General Information about the Project

Contents

- Title:

Numerical exponential integrators for dynamical systems

- Researcher in charge:

André Salles Carvalho, Prof. Dr.

- Beneficiary:

Isabela Miki Suzuki

- Host institution:

Instituto de Matemática e Estatística at the Universidade de São Paulo

- Research team:

Isabela Miki Suzuki

Pedro S. Peixoto, Prof. Dr.

- Number of the research project:

2021/06678-5

- Duration:

1 August 2021 to 31 July 2023

- Period covered by this research report:

1 August 2021 to 31 July 2023

Summary of the proposed project

This is a scientific initiation project that proposes the deep study of some of the main methods of exponential integration for problems in dynamic systems, with emphasis on the paper [1]. Here, the undergraduate will study the construction, analysis, implementation and application of them and at the end, it is expected that she is familiar with modern techniques of numerical methods.

Keywords: exponential integrator, numerical methods, dynamical systems.

Project execution

Motivation - Stiffness

The reason for studying exponential methods is that those are good with **stiff differential equations** in terms of precision and how small the time step is required to be to achieve good accuracy.

Cauchy problem

A **Cauchy problem** is a ordinary differential equation (ODE) with initial conditions. Being its standard scalar form:

$$\begin{cases} y'(t) = f(y(t), t), t \in (t_0, T) \\ y(t_0) = y_0 \in \mathbb{K}, \end{cases}$$

with $\mathbb{K}$ a field, f function with image in $\mathbb{K}$ and $t_0, T \in \mathbb{R}$.

Sometimes, it is convenient to separate the linear part of f as indicated below:

$$f(y(t), t) = g(y(t), t) - \lambda y(t),$$

with $\lambda \in \mathbb{K}$ or $\mathscr{M}_{N\times N}(\mathbb{K})$.

So the system is:

$$\begin{cases} y'(t) + \lambda y(t) = g(y(t), t), t \in (t_0, T) \\ y(0) = y_0. \end{cases}$$

In this project, the stiff ones were those addressed.

Notation as in [1].

Stiffness

The error of the approximation given by a method trying to estimate the solution of a Cauchy problem is always given by a term multiplied by a higher derivative of the exact solution, because of the Taylor expansion with Lagrange form of the remainder. In that way, if that is enough information about this derivative, the error can be estimated.

If the norm of the derivative increases with the time, but the exact solution doesn't, that is possible that the error dominates the approximation and the precision is lost. Those problems are called **stiff equations**.

Between them, there are the **stiff differential equations**, that have exact solution given by the sum of a $transient\ solution$ with a $steady\ state\ solution$.

The **transient solution** is of the form:

$$e^{-ct}, \text{with c} >> 1,$$

which is known to go to zero really fast as t increases. But its nth derivative

$$\mp c^n e^{-ct}$$

doesn't go as quickly and may increase in magnitude.

The **steady state solution**, however, as its name implies, have small changes as time passes, with higher derivative being almost constant zero.

In a system of ODE's, these characteristics are most common in problems in which the solution of the initial value problem is of the form

$$e^A$$

being A a matrix such that λ_{min} and λ_{max} are the eigenvalue with minimum and maximum value in modulus and $\lambda_{min} << \lambda_{max}$. On the bigger magnitude eigenvalue direction, the behaviour is very similar to the transient solution, having drastic changes over time and on the smaller one, comparing to that, changes almost nothing as times passes, like the steady state solution.

Work around these problems and being able to accurately approximate these so contrasting parts of the solutions requires more robust methods than the more classic and common one-step methods addressed at the beginning of the study of numerical methods for Cauchy problems. For the systems, it is also required that that is a precise way to calculate the exponential of a matrix.

In this project, we studied the **exponential methods**, their capabilities to deal with these problems and the comparision with other simpler methods.

Definition from [2].

Classical methods

In order to show that the exponential methods improve in dealing with Stiff problems, that is necessary to know how the previows methods deal with them, so a review on the theory of the classical methods is made in this chapter. In particular there will be focus on the one step methods. All the information is from [3].

One step methods for ODE

In order to find a approximation for the solution of the problem $\begin{cases} y'(t) = f(t, y(t)), t \in [t_0, T] \\ y(t_0) = y_0, \end{cases}$

they are of the form:

$$y_{k+1} = y_k + h\phi(t_k, y_k, t_{k+1}, y_{k+1}, h),$$

with

$$\begin{aligned} k = 0, 1, \ldots, n-1; \\ N \in \mathbb{N}; h = \frac{T - t_0}{N}; \\ \{t_i = t_0 + ih : i = 0, 1, \ldots, N\}; \\ y_n \approx y(t_n). \end{aligned}$$

To analyse the method, there is a model problem

$$\begin{cases} y'(t) = -\lambda y(t) \; ; t \in [t_0, T] \\ y(t_0) = y_0, \end{cases}$$

whose solution is $y(t) = y_0 e^{-\lambda(t-t_0)}$ with $\lambda > 0$.

If that is possible to manipulate the method so that, for this problem, can be written as
$y_{k+1} = \zeta(\lambda, h) y_k,$

then $\zeta(\lambda, h)$ is called **amplification factor** of the method.

By induction, it gives

$$y_{k+1} = \zeta(\lambda, h)^{k+1} y_0.$$

It is well known that this expression only converges as k goes to infinity if $|\zeta(\lambda, h)| < 1$

and then converges to zero.

When it occurs, i.e.,

$$k \to \infty \Rightarrow y_k \to 0$$

such as the exact solution

$$y(t) = y_0 e^{-\lambda(t-t_0)},$$

it is said that there is $\mathbf{stability}$.

The interval with the values of λh such as

$$|\zeta(\lambda, h)| < 1,$$

is called $\mathbf{interval\ of\ stability}$.

And if the interval of stability contains all the points z such that

$$Re(z) < 0,$$

the method is said $\mathbf{A\text{-}stable}$.

The reason for taking this specific problem is that it models the behaviour of the difference between the approximation and the solution on a small neighbourhood of any Cauchy problem:

Taking

$$\begin{cases} y'(t) = f(y(t), t), t \in (t_0, T) \\ y(t_0) = y_0 \in \mathbb{K} \end{cases}$$

and a approximation z of the solution y, doing $\$\sigma(t) = z(t) - y(t) \Rightarrow\$$

$$\dot{\sigma}(t) = \dot{z}(t) - \dot{y}(t) = f(z(t), t) - f(y(t), t) \Rightarrow$$

$$\dot{\sigma}(t) + \dot{y}(t) = \dot{z}(t) = f(z(t), t) = f(y(t) + \sigma(t), t)$$

$$= f(y(t), t) + \sigma(t)\frac{\partial f}{\partial y} + O(\sigma^2(t)),$$

so

$$\begin{cases} \dot{\sigma}(t) \approx \sigma(t)\frac{\partial f}{\partial y}(y(t), t) \\ \sigma(t_k) = \sigma_k. \end{cases}$$

Other important definitions are:

Local truncation error: Is the difference between the exact expression and its numerical approximation in a certain point and with a certain domain discretization. If the domain is equally spaced by h is often denoted by $\tau(h, t_0)$ being t_0 the point.

Order of the local truncation error: the local truncation error (which depends on the h spacing of the discretized domain) $\tau(h)$ has order $n \in \mathbb{N}$ if $\tau(h) = O(h^n)$, i.e., if there is constant $M \in \mathbb{R}$ and $h_0 \in \mathbb{R}$ such that $\tau(h) \leq Mh^n, \forall h \leq h_0$.

Global error: Is the difference between the approximation given by the method for the solution of the problem on a certain point and the exact one (unlike the local truncation error, here we take the solution we got, not the expression used to find the approximation).

Consistency: The method is said consistent if $\lim_{h\to 0} \frac{1}{h}\tau(h, x_0) = 0$.

