian ml=2  mu=2
```

```
//see the notes for the explaination
x=x(:); N=20;
J=zeros(5,2*N);
A=1;B=3;alpha=1/50;
c=alpha*(N+1)^2;
d=B+1+2*c;

// five lines to be defined

J(1,3:2*N)=c;
J(5,1:2*N-3)=c;
J(2,2:2:2*N)=(x(1:2:2*N).^2)';
J(3,1:2:2*N)=(2*x(1:2:2*N).*x(2:2:2*N))'-d;
J(3,2:2:2*N)=J(2,2:2:2*N)-2*c;
J(4,1:2:2*N)=-J(3,1:2:2*N)-2*c-1;
```

Now we can integrate the ODE on the interval $[0, 10]$.
We have to call the function brussdiff.sci, set the $\%ODEOPTIONS$ with Jactype=4,mu=2,ml=2, maxstep=20000, and set the I.V. :

```
-->;getf("/Users/sallet/Documents/Scilab/brusseldiff.sci");
-->%ODEOPTIONS=[1,0,0,%inf,0,4,20000,12,5,0,2,2];
-->N=20;
-->X=(1:N)/(N+1); t0=0; T=linspace(0,10,100);
-->x0=zeros(2*N,1);
-->x0(1:2:2*N)=1+sin(2*\%pi*X)';
-->x0(2:2:2*N)=3;
-->solbru=ode(x0,t0,T,brusseldiff,Jacbruss);
```

The solution is obtained in a snap. We can do the same for $N = 40$. You just have to change in the function brussdiff.sci the value of N , in only two places at the beginning of the function brussdiff and Jacbruss. Don't forget to save the file and make a "getf", then you retype in the window command the same 6 lines of instructions and you get the solution for N=40 !
Now we can plot the solution of the PDE. u_i is the discretization of the solution u. Using the properties of plot3d , thes lines of commands

```
--plot3d1(T,X,solbru(1:2:2*N,:)')
-->xset('window',1)
--plot3d1(T,X,solbru(2:2:2*N,:)')
```

Give the figures

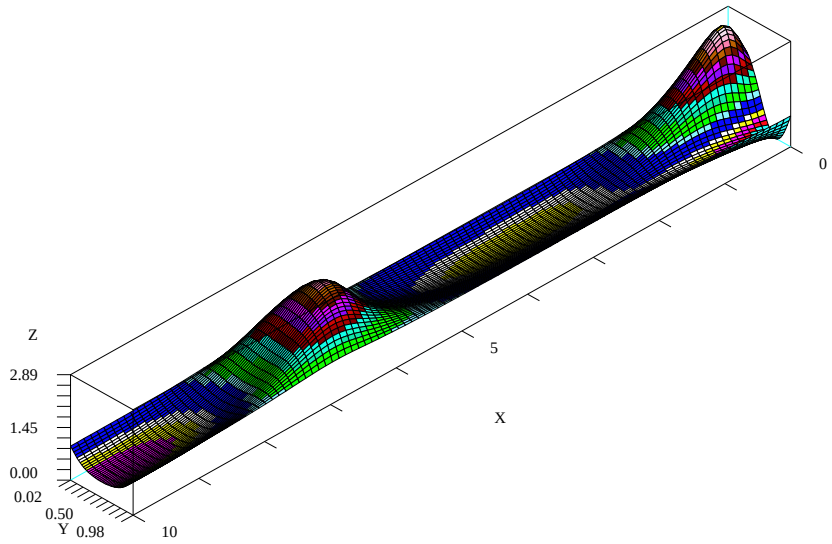

FIG. 21 – solution of $u(x,t)$ of (23)

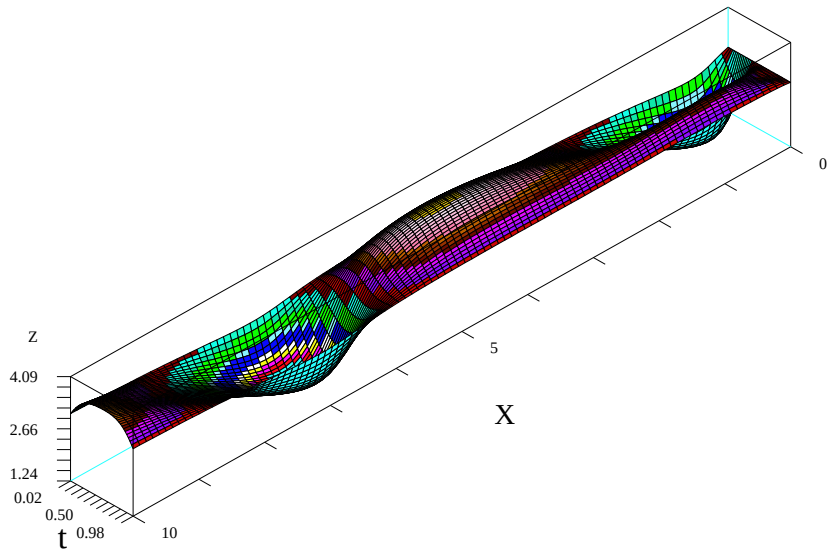

FIG. 22 – Solution of $v(x,t)$ of (23)

To obtain a comparable figure as in HW pp7, we change the point of view, and change the orientation of the x-axis.

```
-->plot3d1(T,X(N:-1:1),solbru(1:2:2*N,:)',170,45)
```

And obtain

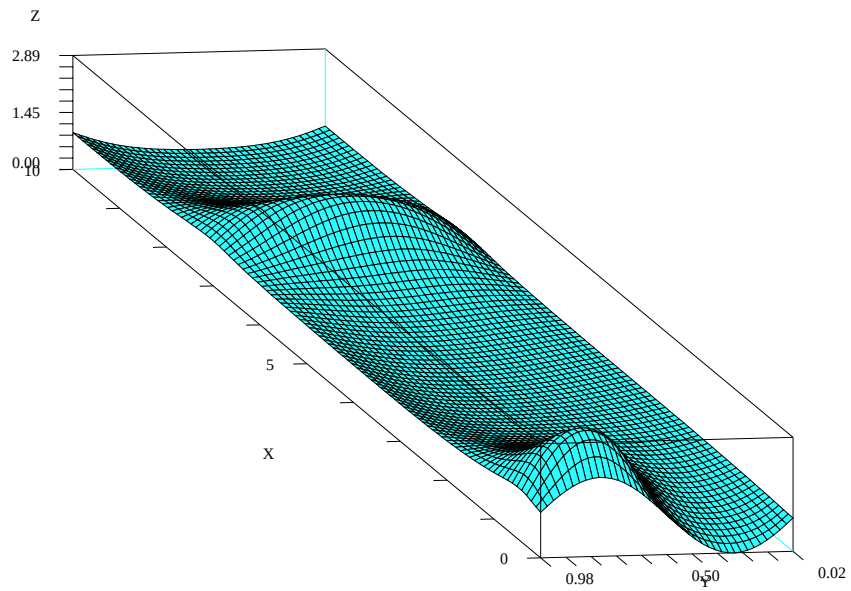

FIG. 23 – Solution $u(x,t)$ other point of view

You are now able to test the ODE solver of MATLAB on classic test problems. See the section of examples. Do it ! Experiment, make your own errors, it is the only way to learn how to use Scilab.

5.5 Passing parameters to functions

It is frequent to study ODE with parameters and to want to see what are the effects of parameters on the solutions. The pedestrian way is to change the parameters in the file of the ODE, save the file and in the window command make a "getf" for this file to redefine the function. In the preceding section we have use this way to change the number of discretization N from 20 to 40. If you want to test, for example for 100 parameters this is not very convenient. In this file N has an affected value in the file.

If a variable in a function is not defined (and is not among the input parameters) then it takes the value of a variable having the same name in the calling environment. This variable however remains local in the sense that modifying it within the function does not alter the variable in the calling environment. A function has then access to all the variables in the superior levels, but cannot modified this variables. Functions can be invoked with less input or output parameters.
It is not possible to call a function if one of the parameter of the calling sequence is not defined
In this section we shall give a way to do affect parameters easily and to automatize the process using a script.
In the syntax of ODE of Scilab you replace the argument f by a list :

```
ode ( x0,t0, t ,lst)
```

Where lst is a list with the following structure

```
lst=list (f, param1,param2, ...,paramk);
```

The argument $param1$, $param2$,..., $paramk$ are the parameters you want.
The function f must be a function with the syntax

```
function  xdot =f(t,x,param1,param2,...,paramk)
```

We shall give an example. If a Jacobian is coded the same rules applies.

5.5.1 Lotka-Volterra equations, passing parameters

The ODE Lotka-Volterra are contained in any book on ODE.
The predator-prey Lotka-Volterra equations are defined by

$$\begin{cases} \dot{x_1} &= (a - b\,x_2)\,x_1 \\ \dot{x_2} &= (-c + dx_1)\,x_2 \end{cases} \tag{26}$$

We have 4 positive parameters : (a, b, c, d)
The we code the function

```
 function xdot=lotka(t,x,a,b,c,d)

xdot=[(a-b*x(2))*x(1);(-c+d*x(1))*x(2)]
```

Now at the command line, we can give values to the parameters and call the solver ODE

```
a= 1;b=1;c=1=d=2; //example
lst=list(lotka,a,b,c,d)
sol=ode(x0,t0,t,lst);
```

To change the parameters it suffice to change the values of (a, b, c, d).
For example we fix $a = 1$, $b = 0.1$,$c = 1$ and we want to see what happens when d varies from 0.1 to 0.9. We want to have the solution $(x_1(10), x_2(10))$ for each value of the parameter $d = 0.1 : 0.1 : 0.9$ with initial value $[1; 1]$. Here is a script lotkascript.sci for getting an array answering the question

```
// Lotka script for the tableau
//
;getf("/Users/sallet/Documents/Scilab/lotka.sci");
a=1;b=0.1;c=1:
d=.1:.1:.9;
t0=0;
tf=10;
x0=[1;1];
tableau=zeros(2,9);
tableau=[alpha;tableau];
n=length(alpha);
for i=1:n
X=ode(x0,t0,tf,list(lotka,a,b,c,d(i)));
tableau(2:3,i)=X;
end
```

In the window command you execute the script

```
-->;exec("/Users/sallet/Documents/Scilab/lotkascript.sci");

-->tableau
tableau=

!   0.1        0.2        0.3        0.4        0.5       !
!   2.7917042  0.7036127  0.5474123  0.5669242  0.6287856 !
!   25.889492  13.875439  6.9745155  4.3248549  3.0348836 !
```

```
         column 6 to 9

!   0.6          0.7          0.8          0.9        !
!   0.7055805    0.7867920    0.8661472    0.9386788  !
!   2.2915144    1.8082973    1.4650893    1.2052732  !
```

5.5.2 variables in Scilab and passing parameters

To sum up the question of passing parameters. If you change often the parameters, Use parameters as input in the function and use list when you invoke solvers with ODE.
You can use in a function some parameters, inside the code, but these parameters must exists somewhere. Created in the file, or existing in the current local memory of the window command.
It is not necessary to create global variables. A warning : a sub-function has not access to the parameters of the other function of the program.
For example, consider the following function foo.sci

```
function w=foo(t)
c3=22.62; omega=%pi/43200;
w=1+subfoo(t)+N
//
function z=subfoo(t)
z=c3*t
```

If you type in Scilab

```
-->N=40;
-->;getf("/Users/sallet/Documents/Scilab/foo.sci");
-->foo(1)
 ans  =

    63.62

-->subfoo(1)
 !--error     4
undefined variable : c3
at line       2 of function subfoo       called by :
subfoo(1)
```

The function subfoo has not access to c3, only foo. To remedy you have 3 solutions

- create c3 in subfoo, giving it value. This is the manner used in subfunction Jacozone of function ozone.sci in section (5.4.3). In this case c3 is a variable of the program foo.sci
- create c3 in the base workspace, by typing in the window command.
- declaring c3, global in foo and in subfoo. Affecting a value to c3 in foo. This equivalent to the first method, with a little subtlety

Ordinarily, each Scilab function, has its own local variables and can "read" all variables created in the base workspace or by the calling functions. The global allow to make variables read/write across functions. Any assignment to that variable, in any function, is available to all the other functions declaring it global. Then if for example you create the two functions

foo1.sci (see the difference with foo.sci)

```
function w=foo1(t)
global c3
c3=22.62;
w=1+subfoo(t)/c3
//
function z=subfoo(t)
global c3
z=c3*t
```

And the function foo.2.sci

```
function w=foo2(t)
global c3
w=1+subfoo2(t)/c3
//
function z=subfoo2(t)
 global c3
z=c3*t
```

If you load by getf these two functions, a simple trial show that foo.1 and foo.2 share the variable c3, even with the variable c3 having no given value in foo2, it is just declared global. but however c3 is not a variable in the base workspace.

```
-->foo2(1)
 ans  =
    2.
-->subfoo1(1)
 ans  =
    22.62
-->c3
  !--error     4
undefined variable : c3
```

5.6 Discontinuities

As we have seen in the theoretical results for existence and uniqueness for ODE, a current hypothesis is a certain regularity of the function defined by the RHS of the ODE. Lipschitzian is the minimum, and very often some smoothness is supposed. Moreover the theoretical results for the convergence and the order of the methods suppose that the function f is sufficiently derivable (at least p). Then when the function in the RHS has some discontinuity we have a problem. Unfortunately it is often the case in applications.

All the algorithms are intended for smooth problems. Even if the quality codes can cope with singularities, this is a great burden on the code, and unexpected results can come out. We shall illustrate this by an example from the book of Shampine.

5.6.1 Pharmacokinetics

This example is a simple compartmental model of Lithium absorption. Mania is a severe form of emotional disturbance in which the patient is progressively and inappropriately euphoric and simultaneously hyperactive in speech and locomotor behaviour. In some patients, periods of depression and mania alternate, giving rise to the form of affective psychosis known as bipolar depression, or manic-depressive disorder. The most effective medications for this form of emotional disorder are the simple salts lithium chloride or lithium carbonate. Although some serious side effects can occur with large doses of lithium, the ability to monitor blood levels and keep the doses within modest ranges (approximately one milliequivalent [mEq] per litre) makes it an effective remedy for manic episodes and it can also stabilize the mood swings of the manic-depressive patient.

The carbonate lithium has a half-life of 24 hours in the blood. The first question is with a dosage of one tablet every 12 hours, how long does it take for the medication to reach a steady-state in the blood. Remember that the safe range is narrow.
This is modeled by a two compartment model. The first is the digestive tract and the second is the blood. The dynamic can be summarized by

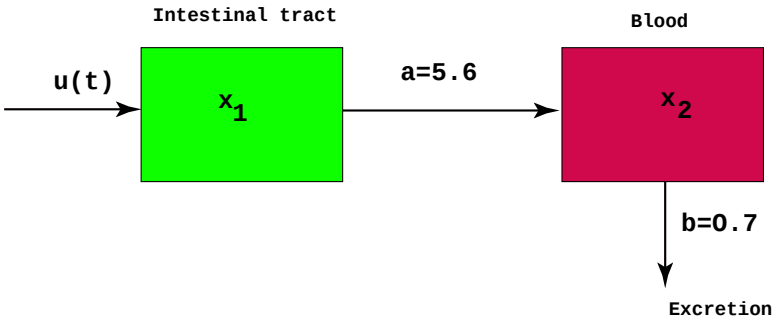

FIG. 24 – Compartment model of Lithium

A mass balance analysis give the equation

$$\begin{cases} \dot{x_1} &= -a\,x_1 + u(t) \\ \dot{x_2} &= a\,x_1 - b\,x_2 \end{cases} \tag{27}$$

Since the half-life of Lithium in the blood is 24 hours, if we use for unity of time the day, the ODE giving the excretion of lithium (without further input) is $\dot{x}_2 = -a\,x_2$, which gives a half-life of $\log(2)/a$ we have

$$b = \log(2) \approx 0.6931472$$

The half-life in the digestive tract is 3 hours which gives

$$a = \log(2)/(3/24) \approx 5.5451774$$

The lithium is administrated in one unity dose, passing during one half-hour, every half-day. Then u(t) is periodic function on 1/2 day, constant on 1/48 and taking zero value on the remaining of the day. The constant is such that the integration on a half-day (the quantity of drug metabolized) is 1.

Then $u(t)$ of period $1/2$, is defined on $[0, 1/2]$ by

$$u(t) = \begin{cases} 48 & \text{if } 0 \leq t \leq \frac{1}{48} \\ 0 & \text{if } \frac{1}{48} < t < \frac{1}{2} \end{cases}$$

Clearly the RHS of the ODE is discontinuous. But the bad points of discontinuity are known.

programming u(t) in Scilab :
Once more there is a clumsy way to code $u(t)$. Doing by case ... and using `if` loops. There is a better manner, which has the great advantage to give a vectorized code, and moreover which is built-in, that is considerably quicker. The keys are the test function and the function `pmodulo`. The function `pmodulo` is vectorized and can be used to build any periodic function.
The function modulo (x,P) is

```
modulo(x,P)=x-P  .*  floor(x  ./  P )
```

That is pmodulo(x,P) is the remainder of the division with integer quotient of x by P (as period). The number pmodulo(x,P) satisfies

$$0 \leq pmodulo(x, P) < P$$

.
When you consider the grid $P\mathbb{Z}$, $modulo(x, P)$ is the distance of x to the nearest left number on the grid, i.e if n is the smallest integer such that

$$nP \leq x \ < (n+1)P$$

$$modulo(x, P) = x - nP$$

.
With these preliminaries it is now easy to code a periodic function, of period P which value is 1 on $[0, w]$ and taking value 0 on $]w, P]$. We call it $pulsep(t, w, P)$

```
function x=pulsep(t,w,p)
//give a step periodic function of the variable t of period p
//value 1 if  0 <=  t  <= w  and  0 if   w < t < p

x= bool2s(pmodulo(t,p) <= w );
```

For safety, we test is the remainder is less or equal to w, we obtain boolean variables, $\%T$ or $\%F$, and we convert in variables 1 or 0. See the comments in the code of the function k_i in section (5.4.3).

Now it is simple to code the Lithium model

```
function xdot= phk(t,x)
// pharmacokinetics example from Shampine
//plot of numerical solutions of ODE pp105
b =log(2); a=log(2)/(3/24):

dotx=[-a 0; a -b]*x + 48 * [pulsep(t,1/48,1/2) ; 0];

//
function pulsep(t,w,p)
x=bool2s( pmodulo(t,p) <= w );
```

We now write a script for testing some solvers. We test the default solver RK, Adams and RKF (Shampine and Watts).