Obs.: For consistency, we usually only analyse for the linear part of the Cauchy problem, since this is the part that most influences in the consistency.

Order of consistency: is the smallest order (varying the points at which the local error is calculated) of the local truncation error.

Convergence: A numerical method is convergent if, and only if, for any well-posed Cauchy problem and for every $t \in (t_0, T)$,

$$\lim_{h\to 0} e_k = 0$$

with $t - t_0 = kh$ fixed and e_k denoting the global error on t_k (following the past notation).

Theorem: A one-step explicit method given by

$$\begin{aligned} y_0 &= y(t_0) \\ y_{k+1} &= y_k + h\phi(t_k, y_k, h) \end{aligned}$$

such that ϕ is Lipschitzian in y, continuous in their arguments, and consistent for any well-posed Cauchy problem is convergent. Besides that, the convergence order is greater or equal to the consistency order.

Prove: [3] pág 29-31.

Examples

Euler method:

$$\phi(t_k, y_k, h) = f(t_k, y_k)$$

Modified Euler method:

$$\phi(t_k, y_k, h) = \frac{1}{2}[f(t_k, y_k) + f(t_{k+1}, y_k + hf(t_k, y_k))]$$

Midpoint method:

$$\phi(t_k, y_k, h) = f(t_k + \frac{h}{2}, y_k + \frac{h}{2}f(t_k, y_k))$$

Classic Runge-Kutta (RK 4-4):

$$\begin{aligned} \phi(t_k, y_k, h) = \frac{1}{6}(\kappa_1 + 2\kappa_2 + 2\kappa_3 + \kappa_4), \text{with} \\ \kappa_1 = f(t_k, y_k) \\ \kappa_2 = f(t_k + \frac{h}{2}, y_k + \frac{h}{2}\kappa_1) \\ \kappa_3 = f(t_k + \frac{h}{2}, y_k + \frac{h}{2}\kappa_2) \\ \kappa_4 = f(t_k + h, y_k + h\kappa_3) \end{aligned}$$

Further detailing the Euler mathod, explicit one-step method of

$$\phi(t_k, y_k, h) = f(t_k, y_k),$$

an analysis on stability, convergence and order of convergence is done on the appendix.

Important concepts for the study of exponential methods

In this chapter, a review on ϕ functions is done, because of its need when applying exponential methods in systems of ODE with initial value. Besides that, the format of the treated problem is shown.

It is worth remembering the importance of the matrix exponential for the linear problems treated here, which is in the appendix.

Linear problem

The linear problem is, following with the used notation:

$$\begin{cases} y'(t) + \lambda y(t) = g(y(t), t), t \in (t_0, T) \\ y(t_0) = y_0, \end{cases}$$

the one with $g \equiv 0$.

So, generaly, it is of the form:

$$\begin{cases} y'(t) = Ay(t), t \in (t_0, T) \\ y(t_0) = y_0, \end{cases}$$

with $A \in \mathscr{M}_{N\times N}(\mathbb{C}), N \in \mathbb{N}$ (remembering that a matrix 1×1 is simply a number).

Because $Ay(t)$ is a C^1 function in y, continuous in t and $t \in (t_0, T)$, a limited interval, by the existence and uniqueness theorem, there is a single solution of the problem.

Since

$$\frac{d}{dt}y_0e^{A(t-t_0)} \doteq \lim_{h\to 0}\frac{y_0e^{A(t-t_0+h)} - y_0e^{A(t-t_0)}}{h}$$

$$= y_0e^{(t-t_0)A}\lim_{h\to 0}\frac{e^{Ah} - I}{h}$$

$$= y_0e^{(t-t_0)A}\lim_{h\to 0}\frac{Ae^{Ah}}{1}$$

$$= y_0e^{(t-t_0)A}\frac{Ae^{A0}}{1}$$

$$= y_0e^{(t-t_0)A}AI = Ay_0e^{(t-t_0)A}$$

using L'Hôpital's rule on the second equality and noting that $A(t - t_0 + h) = A(t - t_0) + Ah$ and $A(t - t_0) \cdot Ah = (t - t_0)hAA = Ah \cdot A(t - t_0)$, so it was possible to apply the last proposition and make $e^{A(t-t_0+h)} = e^{A(t-t_0)} \cdot e^{Ah}$,

taking

$$y(t) = y_0e^{A(t-t_0)},$$

$$y'(t) = Ay_0e^{(t-t_0)A} = Ay(t) \text{ and } y(t_0) = y_0e^{(t_0-t_0)A} = y_0I = y_0.$$

So, the solution for the general linear problem is $y(t) = y_0e^{A(t-t_0)}$.

All information about matrix exponential is from [4].

General problem

Returning to the general case

$$\begin{cases} y'(t) + \lambda y(t) = g(y(t), t), t \in (t_0, T) \\ y(t_0) = y_0, \end{cases}$$

there is the variation of constants formula:

$$y(t) = e^{-t\lambda}y_0 + \int_{t_0}^{t} e^{-\lambda(t-\tau)}g(y(\tau), \tau)d\tau.$$

This well known implicit function, gives a solution of the problem.

If the integral part can be solved, there is a explicit solution, and if the problem satisfies the hypotesis of the Piccard problem, being Lipschitz in t, this is the only solution.

This formula is the basis of all the exponential methods.

ϕ functions

Before introducing exponential methods, it is useful to present the ϕ functions.

They are $\mathbb{C} \to \mathbb{C}$ functions defined as:

$$\phi_0(z) = e^z;$$

$$\phi_n(z) = \int_0^1 e^{(1-\tau)z}\frac{\tau^{n-1}}{(n-1)!}\,d\tau, n \geq 1.$$

By integration by parts,

$$\phi_{n+1}(z) = \int_0^1 e^{(1-\tau)z}\frac{\tau^n}{n!}\,d\tau$$
$$= -\frac{e^{(1-1)z}}{z}\frac{1^n}{n!} + \frac{e^{(1-0)z}}{z}\frac{0^n}{l!} - \int_0^1 -\frac{e^{(1-\tau)z}}{z}\frac{\tau^{n-1}}{(n-1)!}\,d\tau$$
$$= -\frac{1}{n!z} + \frac{1}{z}\int_0^1 e^{(1-\tau)z}\frac{\tau^{n-1}}{(n-1)!}\,d\tau.$$

Since

$$\phi_n(0) = \int_0^1 e^0\frac{\tau^{n-1}}{(n-1)!}\,d\tau = \int_0^1 \frac{\tau^{n-1}}{(n-1)!}\,d\tau = \frac{1^n}{n!} - 0 = \frac{1}{n!},$$

$$\phi_{n+1}(z) = \frac{\phi_n(z) - \phi_n(0)}{z}, \textbf{the recursive characterization}.$$

By the properties of integral [5], if $h \in \mathbb{R}^*, t_k \in \mathbb{R}, t_k + h = t_{k+1}$,

$$\phi_n(z) = \int_0^1 e^{(1-\tau)z}\frac{\tau^{n-1}}{(n-1)!}\,d\tau$$
$$= \frac{1}{h}\int_0^h e^{\frac{(h-\tau)z}{h}}\frac{\tau^{n-1}}{h^{n-1}(n-1)!}\,d\tau$$
$$= \frac{1}{h}\int_{t_k}^{t_k+h} e^{\frac{(h-\tau+t_k)z}{h}}\frac{(\tau-t_k)^{n-1}}{h^{n-1}(n-1)!}\,d\tau,$$
$$\phi_n(z) = \frac{1}{h^l}\int_{t_k}^{t_{k+1}} e^{\frac{1}{h}(t_{k+1}-\tau)z}\frac{(\tau-t_k)^{n-1}}{(n-1)!}\,d\tau.$$

Information from [1].

Exponential methods

```
from basecode import *
```

In this chapter, exponential methods are introduced, with further analysis of some of them, being tested and compared to more classical equivalents.