```
// compute solution for the pharmaco-kinetics problem
//
;getf("/Users/sallet/Documents/Scilab/phk.sci");

//
x0=[0;0];
t0=0;
tf=10;
T=linspace(t0,tf,100);
T=T(:);
X=ode(x0,t0,T,phk);
xset('window',0)
xbasc()
plot2d('RK',T,X(:,2)',2)

X1=ode('rkf',x0,t0,T,phk);
xset('window',1)
xbasc()
plot(T,X1(:,2)',3)
```

```
X2=ode('adams',x0,t0,T,phk);
xset('window',2)
xbasc()
plot(T,X2(:,2)',3)
```

For the method “RKF”

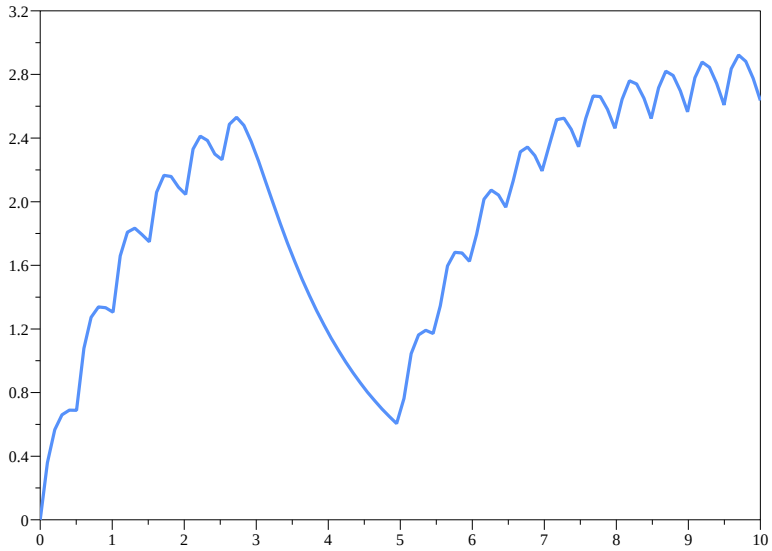

Fig. 25 – concentration of Li in blood, by RKF method

For the method “RK”

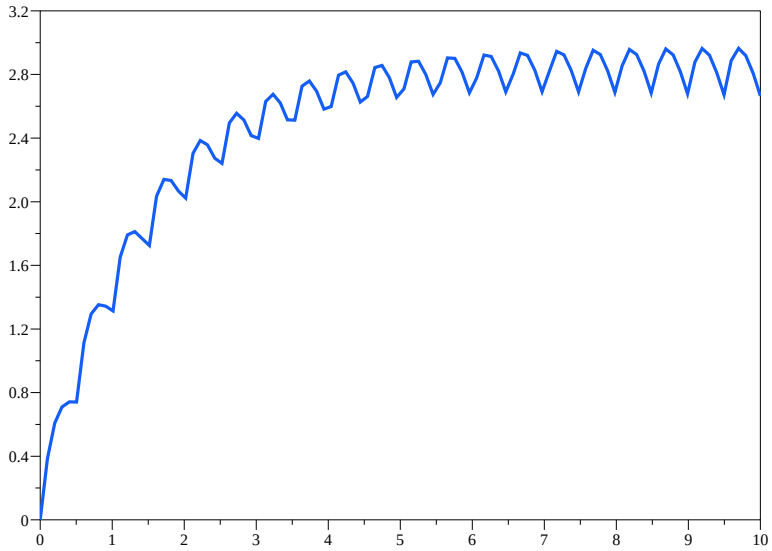

FIG. 26 – concentration of Li in blood, by RK method

For the method “Adams”

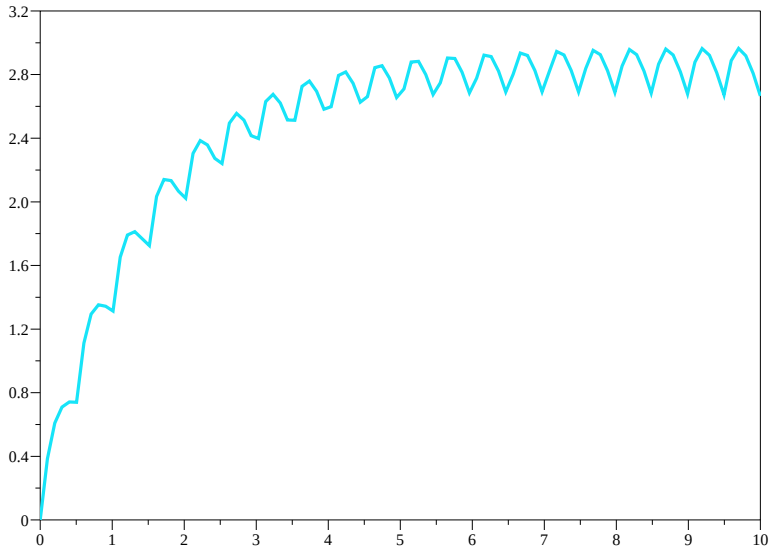

FIG. 27 – concentration of Li in blood, by Adams method

The solver gives contradictory results. To understand what happens, we

should have plot also the first component :

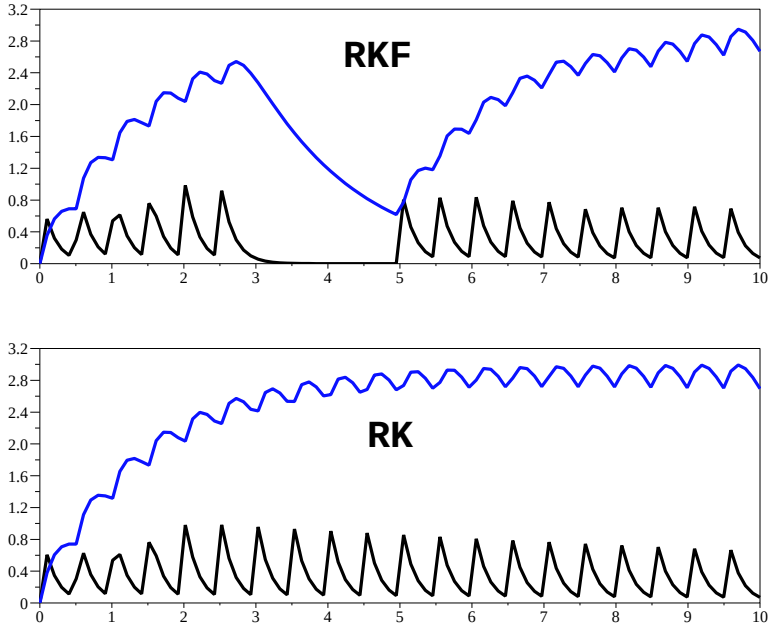

FIG. 28 – Plotting the two components

It is clear that the "RKF" method, which was substantially quicker , has tacken big steps, too much big steps, and has missed some absorption of tablets, namely at time 3, 3.5, 4 and 4.5.
To get the more precise result of the solution is to integrate on interval, stop at bad points and restart.

Integrating on interval of smoothness for f :
We shall now integrate on the intervals of smothness for $u(t)$, for 10 days. We have to integrate on 20 intervals, defined by the mesh

$$[0, 1/481/2, \cdots, p/2 + 1/48, p + 1, \cdots, 19/2 + 1/48, 10]$$

On intervals of the kind $[p/2, p/2 + 1/48]$ the fonction u has value 48, on the intervals of the kind $[p/2 + 1/48, p + 1]$ the function has value 0.
To define this interval, we can use, for example , a for loop. But in Scilab, once again , this is clumsy : a principle is to use the vectorized capacity of

Scilab and avoid loops which are computer time consuming. The idea is to take the interval $[0, 1/48]$ and to make 20 translations of steps $1/2$. To begin we concatene 20 copies of this interval. This done by using the kronecker product of matrices :

```
-->I=[0,1/48] ;
-->J=ones(1,20);
-->A=kron(J,I);
```

now to each block we have to make a translation of a multiple of $1/2$.

```
-->K=ones(1,2);
-->L=0:1/2:19/2 ; // translation
-->M=kron(L,K); //blocks of translations
-->mesh=A+M;
-->mesh=[mesh,10]; // the last point is missing.
```

We obtain a vector of mesh points of length 48.
Now we have the mesh we can integrate. to begin we prepare the output. We take $n = 100$ steps by interval.