All the codes that created the convergence and deduction tables are in the appendix.

The stiff problem used in all the convergence tables is the following one taken from [1]:

$$u'(t) + 100u(t) = \sin(t)$$
$$u(0) = 1.$$

Solution:

$$u(t) = \exp(-100t) + \frac{\exp(-100t) + 100\sin(t) - \cos(t)}{1 + 100^2}.$$

Exponential Euler method

Expression:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + g(y(t_k), t_k)\frac{1 - e^{-h\lambda}}{\lambda} + O(h^2).$$

Table of convergence:

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0,2h)}$
128	0.0078125	4.398075514689716e-05	-
256	0.00390625	2.074422525626487e-05	1.0841625981445133
512	0.001953125	1.0056221183126109e-05	1.0446214904461004
1024	0.0009765625		

The table proved the order of conergence given by the deduction, and, comparing to the one of the classic Euler method:

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0,2h)}$
128	0.0078125	0.2391072699739873	-
256	0.00390625	0.08650412059872986	1.466817233501749
512	0.001953125	0.039214210532948934	1.1413923006132296
1024	0.0009765625	0.018739566082401515	1.0652890085799935

the exponential one has much better approximations since the beginning, proving the efficiency of the exponential method.

Exponential time differencing methods (ETD)

Expression:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + h^2\phi_2(-\lambda h)\frac{dg}{dt}(y(t_k),t_k) + h^3\phi_3(-\lambda h)\frac{d^2g}{dt^2}($$
$$h^n\phi_n(-\lambda h)\frac{d^{n-1}g}{dt^{n-1}}(y(t_k$$

It is possible to note that the exponential euler is essentially the exponential time differencing method of order 1.

In the same way as Taylor methods, the problem here is that at the expense of a higher order of convergence, ends up requiring the evaluation and implementation of multiple derivatives that may not even be easy to calculate. It can be avoided using Runge-Kutta methods, the next to be analyzed.

Exponential time differencing methods with Runge-Kutta time stepping

Here, the exponential Runge-Kutta were compared to methods from deductions following the classic runge-kutta approach in the constant variation formula.

All convergence tables prove the deduced order.

However, it is remarkable how much better the exponential methods are in relation to the new ones presented here, showing that we cannot be naive and apply the integral approximations expecting an exponential Runge-Kutta performance, the treatment must be exact for the linear part, as is done in all exponential methods.

Exponential - Trapezoidal rule

Expression:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + [g(a_k,t_{k+1}) - g(y(t_k),t_k)]h\phi_2(-\lambda h) + \\ + O(h^3)$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda}.$$

Convergence table:

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0,2h)}$
128	0.0078125	4.186569175362864e-08	-
256	0.00390625	1.0575183428604418e-08	1.985085775819591
512	0.001953125	2.652380943352073e-09	1.9953227875115886
1024	0.0009765625	6.638462730912398e-10	1.9983668943519293

Naive deduction - Trapezoidal rule

Expression:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + \frac{h}{2}\left[e^{-\lambda h}g(y(t_k), t_k) + g(a_k, t_{k+1})\right] + O(h^3)$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k), t_k)h\phi_1(-\lambda h).$$

Convergence table:

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0,2h)}$
128	0.0078125	0.0004242643044311458	-
256	0.00390625	0.00010714498082271644	1.9853990333325726
512	0.001953125	2.6871031228085582e-05	1.9954406751889993
1024	0.0009765625	6.725136514989377e-06	1.9984162299862431

Exponential - Third order

Expression:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + g\left(c'_k, t_{k+\frac{1}{2}}\right)h\phi_1($$
$$[g(c_k, t_{k+1}) - g(y(t_k), t_k)]\left(h\phi_2(-h\lambda) - \frac{h\phi_1(-h}{2}\right.$$
$$+4\left[g(c_k, t_{k+1}) + g(y(t_k), t_k) - 2g\left(c'_k, t_{k+\frac{1}{2}}\right)\right]\left(h\phi_3(-h\lambda) + \frac{h\phi_1(-h\lambda)}{8} - \frac{h\phi_2(-h\lambda)}{2}\right) +$$

with

$$c_k = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k), t_k) +$$
$$[g(a_k, t_{k+1}) - g(y(t_k), t_k)]h\phi_2(-\lambda h),$$
$$a_k = e^{-h\lambda}y(t_k) + g(y(t_k), t_k)h\phi_1(-h\lambda),$$
$$c'_k = e^{-\frac{h\lambda}{2}}y(t_k) +$$
$$\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right)g(y(t_k), t_k) +$$
$$\left[g\left(a'_k, t_{k+\frac{1}{2}}\right) - g(y(t_k), t_k)\right]\frac{h}{2}\phi_2\left(-\frac{\lambda h}{2}\right),$$
$$a'_k = e^{-\frac{h\lambda}{2}}y(t_k) +$$
$$g(y(t_k), t_k)\frac{h}{2}\phi_1\left(-\frac{h\lambda}{2}\right).$$

Convergence table:

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0,2h)}$
128	0.0078125	5.0853024048669315e-12	-
256	0.00390625	3.212833644961055e-13	3.9844153810116354
512	0.001953125	2.0132983821752326e-14	3.996213373698299
1024	0.0009765625	1.2602766052971504e-15	3.997748687591092

Better than what expected, giving order 4.

Naive deduction - Trapezoidal rule

Expression:

$$
\begin{aligned}
y(t_{k+1}) = e^{-h\lambda}y(t_k) + \\
\frac{h}{6}\left[e^{-\lambda h}g(y(t_k),t_k) + 4e^{-\frac{\lambda h}{2}}g\left(b'_k, t_k + \frac{h}{2}\right) + g(b_k, t_{k+1})\right] + O(h^4), \\
\text{with } b'_k = e^{-\frac{h\lambda}{2}}y(t_k) + \\
\frac{h}{4}\left[e^{-\frac{h\lambda}{2}}g(y(t_k),t_k) + g\left(a'_k, t_k + \frac{h}{2}\right)\right], \\
b_k = e^{-h\lambda}y(t_k) + \\
\frac{h}{2}\left[e^{-\lambda h}g(y(t_k),t_k) + g(a_k, t_{k+1})\right], \\
a'_k = e^{-\frac{h\lambda}{2}}y(t_k) + \\
g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\lambda\frac{h}{2}\right), \\
a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)h\phi_1(-\lambda h).
\end{aligned}
$$

Convergence table:

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0,2h)}$
128	0.0078125	1.083876968009309e-06	-
256	0.00390625	6.883813637344194e-08	3.9768491535433466
512	0.001953125	4.322307012305515e-09	3.9933345852265947
1024	0.0009765625	2.705360744453822e-10	3.9979086629155343

Also with order 4, better than what expected.

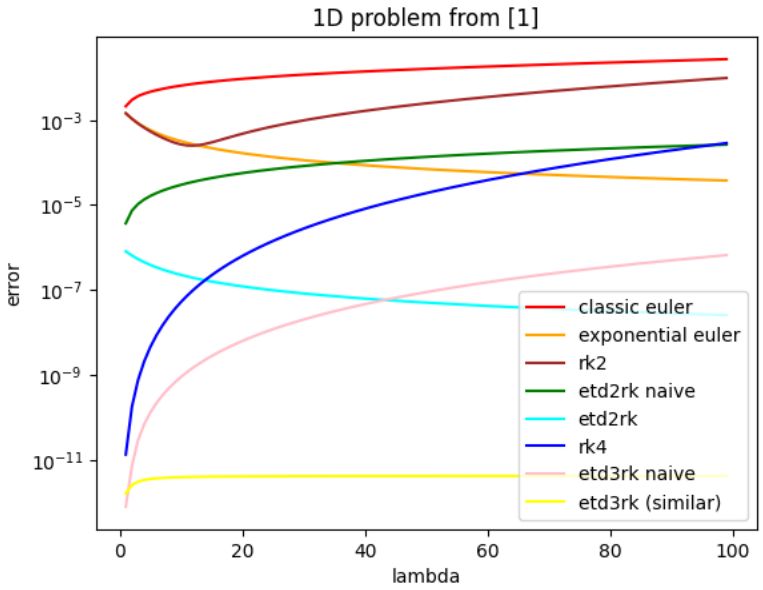
Graphics

Next, it is shown graphics from the same problem, but first showing the error as the linear part, 100, changes from 0 to 100 and next with $\lambda = 100$ but changng the time step.

```
n = 128
lmba0 = 1
lmbaf = 100
t0 = 0.0
tf = 1.0
x0 = np.array([1])
lmba_1D_classic, domain = errors_for_lambdas_array(n, classic_euler, t0, tf, x0,
lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_exponential, domain = errors_for_lambdas_array(n, exponential_euler, t0,
tf, x0, lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_etd2rk, domain = errors_for_lambdas_array(n, etd2rk, t0, tf, x0, lmba0,
lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_etd2rk_trapezoidal_naive, domain = errors_for_lambdas_array(n,
etd2rk_trapezoidal_naive, t0, tf, x0, lmba0, lmbaf, A_1D, g, sol_given_lmba,
vectorize_sol_given_lmba, error_2)
lmba_1D_etd3rk_similar, domain = errors_for_lambdas_array(n, etd3rk_similar, t0,
tf, x0, lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_etd3rk_naive, domain = errors_for_lambdas_array(n, etd3rk_naive, t0, tf,
x0, lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_rk2, domain = errors_for_lambdas_array(n, rk2, t0, tf, x0, lmba0, lmbaf,
A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_rk4, domain = errors_for_lambdas_array(n, rk4, t0, tf, x0, lmba0, lmbaf,
A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
```

```
/home/miki/IC/Relatorio_github/basecode.py:247: ComplexWarning: Casting complex
values to real discards the imaginary part
  return np.sqrt(float(np.sum(v)/x_approx.size)) #normalized
```

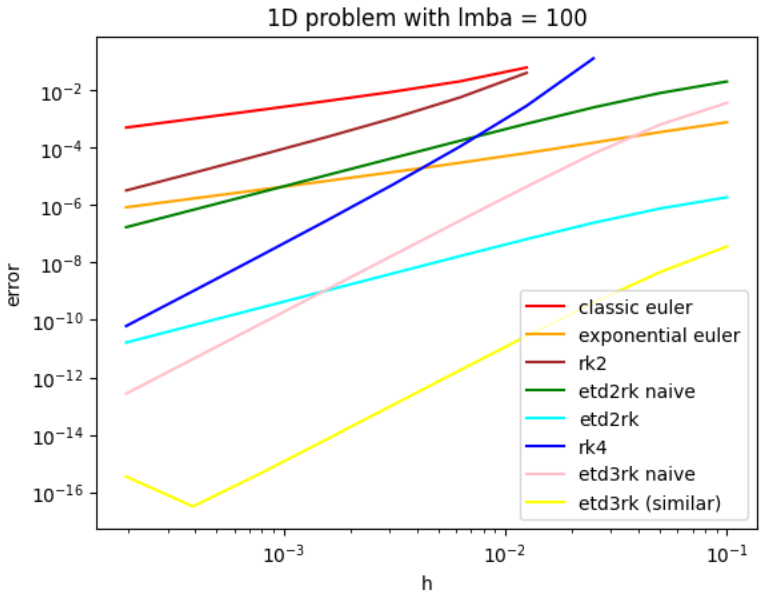
```
matrix_1D = [lmba_1D_classic, lmba_1D_exponential, lmba_1D_rk2,
lmba_1D_etd2rk_trapezoidal_naive, lmba_1D_etd2rk, lmba_1D_rk4,
lmba_1D_etd3rk_naive, lmba_1D_etd3rk_similar]
names = ['classic euler', 'exponential euler', 'rk2', 'etd2rk naive', 'etd2rk',
'rk4', 'etd3rk naive', "etd3rk (similar)"]
fig, ax = graphic_2D(8*[domain], matrix_1D, names, "lambda", "error", "1D problem
from [1]", False, True)
```



Here it is notable that as the lambda increases, and so does the stiffness, the exponential methods deal really well, even dropping the error, since the exponential part is precisely solved, so, as it gains more relevance, the method perfoms better. Meanwhile, the other methods (classic and naive) start to decline, dealing badly with the stiffness. Just as predicted.

```
n0 = 10
k = 10
lmba = 100
A = lmba * np.array([[1]])
t0 = 0.0
tf = 1.0
x0 = np.array([1])
n_1D_classic, domain = errors_2x(n0, k, classic_euler, t0, tf, x0, A, g, sol,
vectorize_sol, error_2)
n_1D_exponential, domain = errors_2x(n0, k, exponential_euler, t0, tf, x0, A, g,
sol, vectorize_sol, error_2)
n_1D_etd2rk, domain = errors_2x(n0, k, etd2rk, t0, tf, x0, A, g, sol,
vectorize_sol, error_2)
n_1D_etd2rk_trapezoidal_naive, domain = errors_2x(n0, k, etd2rk_trapezoidal_naive,
t0, tf, x0, A, g, sol, vectorize_sol, error_2)
n_1D_etd3rk_similar, domain = errors_2x(n0, k, etd3rk_similar, t0, tf, x0, A, g,
sol, vectorize_sol, error_2)
n_1D_etd3rk_naive, domain = errors_2x(n0, k, etd3rk_naive, t0, tf, x0, A, g, sol,
vectorize_sol, error_2)
n_1D_rk2, domain = errors_2x(n0, k, rk2, t0, tf, x0, A, g, sol, vectorize_sol,
error_2)
n_1D_rk4, domain = errors_2x(n0, k, rk4, t0, tf, x0, A, g, sol, vectorize_sol,
error_2)
```

```
matrix_2D = [n_1D_classic, n_1D_exponential, n_1D_rk2,
n_1D_etd2rk_trapezoidal_naive, n_1D_etd2rk, n_1D_rk4, n_1D_etd3rk_naive,
n_1D_etd3rk_similar]
names = ['classic euler', 'exponential euler', 'rk2', 'etd2rk naive', 'etd2rk',
'rk4', 'etd3rk naive', "etd3rk (similar)"]
fig_2D, ax_2D = graphic_2D(8*[1/domain], matrix_2D, names, "h", "error", "1D
problem with lmba = "+str(lmba), False, True)
plt.xscale('log')
```



Here is visually clear the orders already confirmed by the convergence tables.

Swing spring application

Finally, an application is done for the swing spring problem.

```
from basecode import *
```

From [9], the model is, given m, k, l and g,

$$\begin{aligned}\dot{\theta} &= \frac{\rho_\theta}{mr^2}\\ \dot{\rho}_\theta &= -mgr\sin\theta\\ \dot{r} &= \frac{\rho_r}{m}\\ \dot{\rho}_r &= \frac{\rho_\theta^2}{mr^3} - k(r-l) + mg\cos\theta.\end{aligned}$$

In matrix form,

$$\frac{d}{dt}\begin{bmatrix}\theta\\ \rho_\theta\\ r\\ \rho_r\end{bmatrix} = \begin{bmatrix}0 & \frac{1}{ml^2} & 0 & 0\\ -mgl & 0 & 0 & 0\\ 0 & 0 & 0 & \frac{1}{m}\\ 0 & 0 & -k & 0\end{bmatrix}\begin{bmatrix}\theta\\ \rho_\theta\\ r\\ \rho_r\end{bmatrix} + \begin{bmatrix}\frac{\rho_\theta}{m}\left(\frac{1}{r^2} - \frac{1}{l^2}\right)\\ -mg(r\sin\theta - l\theta)\\ 0\\ \frac{\rho_\theta^2}{mr^3} + kl + mg\cos\theta\end{bmatrix}.$$