```
t0=0;
x0=[0;0];
xout=x0'
tout=t0;
n=100;
```

We set the loop

```
// a loop for  solving the problem
// we integrate on a  half a day,
//this half day is divised in two
// integration intervals

for i=1:20

//integration with lithium absorption
// on the first half--day
T=linspace(mesh(2*i-1),mesh(2*i),n);
```

```
X=ode(x0,t0,T,odephk2);
tout=[tout;T(2:$)'];
xout=[xout ;X(:,2:$)'];
//new IV
t0=T($);
x0=X(:,$);

// integration without lithium absorption
//on the second half day
T=linspace(mesh(2*i),mesh(2*i+1),n);
X=ode(x0,t0,T,odephk1);
tout=[tout;T(2:$)'];
xout=[xout ;X(:,2:$)'];
//new IV
t0=T($);
x0=X(:,$);

end
xbasc()
plot2d(tout,xout(:,2))
```

So we obtain the following plot :

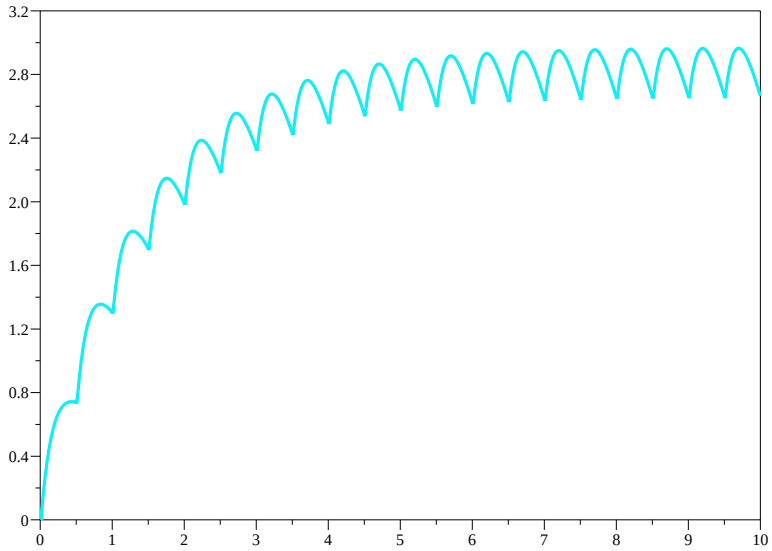

FIG. 29 – concentration of Li in blood, by interval

We can compare the different plots, in blue the "true value" integrating by interval, in red with dots the method RK, in green the method RKF.
So we obtain the following plot :

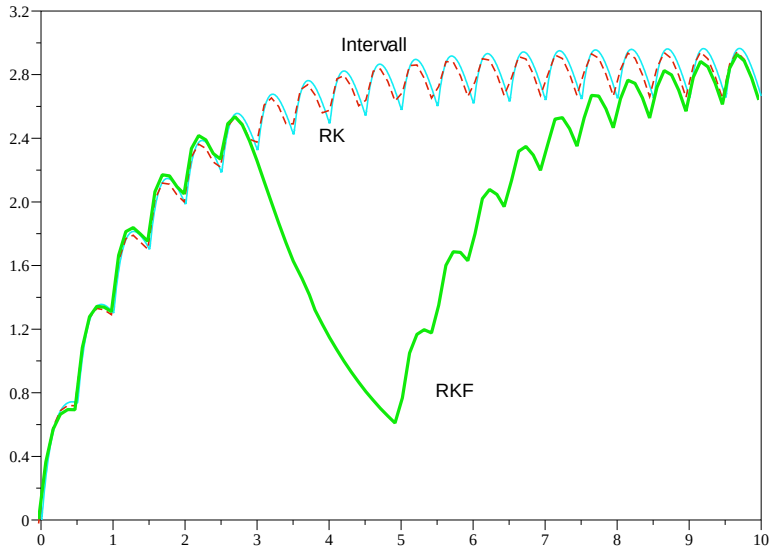

FIG. 30 – comparison of methods

We see that the RKF method, h as bad results, this is due to the skipping of some absorption of Lithium, since RKF choose too big integrating steps (when this is not precised by the user) . The RK and Adams methods has good behavior.
Plot the first component and compare.

5.7 Event locations

The preceding example of the absorption of Lithium is a case of a RHS discontinuous. The "bad" points are known. In this case breaking up the the integration interval solve the problem. Actually breaking up the problem gives rise to a restart. But there is some cases where there can be difficult to approach the problem in this way. If there are a great many places where singularities occur, the restarts becomes expensive.
All this presumes that we know where the bad places occur. When this depends of the solution itself the matter becomes much more difficult because it is necessary to locate bad points. This situation is quite frequent in applications where the RHS depends on a switching function. The problem is to detect when the solution is crossing a surface of discontinuity.

A common example is a problem which can be written

$$\begin{cases} \dot{x} = f_1(x) & \text{if } \ g(x) > 0 \\ \dot{x} = f_2(x) & \text{if } \ g(x) < 0 \end{cases} \tag{28}$$

The function g is called a switching function, and the hypersurface of equation $g(y) = 0$ is the surface of discontinuities. We shall come back on this point later on, when considering the Coulomb example.

Scilab has a event location capabilities.
We shall illustrate this by two examples and explain how to use these capabilities. The two first one are given by a bouncing ball, the third is an example taken from HW of the friction Coulonb's law.
We describe first the syntax of "root" :

5.7.1 Syntax of "Roots "

The syntax is in the window command :

```
[x,rd]=ode('root',x0,t0,tf ,rtol ,atol,f ,jac ,ng,g )
```

The parameters are
- x0 : real vector or matrix (initial conditions).
- t0 : real scalar (initial time).
- t : real vector (times at which the solution is computed).
- f : external i.e. function which gives the RHS of the ODE
- rtol,atol : real constants or real vectors of the same size as x.
- jac : external i.e. function which gives the Jacobian of the RHS of the ODE.
- ng : integer. The number of constraints to be satisfied . The dimension of g
- g : external i.e. function or character string or list.

5.7.2 Description

With this syntax (first argument equal to "root") ode computes the solution of the differential equation $\dot{x} = f(t, x)$ until the state $x(t)$ crosses the surface in $\mathbb{R}^n$ of equation $g(t, x) = 0$.
Needless to say that we must choose the final time `tf` safely, i.e. to be sure that we encounter the surface.

The function g g should give the equation of the surface. It is an external i.e. a function with specified syntax, or the name of a Fortran subroutine or a C function (character string) with specified calling sequence or a list.
If g is a function the syntax should be as follows : $z = g(t, x)$ where t is a real scalar (time) and x a real vector (state). It returns a vector of size ng which corresponds to the ng constraints. If g is a character string it refers to the name of a Fortran subroutine or a C function, with the following calling sequence : $g(n, t, x, ng, gout)$ where ng is the number of constraints and $gout$ is the value of g (output of the program).
If we must pass parameters to g, the function g can be a list with the same conventions as for f apply (see section (5.5)) or `Jac`

Ouput rd is a $1 \times k$ vector. The first entry contains the stopping time. Other entries indicate which components of g have changed sign. k larger than 2 indicates that more than one surface ($(k-1)$ surfaces) have been simultaneously traversed.

The output x is the first solution (computed) reaching the surface defined by g

5.7.3 Precautions to be taken in using "root"

Missing a zero :
Some care must be taken when using "root" for event location. The solver use the root finding capabilities of Scilab . But in other hand the solver adjust the steps according the tolerance required. When the ODE is particularly smooth and easy the solver can take great step and miss some zeros of the function.
To show what can happen, we use an explicit example. We define a function with some zeros near the origin, and constant outside the neighborhood of the origin, and we choose a very simple ODE, namely, `xdot =0`!
The function g is obtain by a a sinusoid, $\sin(50\pi t)$, with zeros at each step $1/50$, we multiply this sinusoid by the characteristic function of the interval $[0, 0.05]$ and we add the characteristic function of the interval $[0.05, \infty]$. Then our function g depending only of time, has 2 zeros located at $0, 0.02, 0.04$. This function is constant, equal to 1 on $[0.05, \infty]$.
With this we use "root"

```
-->deff('xdot=testz(t,x)','xdot=0')
-->deff('y=g(t,x)','y=sin(50*%pi*t).*(t>0).*(t<.05)+...
bool2s(t> =.05)')
-->[xsol,rd]=ode('root',x0,t0,tf,testz,1,g);
-->rd
 rd  =

     []
```

The vector rd is empty. No zero has been detected!

The remedy for this is to adjust the maxstep used by the solver (see section (5.3.4). We use %ODEOPTIONS, and set hmax to 0.01 :

```
-->%ODEOPTIONS=[1,0,0,.01,0,2,10000,12,5,0,-1,-1];

-->[xsol,rd]=ode('root',x0,t0,tf,testz,1,g);
```

```
-->rd
 rd  =

!   0.02    1. !
```

The solver stop at the first zero of g. We can restart with this new I.V.

```
-->t0=rd(1);x0=xsol;

--[xsol,rd]=ode('root',x0,t0,tf,testz,1,g);

-->rd
 rd  =

!   0.04    1. !
```

To obtain the second zero.

¡

Restarting from an event location :
There is here another subtlety. Sometimes when we ask Scilab to find the next event, without precautions, we get a warning message :

```
lsodar-  one or more components of g has a root
      too near to the initial point
```

That is, in other words, the root finding capabilities find the event you have previously detected. In our example, this has not occurred. But it can happens (see the next examples). You can remedy to this situation with three ways

1. Adjust the minimum step `hmin`, `%ODEOPTIONS(5)`, which is 0 by default (see section (5.3.5)
2. Adjust the first step tried `h0`, `%ODEOPTIONS(3)`, which is 0 by default (see section (5.3.3)
3. Or integrate a little bit the solution, and restart from this new point to search the new events. We have to use this trick in the example of the ball on a ramp, in section (5.7.5), and in the example of the dry friction of Coulomb's law.

Remark 5.1 *:*
Modifying `h0` *or* `hmin` *is not without harmlessness. You can miss some singularities of your system when one of these two quantities are too big.*

5.7.4 The bouncing ball problem

A ball is thrown in a vertical plane. The position of the ball is represented by the abscissa $x(t)$ and his height $h(t)$.
The ODE that $(x(t), h(t))$ satisfies is coded by

```
 function xdot=fallingball2d(t,x)

xdot=[x(2); 0; x(4); -9.81]
```

At time $t_0 = 0$, the ball is thrown with initial condition $x(0) = 0$, $\dot{x}(0) = 5$, $h(0) = 0$ and $\dot{h}(0) = 10$.
We give numerically, the time at which the ball hits the ground.