For $m = 1, l = 1, g = \pi^2, k = 100\pi^2$, using the etd3rk method deduced, the best one tested,

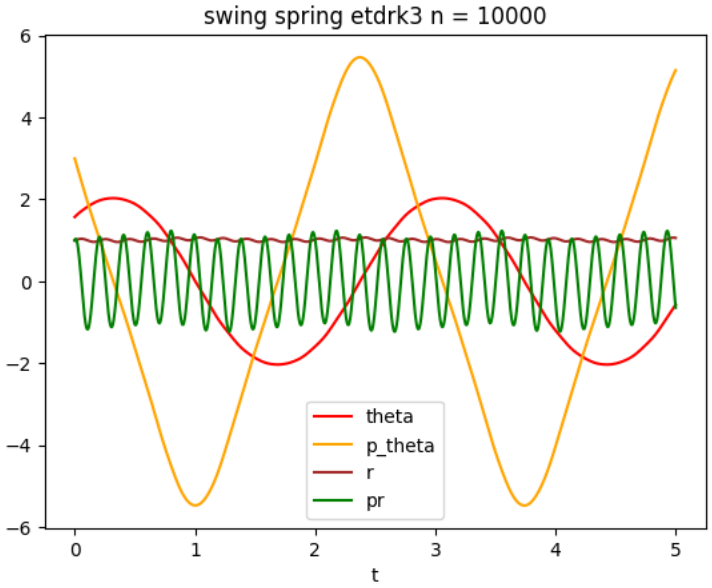
```
def g_swing_spring(x, t):
  m = 1
  l = 1
  gr = np.pi**2
  k = 100 * np.pi**2
  vector = np.zeros(x.size)
  theta = x[0]
  p_theta = x[1]
  r = x[2]
  pr = x[3]
  vector[0] = p_theta/m * (1/r**2 - 1/l**2)
  vector[1] = -m*gr*(r*np.sin(theta)-l*theta)
  vector[3] = p_theta**2/(m*r**3) + k*l + m*gr*np.cos(theta)
  return vector
```

```
m = 1
l = 1
g = np.pi**2
k = 100 * np.pi**2
A = np.matrix([[0,-1/(m*l**2),0,0], [m*g*l,0,0,0], [0,0,0,-1/m], [0,0,k,0]])
n = 10000
t0 = 0.0
tf = 5.0
x0 = np.array([np.pi/2,3,l,1])
domains = 4*[np.arange(t0, tf, (tf-t0)/n)]
x = etd3rk_similar(t0, tf, n, x0, A, g_swing_spring)
names = ['theta','p_theta', 'r', 'pr']
matrix1 = [x[0,:], x[1,:], x[2,:], x[3,:]]

fig1, ax1 = graphic_2D(domains, matrix1, names, 't', ' ', 'swing spring etdrk3 n =
'+str(n), False, False)
```

```
/tmp/ipykernel_12326/2307441015.py:11: ComplexWarning: Casting complex values to
real discards the imaginary part
  vector[0] = p_theta/m * (1/r**2 - 1/l**2)
/tmp/ipykernel_12326/2307441015.py:12: ComplexWarning: Casting complex values to
real discards the imaginary part
  vector[1] = -m*gr*(r*np.sin(theta)-l*theta)
/tmp/ipykernel_12326/2307441015.py:13: ComplexWarning: Casting complex values to
real discards the imaginary part
  vector[3] = p_theta**2/(m*r**3) + k*l + m*gr*np.cos(theta)
```

```
/home/miki/.local/lib/python3.10/site-packages/matplotlib/cbook/__init__.py:1369:
ComplexWarning: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



References

1. HOCHBRUCK, M.; OSTERMANN, A. Exponential integrators. Acta Numer, Cambridge Univ Press, v. 19, p. 209–286, 2010.
2. BURDEN, Richard L.; FAIRES, J. Douglas. Numerical Analysis. 9.ed. Boston:Brooks/Cole, 2010. p.348-353.
3. ROMA, Alexandre. Lecture notes. Introdução à Resolução Numérica do Problema de Cauchy (Introduction to numerical resolution of Cauchy problem), MAP5002. Jan. and Feb. 2023. IME-USP University of São Paulo.
4. TAL, Fábio A. Lecture notes. Técnicas em Teoria do Controle (techniques in control theory), MAP2321. Aug. to Dez. 2022. IME-USP University from São Paulo.
5. Apostol, T.M. Calculus v. 1. Blaisdell book in pure and applied mathematics. https://books.google.com.br/books?id=sR_vAAAAMAAJ. 1961. Blaisdell Publishing Company.

6. S.M. Cox, P.C. Matthews, Exponential Time Differencing for Stiff Systems, Journal of Computational Physics, Volume 176, Issue 2, 2002, Pages 430-455, ISSN 0021-9991, https://doi.org/10.1006/jcph.2002.6995.
7. Hochbruck, Marlis, and Alexander Ostermann. “Explicit Exponential Runge-Kutta Methods for Semilinear Parabolic Problems.” SIAM Journal on Numerical Analysis, vol. 43, no. 3, 2006, pp. 1069–90. JSTOR, http://www.jstor.org/stable/4101280. Accessed 27 June 2023.
8. DOBRUSHKIN, Vladimir. “MATHEMATICA TUTORIAL for the Second Course. Part III: Spring Pendulum”. Monday, September 4, 2023 10:24:40 PM. https://www.cfm.brown.edu/people/dobrush/am34/Mathematica/ch3/spendulum.html
9. Lynch, Peter. (2000). The Swinging Spring: A Simple Model of Atmospheric Balance. https://maths.ucd.ie/~plynch/Publications/AOD_Paper.pdf

Participation in scientific event, list of publications and list of papers prepared or submitted

Nothing to declare.

Appendix

Code

All the functions coded are in the following environment.

```
from math import *
import numpy as np
from collections import deque
import matplotlib.pyplot as plt
from scipy.linalg import expm
from scipy import linalg

stab_lim = 1000.0

def classic_euler(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    for i in range(1, n):
        x[:,i] = x[:,i-1] + h*(np.matmul(-A,x[:,i-1]) + g(x[:,i-1],t))
        t = t0 + i*h
        if np.any(x[:,i].real > stab_lim):
            x[:,i] = np.nan
    return x

def exponential_euler(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0] = x0
    t = t0
    exponential_matrix = expm(-h*A)
    hphi1 = calculate_hphi1(h, A)
    for i in range(1, n):
        x[:,i] = np.matmul(exponential_matrix, x[:,i-1]) +
np.matmul(hphi1,g(x[:,i-1],t))
        t = t0 + i*h
    return x

def calculate_hphi1(h, A):
  '''(float, np.matrix) -> np.matrix'''
  dim = A.shape[0]
  hphi1 = np.matmul(np.eye(dim)-expm(-h*A), linalg.inv(A))
  return hphi1

def calculate_hphi2(h, A, hphi1):
    #IT IS NOT H2PHI2
    '''(float, np.matrix, np.matrix) -> np.matrix'''
    dim = A.shape[0]
    hphi2 = np.matmul(np.eye(dim)-hphi1/h, linalg.inv(A))
    return hphi2

def calculate_hphi3(h, A, hphi2):
    '''(float, np.matrix, np.matrix) -> np.matrix'''
    dim = A.shape[0]
    hphi3 = np.matmul(1/2*np.eye(dim)-hphi2/h, linalg.inv(A))
    return hphi3

def etd2(t0, tf, n, x0, A, g, derivate_of_g):
    '''(float, float, int, np.array, np.matrix, function, function) ->
np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0] = x0
    t = t0
    exponential_matrix = expm(-h*A)
    hphi1 = calculate_hphi1(h, A)
    hphi2 = calculate_hphi2(h, A, hphi1)
    for i in range(1, n):
        x[:,i] = np.matmul(exponential_matrix, x[:,i-1]) +
np.matmul(hphi1,g(x[:,i-1],t)) + h*np.matmul(hphi2,derivate_of_g(x[:,i-1],t))
        t = t0 + i*h
    return x

def rk2(t0, tf, n, x0, A, g): #heun s method
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    for i in range(1, n):
        a = x[:,i-1] + h*(np.matmul(-A,x[:,i-1]) + g(x[:,i-1],t))
        f1 = np.matmul(-A,x[:,i-1]) + g(x[:,i-1],t)
        f2 = np.matmul(-A,a) + g(a,t)
        x[:,i] = x[:,i-1] + .5 * h * (f1 + f2)
        t = t0 + i*h
        if np.any(x[:,i].real > stab_lim):
            x[:,i] = np.nan
    return x
```

```
def etd2rk(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    exponential_matrix = expm(-h*A)
    hphi1 = calculate_hphi1(h, A)
    hphi2 = calculate_hphi2(h, A, hphi1)
    for i in range(1, n):
        a = np.matmul(exponential_matrix, x[:,i-1]) + np.matmul(hphi1,g(x[:,i-
1],t))
        x[:,i] = np.matmul(exponential_matrix, x[:,i-1]) +
np.matmul(hphi1,g(x[:,i-1],t)) + np.matmul(hphi2,g(a, t0 + i*h)-g(x[:,i-1],t))
        t = t0 + i*h
    return x

def etd2rk_midpoint_rule(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    exponential_matrix = expm(-h*A)
    exponential_matrix_2 = expm(-h/2*A)
    h_2phi1_2 = calculate_hphi1(h/2, A)
    hphi1 = calculate_hphi1(h, A)
    hphi2 = calculate_hphi2(h, A, hphi1)
    for i in range(1, n):
        b = np.matmul(exponential_matrix_2, x[:,i-1]) +
np.matmul(h_2phi1_2,g(x[:,i-1],t))
        x[:,i] = np.matmul(exponential_matrix, x[:,i-1]) +
np.matmul(hphi1,g(x[:,i-1],t)) + 2*np.matmul(hphi2,g(b, t + h/2)-g(x[:,i-1],t))
        t = t0 + i*h
    return x

def etd2rk_trapezoidal_naive(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    exponential_matrix = expm(-h*A)
    hphi1 = calculate_hphi1(h, A)
    for i in range(1, n):
        a = np.matmul(exponential_matrix, x[:,i-1]) + np.matmul(hphi1,g(x[:,i-
1],t))
        x[:,i] = np.matmul(exponential_matrix, x[:,i-1]) + .5 * h *
(np.matmul(exponential_matrix, g(x[:,i-1],t)) + g(a, t0 + i*h))
        t = t0 + i*h
    return x

def etd2rk_midpoint_rule_naive(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function, np.matrix) ->
np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    exponential_matrix = expm(-h*A)
    exponential_matrix_2 = expm(-h/2*A)
    h_2phi1_2 = calculate_hphi1(h/2, A)
    for i in range(1, n):
        b = np.matmul(exponential_matrix_2, x[:,i-1]) +
np.matmul(h_2phi1_2,g(x[:,i-1],t))
        x[:,i] = np.matmul(exponential_matrix, x[:,i-1]) + h *
np.matmul(exponential_matrix_2, g(b, t+h/2))
        t = t0 + i*h
    return x

def rk4(t0, tf, n, x0, A, g):
    '''(float, float, int, np.array, np.matrix, function) -> np.matrix'''
    h = (tf-t0)/n
    x = np.zeros((x0.size,n), dtype=np.complex_)
    x[:,0]=x0
    t = t0
    for i in range(1, n):
        k1 = np.matmul(-A,x[:,i-1]) + g(x[:,i-1],t)
        x2 = x[:,i-1] + h * k1 / 2
        k2 = np.matmul(-A,x2) + g(x2,t+h/2)
        x3 = x[:,i-1] + h * k2 / 2
        k3 = np.matmul(-A,x3) + g(x3,t+h/2)
        x4 = x[:,i-1] + h * k3
        k4 = np.matmul(-A,x4) + g(x4,t0 + i*h)
        x[:,i] = x[:,i-1] + h / 6 * (k1 + 2*k2 + 2*k3 + k4)
        t = t0 + i*h
        if np.any(x[:,i].real > stab_lim):
```

```
            x[:,i] = np.nan
    return x

def etd3rk_similar(t0, tf, n, x0, A, g):
  '''(float, float, int, np.array, np.matrix, function, np.matrix) -> np.matrix'''
  h = (tf-t0)/n
  x = np.zeros((x0.size,n), dtype=np.complex_)
  x[:,0]=x0
  t = t0
  exponential_matrix = expm(-h*A)
  exponential_matrix_2 = expm(-h/2*A)
  hphi1 = calculate_hphi1(h, A)
  h_2phi1_2 = calculate_hphi1(h/2, A)
  hphi2 = calculate_hphi2(h, A, hphi1)
  h_2phi2_2 = calculate_hphi2(h/2, A, h_2phi1_2)
  hphi3 = calculate_hphi3(h, A, hphi2)
  for i in range(1, n):
    fst_term = np.matmul(exponential_matrix, x[:,i-1])
    fst_term_2 = np.matmul(exponential_matrix_2, x[:,i-1])
    a = fst_term + np.matmul(hphi1,g(x[:,i-1],t))
    a_ = fst_term_2 + np.matmul(h_2phi1_2,g(x[:,i-1],t))
    c = fst_term + np.matmul(hphi1,g(x[:,i-1],t)) + np.matmul(hphi2,g(a, t0 +
i*h)-g(x[:,i-1],t))
    c_ = fst_term_2 + np.matmul(h_2phi1_2,g(x[:,i-1],t)) +
np.matmul(h_2phi2_2,g(a_, t0 + i*h)-g(x[:,i-1],t))
    snd_term = np.matmul(hphi1, g(c_, t+h/2))
    trd_term = np.matmul(hphi2 - hphi1/2,g(c, t0 + i*h)-g(x[:,i-1],t))
    fth_term = 4 * np.matmul(hphi3+hphi1/8-hphi2/2, g(c, t0 + i*h)+g(x[:,i-
1],t)-2*g(c_, t + h/2))
    x[:,i] = fst_term + snd_term + trd_term + fth_term
    t = t0 + i*h
  return x

def etd3rk(t0, tf, n, x0, A, g):
  '''(float, float, int, np.array, np.matrix, function, np.matrix) -> np.matrix'''
  h = (tf-t0)/n
  x = np.zeros((x0.size,n), dtype=np.complex_)
  x[:,0]=x0
  t = t0
  exponential_matrix = expm(-h*A)
  exponential_matrix_2 = expm(-h/2*A)
  hphi1 = calculate_hphi1(h, A)
  h_2phi1_2 = calculate_hphi1(h/2, A)
  hphi2 = calculate_hphi2(h, A, hphi1)
  h_2phi2_2 = calculate_hphi2(h/2, A, h_2phi1_2)
  hphi3 = calculate_hphi3(h, A, hphi2)
  for i in range(1, n):
    fst_term = np.matmul(exponential_matrix, x[:,i-1])
    fst_term_2 = np.matmul(exponential_matrix_2, x[:,i-1])
    a = fst_term_2 + np.matmul(h_2phi1_2,g(x[:,i-1],t))
    b = fst_term + np.matmul(hphi1,2*g(a,t+h/2)-g(x[:,i-1],t))
    snd_term = np.matmul(hphi1, g(a, t+h/2))
    trd_term = np.matmul(hphi2 - hphi1/2,g(b, t0 + i*h)-g(x[:,i-1],t))
    fth_term = 4 * np.matmul(hphi3+hphi1/8-hphi2/2, g(b, t0 + i*h)+g(x[:,i-
1],t)-2*g(a, t + h/2))
    x[:,i] = fst_term + snd_term + trd_term + fth_term
    t = t0 + i*h
  return x