```
 -->;getf("/Users/sallet/Documents/Scilab/fallingball2d.sci");

-->t0=0;

-->x0=[0;5;0;10];

-->tf=100;

-->deff('z=g(t,x)','z=x(3)')

-->[x,rd]=ode('roots',x0,t0,tf,fallingball2d,1,g);

-->rd(1)
 ans  =

    2.038736

 -->x
 x  =
```

```
!   10.19368 !
!   5.       !
!   0.       !
! -10.       !
```

What are the value of $x, \dot{x}, h, \dot{h}$? they are obtained by calling x, so

$$x = 10.19...; \dot{x} = 5; y = 0; \dot{y} = -10$$

When the ball hits the ground, it bounces, or in other words , if t_{hit} is the time of hit, the arrival speed is $(\dot{x}(t_{hit}), \dot{h}(t_{hit}))$ and becomes

$$(\dot{x}(t_{hit}), -k\ \dot{h}(t_{hit}))$$

Where k is a coefficient of elasticity. We set $k = 0.8$.
We write the following script

```
// script for a falling ball
//
// script for a falling ball
//
;getf("/Users/sallet/Documents/Scilab/fallingball.sci");
tstart = 0;
tfinal = 30;
x0 = [0; 20];

deff('z=g(t,x)','z=x(1)')

tout = tstart;
xout = x0.';
teout = [];
xeout = [];
for i = 1:10
  //solve till the first stop
  [x,rd]=ode('root',x0,tstart,tfinal,fallingball,1,g);
   //accumulate output
```

```
   //
T=linspace(tstart,rd(1),100)';
T=T(:);
X=ode(x0,tstart,T,fallingball);
tout=[tout;T(2:$)];
xout=[xout;X(:,2:$)'];
teout=[teout;rd(1)];
xeout=[xeout;x'];

// new IV (initial values)

x0(1)=0;
x0(2)=-.9*x(2);
tstart=rd(1);
end
xset('window',1)
xbasc()
plot2d(tout,xout(:,1))
plot2d(teout,zeros(teout),-9,"000")
```

When the script is executed we obtain a tableau of the 10 first times where the ball hits the ground, give the values of $x, \dot{x}, h, \dot{h}$ at this events. The time is given in the first column

```
-->[teout xeout]
 ans  =

!   2.038736     10.19368     5.    0.          -10.         !
!   3.6697248    18.348624    5.    0.          - 8.         !
!   4.9745158    24.872579    5.    0.          - 6.4        !
!   6.0183486    30.091743    5.  - 3.109E-15   - 5.12       !
!   6.0183486    30.091743    5.    1.061E-15     4.096      !
!   6.0183486    30.091743    5.  - 1.850E-15   - 3.2768     !
!   6.0183486    30.091743    5.    8.147E-16     2.62144    !
!   6.0183486    30.091743    5.  - 9.009E-17   - 2.097152   !
!   6.0183486    30.091743    5.    6.549E-16     1.6777216  !
!   6.0183486    30.091743    5.  - 1.103E-15   - 1.3421773  !
```

With accumulation of output (see the script) we obtain the following plot

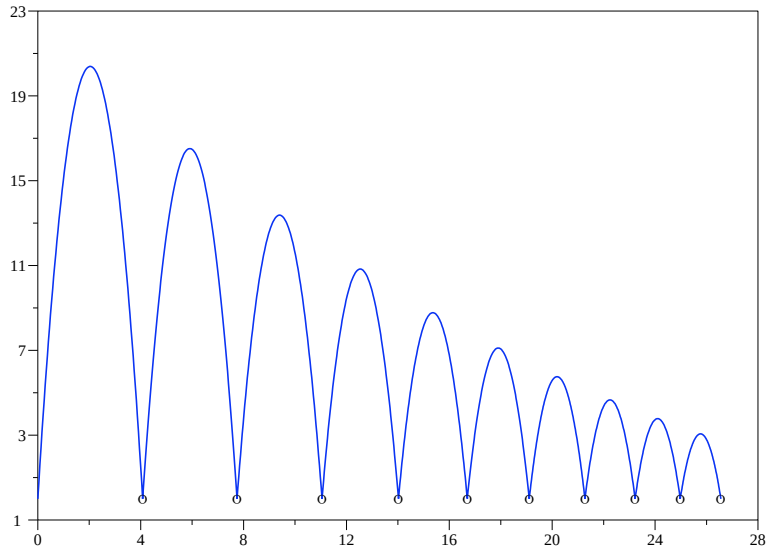

FIG. 31 – A Bouncing ball on the ground

5.7.5 The bouncing ball on a ramp

We consider the same problem, but this time the ball is bouncing on a ramp (equation $x = y = 1$)
The rule for restarting is a reflection rule on a surface : the new starting vector speed is symmetric with respect to the normal at the hitting point, and reduced accordingly to the reduction coefficient.
We need here a refinement : when we ask Scilab to find the next event, without precautions, we get , as we have already said a warning message :

```
lsodar-  one or more components of g has a root
      too near to the initial point
```

Simply when starting from the new initial point, which is precisely an “event ” location point, the solver find **this** point.

Then we need to advance a little bit the solution et restart from this point for searching for the new events.

```
// script for a falling ball
//
;getf("/Users/sallet/Documents/Scilab/fallingball2d.sci");
t0 = 0;
tfinal = 100;
x0 = [0;0;2;0];

deff('z=g(t,x)','z=[x(1)+x(3)-1;x(3)]')

tout = t0;
xout = x0.';
teout = [];
xeout = [];
k=0.422;
eps=0.001;
A=[1 0 0 0 ;0 0 0 -k;0 0  1 0; 0 -k 0 0];
for i = 1:7
  //solve till the first stop
  [x,rd]=ode('root',x0,t0,tfinal,fallingball2d,2,g);
   //accumulate output
   //
T=linspace(t0,rd(1),100)';
T=T(:);
X=ode(x0,t0,T,fallingball2d);
tout=[tout;T(2:$)];
xout=[xout;X(:,2:$)'];
teout=[teout;rd(1)];
xeout=[xeout;x'];

// new IV (initial values)
//  must push a little bit the solution

x0=A*x;
t0=rd(1);
Z1=ode(x0,t0,t0+eps,fallingball2d);
```

```
tout=[tout;t0+eps];
xout=[xout;Z1'];

// new I.V.
x0=Z1(:,$);
t0=t0+eps;
end
xset('window',1)
xbasc()
plot2d(xout(:,1),xout(:,3))
plot2d(xeout(:,1),xeout(:,3),-9,"000")
```

We obtain the plot

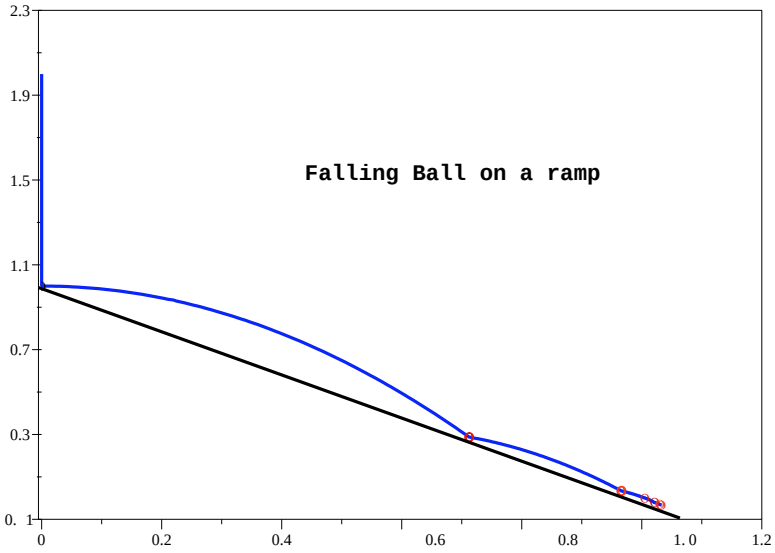

FIG. 32 – A bouncing ball on a ramp

Try the preceding code without "the little push" to see the reaction of Scilab. Try the other solutions : modifying `h0`, or `hmin`.

Suppose that in the preceding configuration, there is a vertical wall located at the end of the ramp. Compute the path of theball

In the first example, the ground is flat. Suppose that the ground is given by

$0.1\sin(5\pi t)$. The ball is thrown at time 0 with I.V. : $[2, 0.1]$. The coefficient of restitution is $k = 0.9$. Simulates the path of the ball.

5.7.6 Coulomb's law of friction

We consider the example

$$\ddot{x} + 2D\,\dot{x} + \mu\, sign(\dot{x}) + x = A\cos(\omega t) \tag{29}$$

The parameters are $D = 0.1$, $\mu = 4$, $A = 2$, and $\omega = \pi$, the initial values are $x(0) = 3$ and $\dot{x}(0) = 4$.

This equation model the movement of a solid of mass M moving on a line, on which acts a periodic force $A\cos(\omega t$, and subject to a dry friction force of intensity mu and to a viscous friction of intensity $2D$.
We write the equation in standard form

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -0.2\,x_2 - x_1 + 2\cos(\pi t) - 4sign(x_2) \end{cases} \tag{30}$$

The switching function is x_2, the surface is then the x_1–axis. When the solution crosses the x_1–axis different cases occur.
We follow a solution starting in the upper half–plane till it hits the manifold $x_2 = 0$. Let denote t_I the time where the solution reaches the manifold, $f_I(x,t) = -x_1 + 2\cos(\pi t) - 4$ and $f_{II}(x,t) = -x_1 + 2\cos(\pi t) + 4$ the different values of the vectors field on the x_1–axis.
Let look at the different vectors fields near the intersection with the x_1–axis :

1. $f_I(x, t_I) < 0$ and $f_{II}(x, t_I) < 0$. We continue the integration in the $x_2 < 0$ half–plane.
2. $f_I(x, t_I) < 0$ and $f_{II}(x, t_I) > 0$, the solution remains trapped in the manifold $x_2 = 0$ and $x_1 =$ Constant $= x_1(t_I)$ until one of the value of f_I or f_{II} changes sign and becomes of same positive sign, then the solution will leave the manifold and go toward the common direction given by the vector fields.

We shall now implement the solution of this problem.
We have for the differential equation 3 possibilities.

1. The term of the dry friction has value 4 in the expression of $\dot{x}_2$ in equation (30)

2. Or the term of the dry friction has value -4 in the expression of $\dot{x}_2$ in equation (30)
3. Or if the speed is 0 (we are then in the x_1–axis manifold) and if the components of two vector fields on the x_2 axis given by the formulas

$$-x_1 + 2\cos(\pi t) - 4$$

and

$$-x_1 + 2\cos(\pi t) + 4$$

have opposite sign, then we are trapped in the manifold $x_2 = 0$, and then the ODE is given by $\dot{x} = 0$.

We shall code, these 3 possibilities, by the introduction of a parameter `MODE`, which takes values $1, -1$ or 0, and is given by `sign(x_2)`
The event function is twofold. When we are not in the manifold $x_2 = 0$, the event is : **the component x_2 taking the value** 0. At this point we have to decide which way the solution of the ODE is going.

If the the components of the two vector fields on the x_2 axis, given by the 2 formulas $-x_1 + 2\cos(\pi t) - 4$ and $-x_1 + 2\cos(\pi t) + 4$, have same sign the solution goes through the x_1–axis and we choose the appropriate vector field, corresponding to `MODE=1` or `MODE=-1`.

If the two preceding expressions have opposite signs, then we set `MODE=0` and the next event that we must detect is the change of sign of the expression

$$(-x_1 + 2\cos(\pi t) - 4)(-x_1 + 2\cos(\pi t) + 4)$$

The event is then : **looking for a zero of this expression**.

With this analysis, an solution for the code could be