def etd3rk_naive(t0, tf, n, x0, A, g):
  '''(float, float, int, np.array, np.matrix, function, np.matrix) -> np.matrix'''
  h = (tf-t0)/n
  x = np.zeros((x0.size,n), dtype=np.complex_)
  x[:,0]=x0
  t = t0
  exponential_matrix = expm(-h*A)
  exponential_matrix_2 = expm(-h/2*A)
  hphi1 = calculate_hphi1(h, A)
  h_2phi1_2 = calculate_hphi1(h/2, A)
  for i in range(1, n):
    fst_term = np.matmul(exponential_matrix, x[:,i-1])
    fst_term_2 = np.matmul(exponential_matrix_2, x[:,i-1])
    a = fst_term + np.matmul(hphi1,g(x[:,i-1],t))
    a_ = fst_term_2 + np.matmul(h_2phi1_2,g(x[:,i-1],t))
    c = fst_term + .5 * h * (np.matmul(exponential_matrix, g(x[:,i-1],t)) + g(a,
t0 + i*h))
    c_ = fst_term_2 + .25 * h * (np.matmul(exponential_matrix_2, g(x[:,i-1],t)) +
g(a_, t0 + i*h))
    snd_term = np.matmul(exponential_matrix, g(x[:,i-1],t))
    trd_term = 4*np.matmul(exponential_matrix_2, g(c_,t+h/2))
    fth_term = g(c, t0 + i*h)
    x[:,i] = fst_term + h*(snd_term + trd_term + fth_term)/6
    t = t0 + i*h
  return x

def error_2(x_approx, x_exact):
    ''' (np.vector, np.vector) -> float '''
```

```
    #make sure that x_approx and x_exact have the same lenght
    v = (x_approx - x_exact)*(x_approx - x_exact).conjugate()
    #^certainly pure real
    return np.sqrt(float(np.sum(v)/x_approx.size)) #normalized

def error_sup(x_approx, x_exact):
    ''' (np.vector, np.vector) -> float '''
    #make sure that x_approx and x_exact have the same lenght
    v = abs(x_approx - x_exact)
    return np.amax(v)

def g(x, t):
    ''' (np.array, float) -> float
        (x, t) -> g(x, t)
    '''
    g = np.array([np.sin(t)])
    return g

def g_linear( x, t ):
    ''' (np.array, float) -> np.array
        (x, t) -> g(x, t)
    '''
    g = np.zeros(x.size)
    return g

def sol( t ):
    ''' (float, float) -> float
    RECEIVES the initial value and a real (t).
    APPLIES the cauchy problem solution to this initial value at this point.
    RETURNS a real value.
    '''
    lmba = 100
    sol = np.exp(-lmba*t)+(np.exp(-lmba*t)+lmba*np.sin(t)-np.cos(t))/(1+lmba*lmba)
    return sol

def sol_given_lmba(lmba, t ):
    ''' (float, float) -> float
    RECEIVES the initial value and a real (t).
    APPLIES the cauchy problem solution to this initial value at this point.
    RETURNS a real value.
    '''
    sol = np.exp(-lmba*t)+(np.exp(-lmba*t)+lmba*np.sin(t)-np.cos(t))/(1+lmba*lmba)
    return sol

def vectorize_sol_given_lmba(lmba, t0, t1, n, sol):
    '''
    (float, float, float, int, function) -> np.vector
    n is the number of steps
    '''
    x = np.zeros((sol(lmba,t0).size,n), dtype=np.complex_)
    h = (t1-t0)/n
    for i in range(n):
        x[:,i] = sol(lmba, t0+i*h)
    return x

def vectorize_sol(t0, t1, n, sol):
    '''
    (float, float, int, function) -> np.vector
    n is the number of steps
    '''
    x = np.zeros((sol(t0).size,n), dtype=np.complex_)
    h = (t1-t0)/n
    for i in range(n):
        x[:,i] = sol(t0+i*h)
    return x

def A_1D(lmba):
  '''(int) -> np.matrix'''
  return np.array([[lmba]])

def A_2D(lmba):
  '''(int) -> np.matrix'''
  return np.array([[0, -lmba],[lmba, 0]])

def errors_for_lambdas_array(n, method, t0, tf, x0, lmba0, lmbaf, Af, g,
sol_given_lmba, vectorize_sol_given_lmba, error):
    '''
    This function is a variation of the errors_array function. Here, the linear
    part of the problem is varying instead of the number of steps, which is now
    fixed.
    This function will RETURN 2 arrays.
    The first one has the errors of the approximations given by the method with
    coefficient of the linear part of the ploblem
    A = Af(lmba0), Af(lmba0+1), Af(lmba0+2), ..., Af(lmbaf-1).
    The second is [lmba0, lmba0+1, lmba0+2, ..., lmbaf-1]

    RECEIVES:
```

```
    n is the number of steps. (int)
    method have arguments (t0, tf, n, x0, lmba, g) and return a
    np.vector of length n (0, 1, 2, ..., n-1), n is the number of steps.
(function)
    t0 is the initial point of the approximation. (float)
    tf is the last one. (float)
    x0 is the initial value of the Cauchy problem. (np.array)
    lmba0 and lmbaf are integers as described before. (int)
    Af is a function that receives the stiffness parameter and returns the
    corresponding linear coefficient. (function)
    g is a function (float, float) -> (float). (function)
    sol is a function (float) -> (float). (function)
    vectorize_sol is a function that "transforms sol in a vector" (function)
    (float, float, int, function) -> (np.array)
    (t0, tf, n, sol) -> np.array([sol[t0], sol[t0+h], sol[t0+2h], ..., sol[tf-1]])
    error is a function (np.array, np.array) -> (float) (function)
    '''
    v = np.zeros(lmbaf-lmba0)
    domain = np.arange(lmba0, lmbaf)
    for i in range(lmbaf-lmba0):
        lmba = lmba0 + i
        m = method(t0, tf, n, x0, Af(lmba), g)
        exact = vectorize_sol_given_lmba(lmba, t0, tf, n, sol_given_lmba)
        if np.max(np.abs(m))>1000:
            v[lmba-lmba0]=np.nan
        else:
            v[lmba-lmba0] = error(m, exact)
    return v, domain

def errors_array(n0, nf, method, t0, tf, x0, A, g, sol, vectorize_sol, error):
  '''
  This function will RETURN 2 arrays.
  The first one has the errors of the approximations given by the method with
  number of steps n = n0, n0+1, n0+2, ..., nf-1.
  The second is [n0, n0+1, n0+2, ..., nf-1]