```
function xdot = coulomb(t,x,MODE)
// Example of a spring with  dry and viscous  friction
// and a periodic force
// MODE  describe the vector field
// MODE=0 we are on the x1 axis

  M=1; // Mass
 A=2;  // Amplitude of input force
```

```
 mu=4; // Dry Friction force
D=0.1;  // viscous force
 //
if MODE==0
   xdot = [0; 0];
   else
xdot=[x(2); (-2*D*x(2)- mu*MODE +A*cos(%pi*t)-x(1))/M];
end
//-------------------------------------------------------
//sub function for event location

function value=gcoul(t,x,MODE)
 M=1;  A=2; mu=4; D=0.1;

if MODE==0
F=( (-mu+A*cos(%pi*t)-x(1)) *( mu+A*cos(%pi*t)-x(1) ) );
   value = F ;
else
value=x(2);
end
```

Now we write a script for finding the events and accumulate the output. We have the same remark that we have to push a little bit the solution from the "event point ".

```
// script for plotting Coulomb example
//-----------------------------------------------------
clear

A = 2;  M = 1;  mu=4;  D=0.1;
t0=0;
tstart =t0;
tfinal = 10;
x0 = [3; 4];
;getf("/Users/sallet/Documents/Scilab/coulomb.sci");
// limiting hmax
%ODEOPTIONS=[1,0,0,.5,0,2,20000,12,5,0,-1,-1];
//
```

```
//-------------------------------------------------------
MODE=sign(x0(2));
tout = tstart;
xout = x0;
teout = [];
xeout = [];

while tfinal >= tstart
  // Solve until the first terminal event.

//updating value of MODE
list1=list(coulomb,MODE);
list2=list(gcoul,MODE);

// We need to push a little bit the integration
// in order that gcoul has not a zero too near
// the initial point
Tplus=tstart+0.05;
  xsolplus=ode(x0,tstart,Tplus,list1);

// Looking for a new zero of gcoul
x0=xsolplus;
tstart=Tplus;
[xsole,rd] = ode('root',x0,tstart,tfinal,list1,1,list2);

//If no event occur, then terminates till final time
if rd(1)==[] then
T=linspace(tstart,tfinal,100);
xsol=ode(x0,tstart,T,list1);
xout=[xout,xsol(:,2:$)];
tout=[tout,T(2:$)];
break,
end;

// If event occurs accumulate events
xeout=[xeout,xsole];
teout=[teout,rd(1)];
```

```
// Accumulate output.
T=linspace(tstart,rd(1),100);
xsol=ode(x0,tstart,T,list1);
 tout = [tout, T(2:$)];
  xout = [xout, xsol(:,2:$)];

//start at end of the preceding integration
  tstart = rd(1);
  x0 = [xsole(1);0];
  a1=-mu+A*cos(%pi*tstart)-x0(1);
  F=(-mu+A*cos(%pi*tstart)-x0(1) ) * ...
(mu+A*cos(%pi*tstart)-x0(1));

if MODE==0
  MODE=sign(a1);
   elseif F<=0
  MODE= 0 ;
   else
  MODE=sign(a1);
    end
end
// plotting
xset('window',1)
xbasc()

// plot the force
t = t0 + (0:100)*((tfinal-t0)/100);
plot2d(t,A*cos(%pi*t),5);
//plot the solution
plot2d(tout',xout')

//Mark the events with a little "o"
plot2d(teout',xeout',style=[-9,-9])
```

Running this script gives the times and location of events :

```
-->[teout',xeout']
 ans  =
```

```
!   0.5628123    4.2035123   - 2.845E-16 !
!   2.0352271    3.2164227     7.683E-16 !
!   2.6281436    3.2164227     0.        !
!   3.727185     2.9024581     7.175E-16 !
!   4.6849041    2.9024581     0.        !
!   5.6179261    2.7281715     1.410E-15 !
!   6.7193768    2.7281715     0.        !
!   7.5513298    2.6140528     9.055E-16 !
!   8.7436998    2.6140528     0.        !
!   9.5042179    2.5325592     9.558E-16 !
```

Notice that the second component x_2 is zero, but within precision machine accuracy. Notice also that we set to zero, in the script, this component when we restart from these points.
The script gives the following plot :

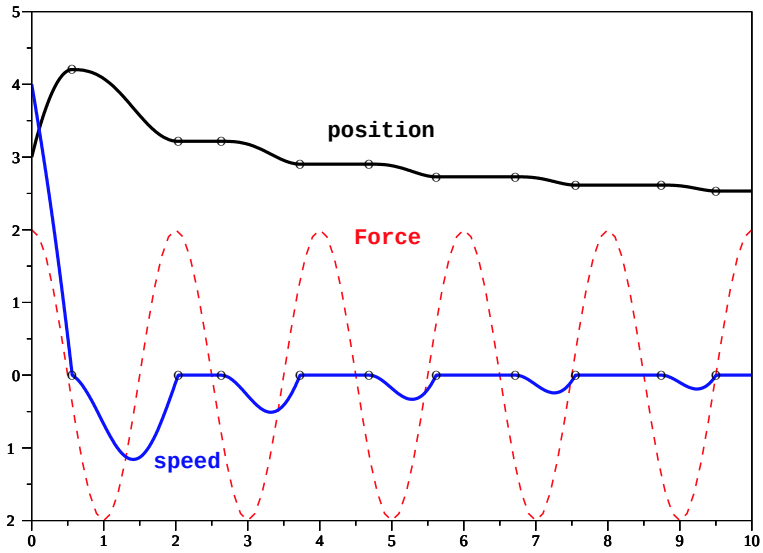

FIG. 33 – Solution of (30)

5.8 In the plane

Scilab has tools for ODE in the plane. We can plot vector fields in the plane, and it is possible to plot solution directly from a mouse click in the graphic window.

5.8.1 Plotting vector fields

CALLING SEQUENCE :

```
fchamp(f,t,xr,yr,[arfact,rect,strf])
fchamp(x,y,xr,yr,<opt_args>)
```

PARAMETERS :

- `f` : An external (function or character string) or a list which describes the ODE.
  - - It can be a function name `f`, where `f` is supposed to be a function of type $y = f(t, x, [u])$. `f` returns a column vector of size 2, `y`, which gives the value of the direction field `f` at point `x` and at time `t`.
  - It can also be an object of type `list`, `list(f,u1)` where `f` is a function of type `y=f(t,x,u)` and `u1` gives the value of the parameter `u`.
- `t` : The selected time.
- `xr,yr` : Two row vectors of size `n1` and `n2` which define the grid on which the direction field is computed.
- `<opt args>` : This represents a sequence of statements `key1=value1`, `key2=value2`,... where `key1, key2`,... can be one of the following : `arfact, rect, strf`

DESCRIPTION :

fchamp is used to draw the direction field of a 2D first order ODE defined by the external function `f`. Note as usual, that if the ODE is autonomous, argument `t` is useless, but it must be given.

Example 5.2 *:*
We consider the equation of the pendulum (5) and we plot the corresponding vector field

```
// plotting the vector field of the pendulum
//
// get  Equation ODE pendulum
```

```
;getf("/Users/sallet/Documents/Scilab/pendulum.sci");
//
n = 30;
delta_x = 2*%pi;
delta_y = 2;
x=linspace(-delta_x,delta_x,n);
y=linspace(-delta_y,delta_y,n);
xbasc()
fchamp(pendulum,0,x,y,1,...
[-delta_x,-delta_y,delta_x,delta_y],"031")
xselect()
```

Running this script give you the picture

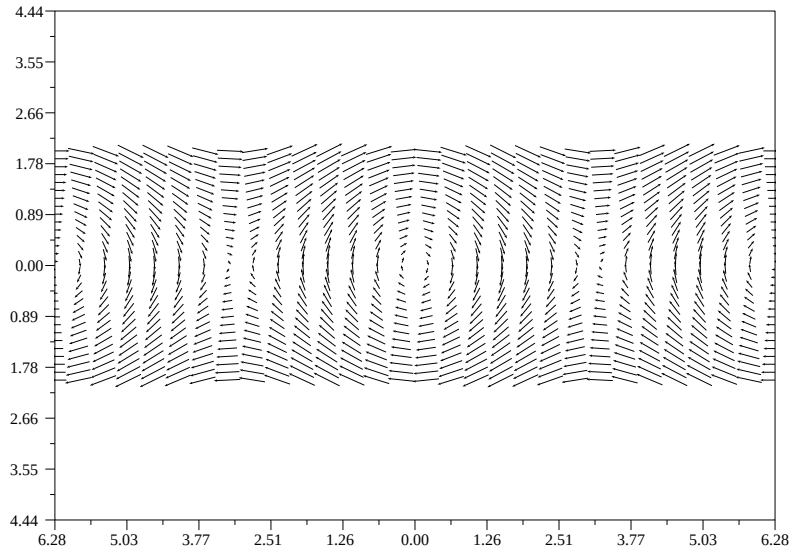

FIG. 34 – Vectors field pendulum

5.8.2 Using the mouse

The following section is taken from B. Pinçon 's book [21].
We can plot solution of ODE directly from the mouse. We use `xclick`. We get I.V. (for autonomous systems) with the mouse. Then we integrates from this I.V.

The primitive `xclick` is governed by
CALLING SEQUENCE :

```
[c_i,c_x,c_y,c_w,]=xclick([flag])
```

PARAMETERS :

- `c i` : integer, mouse button number.
- `c x,c y` : real scalars, position of the mouse.
- `c w` : integer, window number.
- `flag` : integer. If present, the click event queue is not cleared when entering xclick.

DESCRIPTION :
xclick waits for a mouse click in the graphics window.
If it is called with 3 left hand side arguments, it waits for a mouse click in the current graphics window.
If it is called with 4 left hand side arguments, it waits for a mouse click in any graphics window.
`c i` : an integer which gives the number of the mouse button that was pressed 0, 1 or 2 (for left, middle and right) or -1 in case of problems with xclick.
`c x,c y` : the coordinates of the position of the mouse click in the current graphics scale.
`c w` : the window number where the click has occurred.

Example 5.3 *:*
We consider the Brusselator equation (ODE), we plot the vector fields and solve with some clicks

The ODE in the plane. This is the Brusselator without diffusion. We pass a parameter

```
function [f] = Brusselator(t,x,eps)
//
f =[ - (6+eps)*x(1) + x(1)^2*x(2); (5+eps)*x(1) - x(1)^2*x(2)]
```

We now have the following script from [21] :

```
// Plotting Brusselator
;getf("/Users/sallet/Documents/Scilab/Brusselator.sci");
eps = -4;
P_stat = [2 ; (5+eps)/2];
```

```
// plotting limits
delta_x = 6; delta_y = 4;
x_min = P_stat(1) - delta_x; x_max = P_stat(1) + delta_x;
y_min = P_stat(2) - delta_y; y_max = P_stat(2) + delta_y;
n = 20;
x = linspace(x_min, x_max, n);
y = linspace(y_min, y_max, n);
// 1/ vector field plotting
xbasc()
fchamp(list(Brusselator,eps),0,x,y,1,[x_min,...
y_min,x_max,y_max],"031")
xfrect(P_stat(1)-0.08,P_stat(2)+0.08,0.16,0.16)
// critical point
xselect()
// 2/solving ODE
m = 500 ; T = 5 ;
rtol = 1.d-09; atol = 1.d-10; // solver tolerance
t = linspace(0,T,m);
color = [21 2 3 4 5 6 19 28 32 9 13 22 ...
18 21 12 30 27];
num = -1;
while %t

[c_i,c_x,c_y]=xclick();
if c_i == 0 then
plot2d(c_x, c_y, -9, "000")
u0 = [c_x;c_y];
[u] = ode('stiff',u0, 0, t, rtol, atol, list(Brusselator,eps));
num = modulo(num+1,length(color));
plot2d(u(1,:)',u(2,:)',color(num+1),"000")
elseif c_i == 2 then
break
end
end
```

Try it

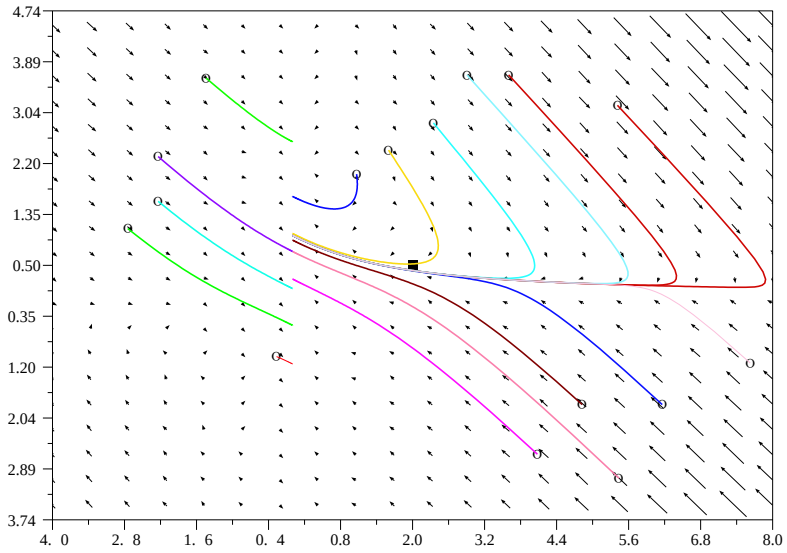

FIG. 35 – Brusselator trajectories from the mouse

5.9 test functions for ODE

To understand a solver's performances experimentation is a good way. Some collections of test problem exist. We give here some test problems. The reader is invited to test his ability with Scilab on these problems.

5.9.1

$$\begin{cases} \dot{x} = \frac{4t^2 - x}{t^4 - 1} x \\ x(2) = 15 \end{cases}$$

To integrate on $[0, 10]$, $0, 30]$, $[0, 50]$
The solution is the polynomial

$$x(t) = 1 + t + t^2 + t^3$$

$$\begin{cases} \dot{x} = \frac{4t^2 - x}{t^4 - 1} x \\ x(2) = 15 \end{cases}$$

To integrate on $[0, 10]$, $0, 30]$, $[0, 50]$
The solution is the polynomial

$$x(t) = 1 + t + t^2 + t^3$$

5.9.2

$$\begin{cases} \dot{x} = x \\ x(0) = 1 \end{cases}$$

To integrate on $[0, 10]$, $0, 30]$, $[0, 50]$

5.9.3

$$\begin{cases} \dot{x} = -0.5x^3 \\ x(0) = 1 \end{cases}$$

To integrate on $[0, 20]$
The solution is

$$x(t) = \frac{1}{\sqrt{t+1}}$$

5.9.4

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -x_1 \\ \\ x_1(0) = 0 \ , \ x_2(0) = 1 \end{cases}$$

To integrate on $[0, 2\pi]$, $0, 6\pi]$, $[0, 16\pi]$

5.9.5

$$\begin{cases} \dot{x} = Ax \\ x(0) = (1, 0, \ldots, 0)^T \end{cases}$$

Where A is a $N \times N$ tridiagonal matrix.

$$A = -2 * eye(N, N) + diag(ones(N-1, 1), 1) + diag(ones(N-1, 1), 1)$$

To integrate on $[0, 10]$, $0, 30]$, $[0, 50]$ for $N = 50$
The solution is

$$x(t) = Q\ diag(exp(\lambda_i.t))Q^{-1}.x_0$$

Where the eigenvalues

$$\lambda_i = -4\sin^2\left(\frac{i\pi}{2(N+1)}\right)$$

and R is an orthogonal matrix

$$R_{ij} = \sqrt{\frac{2}{N+1}}\sin\left(\frac{ij\pi}{N+1}\right)$$

5.9.6

$$\begin{cases} \dot{x}_1 = x_2x_3 \\ \dot{x}_2 = -x_1x_3 \\ \dot{x}_3 = -0.51x_1x_2 \\ x_1(0) = 0\ ,\ x_2(0) = 1\ \ x_3 = 1 \end{cases}$$

To integrate on $[0, 4K]$, $0, 6K]$, $[0, 28K$
The solution is

$$x_1(t) = sn(t, 0.51)\ \ x_2(t) = cn(t, 0.51)\ \ x_3(t) = dn(t, 0.51)$$

With period `K=1.862640080233273855203`

5.9.7

$$\begin{cases} \ddot{x}_1 = -\frac{x_1}{r^3} \\ \ddot{x}_2 = -\frac{x_2}{r^3} \\ \\ r = sqrtx_1^2 + x_2^2 \\ x_1(0) = 1\ ,\ \dot{x}_1(0) = 0 \\ x_2(0) = 0\ ,\ \dot{x}_2(0) = 1 \end{cases}$$

To integrate on $[0, 2\pi]$, $0, 6\pi]$, $[0, 16\pi]$
The solution is

$$x_1(t) = \cos(t) \;\; x_2 t) = \sin(x)$$

5.9.8

$$\begin{cases} \ddot{x}_1 = -\frac{x_1}{r^3} \\ \ddot{x}_2 = -\frac{x_2}{r^3} \\ \\ r = sqrtx_1^2 + x_2^2 \\ x_1(0) = 0.4 \ , \ \dot{x}_1(0) = 0 \\ x_2(0) = 0 \ , \ \dot{x}_2(0) = 2 \end{cases}$$

To integrate on $[0, 2\pi]$, $0, 6\pi]$, $[0, 16\pi]$
The solution is

$$x_1(t) = \cos(u) - 0.6 \;\; x_2 t) = 0.8 \sin(u)$$

where u is the solution of $u = -0.6 \sin u$

5.9.9 Arenstorf orbit

$$\begin{cases} \ddot{x}_1 = 2\dot{x}_2 + x_1 - \mu^* \frac{x_1+\mu}{r_1^3} - \mu \frac{x_1-\mu^*}{r_2^3} \;\; \ddot{x}_2 = -2\dot{x}_1 + x_2 - \mu^* \frac{x_2}{r_1^3} - \mu \frac{x_2}{r_2^3} \\ \\ r_1 = \sqrt{(x_1+\mu)^2 + x_2^2} \\ r_2 = \sqrt{(x_1-\mu^*)^2 + x_2^2} \\ x_1(0) = 0.4 \ , \ \dot{x}_1(0) = 0 \\ \mu = 1/82.45; \mu* = 1 - \mu \\ \\ x_1(0) = 0 \ , \ \dot{x}_1(0) = 0 \\ x_2(0) = 0 \ , \ \dot{x}_2(0) = -1.0493575098303199 \end{cases}$$

There are the equations of motion of a restricted three body problem. There is no analytic expression available for the solution. The error is measured at the end of the period T

$$T = 6.19216933131963970$$

5.9.10 Arenstorf orbit

$$\begin{cases} \ddot{x}_1 = 2\dot{x}_2 + x_1 - \mu^* \frac{x_1+\mu}{r_1^3} - \mu \frac{x_1-\mu^*}{r_2^3} \;\; \ddot{x}_2 = -2\dot{x}_1 + x_2 - \mu^* \frac{x_2}{r_1^3} - \mu \frac{x_2}{r_2^3} \\ \\ r_1 = \sqrt{(x_1+\mu)^2 + x_2^2} \\ r_2 = \sqrt{(x_1-\mu^*)^2 + x_2^2} \\ x_1(0) = 0.4 \;,\; \dot{x}_1(0) = 0 \\ \mu = 1/82.45; \mu* = 1 - \mu \\ \\ x_1(0) = 0.994 \;,\; \dot{x}_1(0) = 0 \\ x_2(0) = 0 \;,\; \dot{x}_2(0) = -2.00158510637908252 \end{cases}$$

Period T

$$T = 17.06521656015796255$$

5.9.11 The knee problem

[Dahlquist] :

$$\begin{cases} \varepsilon \dot{x} = (1-t)x - x^2 \\ \\ x(0) = 1 \end{cases}$$

The parameter ε satisfies $0 < \varepsilon \ll 1$
This is a stiff problem. $\varepsilon = 10^{-4}$, $\varepsilon = 10^{-6}$. To integrate on $[0, 2]$

5.9.12 Problem not smooth

[Dahlquist] :

$$\begin{cases} \ddot{x} = -x - sign(x) - 3\sin(2t) \\ \\ x(0) = 0 \; \dot{x}(0) = 3 \end{cases}$$

To integrate on $[0, 8\pi]$

5.9.13 The Oregonator

$$\begin{cases} \dot{x}_1 = 77.27(x_2 + x_1(1_8.375\,10^{-6}x_1 - x_2) \\ \dot{x}_2 = \frac{1}{77.27}(x_3 - (1 + x_1)x_2 \\ \dot{x}_3 = 0.161(x_1 - x_2) \\ x_1(0) = 1 \;\; x_2(0) = 2 \;\; x_3(0) = 3 \end{cases}$$

To integrate on $[0, 30]$, $[0, 60]$
This is a stiff problem with a limit cycle.

Références

[1] P. Bogacki and L.F. Shampine. A 3(2) pair of Runge-Kutta formulas. *Appl. Math. Lett.*, 2(4) :321–325, 1989.

[2] P. Bogacki and L.F. Shampine. An efficient Runge-Kutta $(4, 5)$ pair. *Comput. Math. Appl.*, 32(6) :15–28, 1996.

[3] Peter N. Brown, George D. Byrne, and Alan C. Hindmarsh. VODE : A variable-coefficient ODE solver. *SIAM J. Sci. Stat. Comput.*, 10(5) :1038–1051, 1989.

[4] J.C. Butcher. *Numerical methods for ordinary differential equations.* Wiley, 2003.

[5] George D. Byrne and Alan C. Hindmarsh. Stiff ODE solvers : A review of current and coming attractions. *J. Comput. Phys.*, 70 :1–62, 1987.

[6] Gear C.W. *Numerical initial value problems in ODE.* Prentice Hall, 1971.

[7] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer methods for mathematical computations.* Prentice Hall, 1977.

[8] C. W. Gear. Algorithm 407 : Difsub for solution of ordinary differential equations [d2]. *Commun. ACM*, 14(3) :185–190, 1971.

[9] Ernst Hairer and Martin Hairer. GniCodes –Matlab programs for geometric numerical integration. In *Frontiers in numerical analysis*, 2002.

[10] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving ordinary differential equations. I : Nonstiff problems. 2. rev. ed.* Springer-Verlag, 1993.

[11] Ernst Hairer and Gerhard Wanner. *Solving ordinary differential equations. II : Stiff and differential-algebraic problems. 2nd rev. ed.* Springer-Verlag, 1980.

[12] Jack K. Hale. *Ordinary differential equations. 2nd ed.* R.E. Krieger Publishing Company, 1980.

[13] A.C. Hindmarsch and G.D. Byrne. Episode. In *Numerical methods for differential systems*, 1976.

[14] A.C. Hindmarsch and G.D. Byrne. Odepack, a systematized collection of ode solvers. In Eds. Scientific Computing, R. S. Stepleman et al., editor, *Scientific Computing*, pages 55–64. North-Holland Publ., Amsterdam, 1976.

[15] A.C. Hindmarsch and G.D. Byrne. Lsode and lsodi, two new initial value ordinary differential equation. *ACM SIGNUM*, 15 :10–11, 1980.

[16] Morris W. Hirsch and Stephen Smale. *Differential equations, dynamical systems, and linear algebra.* Academic Press, 1974.

[17] T.E. Hull, W.H. Enright, B.M. Fellen, and A.E. Sedgwick. Comparing numerical methods for ordinary differential equations. *SIAM J. Numer. Anal.*, 9 :603–637, 1972.

[18] David Kahaner, Cleve Moler, and Stephen Nash. *Numerical methods and software. With disc.* Prentice Hall, 1989.

[19] Fred T. Krogh. On testing a subroutine for the numerical integration of ordinary differential equations. *J. Assoc. Comput. Mach.*, 20 :545–562, 1973.

[20] F.T. Krogh. A test for instability in the numerical solution of ordinary differential equations. *J. Assoc. Comput. Mach.*, 14 :351–354, 1967.

[21] B. Pinçon . *Une introduction à Scilab.* IECN, 2000.

[22] Lawrence F. Shampine. *Numerical solution of ordinary differential equations.* Chapman and Hall, 1994.

[23] L.F. Shampine. What everyone solving differential equations numerically should know. In *Computational techniques for ordinary differential equations*, pages 1–17, 1980.

[24] L.F. Shampine and C.W. Gear. A user's view of solving stiff ordinary differential equations. *SIAM Rev.*, 21 :1–17, 1979.

[25] L.F. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB.* Cambridge University Press, 2003.

[26] L.F. Shampine and M.K. Gordon. *Computer solution of ordinary differential equations. The initial value problem.* W.H. Freeman, 1975.

[27] L.F. Shampine and S. Thompson. Event location for ordinary differential equations. *Comput. Math. Appl.*, 39(5-6) :43–54, 2000.

[28] L.F. Shampine, H.A. Watts, and S.M. Davenport. Solving nonstiff ordinary differential equations - the state of the art. *SIAM Rev.*, 18 :376–411, 1976.

[29] Andrew H. Sherman and Alan C. Hindmarsh. GEARS : A package for the solution of sparse, stiff ordinary differential equations. In *Initial value problems for ODE*, pages 190–200, 1980.