  RECEIVES:
  n0 is the first number of steps. (int)
  nf is the last one plus 1. (int)
  method have arguments (t0, tf, n, x0, A, lmba, g) and return a
  np.vector of length n (0, 1, 2, ..., n-1), n is the number of steps. (function)
  t0 is the initial point of the approximation. (float)
  tf is the last one. (float)
  x0 is the initial value of the Cauchy problem. (float)
  A is the coefficient os the linear part of the ploblem. (float)
  g is a function (int, float, float) -> (float). (function)
  sol is a function (int, float) -> (float). (function)
  vectorize_sol is a function that "transforms sol in a vector" (function)
  (float, float, int, function) -> (np.array)
  (t0, tf, n, sol) -> np.array([sol[t0], sol[t0+h], sol[t0+2h], ..., sol[tf-1]])
  error is a function (np.array, np.array) -> (float) (function)
  '''
  v = np.zeros(nf-n0)
  domain = np.arange(n0, nf)
  for n in range(n0, nf):
    m = method(t0, tf, n, x0, A, g)
    exact = vectorize_sol(t0, tf, n, sol)
    if np.max(np.abs(m))>1000:
        v[n-n0]=np.nan
    else:
        v[n-n0] = error(m, exact)
  return v, domain

def graphic_2D(domain, matrix, names, labelx, labely, title, key1, key2):
  '''
  domain is a list of np.arrays [[length n1], [legth n2], ..., [length nk]]
  k = 1, 2, ..., 5 lines. (list)
  matrix is a list of np.arrays [[length n1], [legth n2], ..., [length nk]]
  k = 1, 2, ..., 5 lines - same length that domain. (list)
  names is a list of the labels for the graphs, must have the same length that
  the number of lines in matrix. (list of Strings)
  labelx is the name of the x coordinate. (String)
  labely is the name of the y coordinate. (String)
  title is the title of the graph. (String)
  key1 is a boolean that indicates if the last graph must be black. (bool)
  key2 is a boolean that indicates if it should use the log scale. (bool)
  '''
  fig, ax = plt.subplots()

  colors = ['red', 'orange', 'brown', 'green', 'cyan', 'blue', 'pink', 'yellow',
'gold', 'maroon']
  for i in range(len(names)-1):
    ax.plot(domain[i], matrix[i], color=colors[i], label=names[i])
  if key1:
    ax.plot(domain[len(names)-1], matrix[len(names)-1], color='black',
label=names[len(names)-1])
  else:
```

```
    ax.plot(domain[len(names)-1], matrix[len(names)-1],
color=colors[len(names)-1], label=names[len(names)-1])
  if key2:
    plt.yscale('log')
  ax.legend()
  ax.set_xlabel(labelx)
  ax.set_ylabel(labely)
  ax.set_title(title)
  return fig, ax

def graphic_3D(domain, matrix1, matrix2, names, labelx, labely, labelz, title,
key1, key2):
  '''
  domain is a list of np.arrays [[length n1], [legth n2], ..., [length nk]]
  k = 1, 2, ..., 5 lines. (list)
  matrix1 and matrix2 are lists of np.arrays [[length n1], [legth n2], ...,
[length nk]]
  k = 1, 2, ..., 5 lines - same length that domain. (list)
  names is a list of the labels for the graphs, must have the same length that
  the number of lines in matrix. (list of Strings)
  labelx is the name of the x coordinate. (String)
  labely is the name of the y coordinate. (String)
  labelz is the name of the z coordinate. (String)
  title is the title of the graph. (String)
  key1 is a boolean that indicates if the last graph must be black. (bool)
  key2 is a boolean that indicates if it should use the log scale. (bool)
  '''
  fig = plt.figure()
  ax = plt.figure().add_subplot(projection='3d')

  colors = ['blue', 'green', 'red', 'cyan', 'magenta', 'yellow']
  for i in range(len(names)-1):
    ax.plot(domain[i], matrix1[i], matrix2[i], color=colors[i], label=names[i])
  if key1:
    ax.plot(domain[len(names)-1], matrix1[len(names)-1], matrix2[len(names)-1],
color='black', label=names[len(names)-1])
  else:
    ax.plot(domain[len(names)-1], matrix1[len(names)-1], matrix2[len(names)-1],
color=colors[len(names)-1], label=names[len(names)-1])
  if key2:
    plt.yscale('log')
  ax.legend()
  ax.set_xlabel(labelx)
  ax.set_ylabel(labely)
  ax.set_zlabel(labelz)
  ax.set_title(title)
  return fig, ax

def errors_2x(n0, k, method, t0, tf, x0, A, g, sol, vectorize_sol, error):
  '''
  This function will RETURN a np.array with the errors of the approximations given
  by the method with number of steps n = n0, 2*n0, 2**2*n0, ..., 2**(k-1)*n0.

  RECEIVES:
  n0 is the first number of steps. (int)
  k is the number of errors in the final array. (int)
  method have arguments (t0, tf, n, x0, lmba, g) and return a
  np.vector of length n (0, 1, 2, ..., n-1), n is the number of steps. (function)
  t0 is the initial point of the approximation. (float)
  tf is the last one. (float)
  x0 is the initial value of the Cauchy problem. (float)
  A is the coefficient of the linear part of the ploblem. (np.matrix)
  g is a function (float, float) -> (float). (function)
  sol is a function (float) -> (float). (function)
  vectorize_sol is a function that "transforms sol in a vector" (function)
  (float, float, int, function) -> (np.array)
  (t0, tf, n, sol) -> np.array([sol[t0], sol[t0+h], sol[t0+2h], ..., sol[tf-1]])
  error is a function (np.array, np.array) -> (float) (function)
  '''
  v = np.zeros(k)
  domain = np.zeros(k)
  for i in range(k):
    domain[i] = n0*2**i
    m = method(t0, tf, n0*2**i, x0, A, g)
    exact = vectorize_sol(t0, tf, n0*2**i, sol)
    v[i] = error(m, exact)
  return v, domain

def convergence_table(errors_2x, n0, k, t0, tf):
  '''
  RECEIVES:
  errors_2x is a array with the errors of the approximations given
  by a method with number of steps n = n0, 2*n0, 2**2*n0, ..., 2**(k-1)*n0.
(np.array)
  n0 is the first number of steps. (int)
  k is the number of errors in the final array. (int)
  t0 is the initial point of the approximation. (float)
```

```
  tf is the last one. (float)
  '''
  n = n0
  print(n, (tf-t0)/n, errors_2x[0], "-", sep=" & ", end=" \\\\ \n")
  for i in range(1, k):
      n = n0 * 2 ** i
      h = (tf-t0)/n
      q = errors_2x[i-1]/errors_2x[i] #q=erro(h)/erro(h)
      r = ((tf-t0)/(n/2))/((tf-t0)/n)
      print(n, h, errors_2x[i], log(q,2)/log(r,2), sep=" & ", end=" \\\\ \n")

def convergence_table(errors_2x, n0, k, t0, tf):
  '''
  RECEIVES:
  errors_2x is a array with the errors of the approximations given
  by a method with number of steps n = n0, 2*n0, 2**2*n0, ..., 2**(k-1)*n0.
(np.array)
  n0 is the first number of steps. (int)
  k is the number of errors in the final array. (int)
  t0 is the initial point of the approximation. (float)
  tf is the last one. (float)
  '''
  n = n0
  print("| n | h = $\\frac{1}{h}$ | $\\tau(0,h)$ | q = $\\frac{tau(0,h)}{tau(0,
2h)}$ |")
  print("|---|-----------------|-----------|---------------------------------|")
  print("", n, (tf-t0)/n, errors_2x[0], "-", sep=" | ", end=" | \n")
  for i in range(1, k):
      n = n0 * 2 ** i
      h = (tf-t0)/n
      q = errors_2x[i-1]/errors_2x[i] #q=erro(h)/erro(h)
      r = ((tf-t0)/(n/2))/((tf-t0)/n)
      print( "", n, h, errors_2x[i], log(q,2)/log(r,2), sep=" | ", end=" | \n")

def lmba_n_error(errors_for_lambdas_array, method, x0, Af, g, sol_given_lmba,
vectorize_sol_given_lmba, error, method_name):
  lmba0 = 5
  lmbaf = 100
  n0 = 10
  nf = 128
  t0 = 0.0
  tf = 1.0
  # Create data for X, Y
  lmba_values = np.arange(lmba0, lmbaf)
  n_values = np.arange(n0, nf)
  X, Y = np.meshgrid(lmba_values, 1/n_values)
  # Create a matrix of zeros for Z
  Z = np.zeros_like(X)
  # Populate the Z matrix with data using a function
  for n in range(n0, nf):
    Z[n-n0], domain = errors_for_lambdas_array(n, method, t0, tf, x0, lmba0,
lmbaf, Af, g, sol_given_lmba, vectorize_sol_given_lmba, error)
  # Create filled contour plot
  plt.contourf(X, Y, Z)
  # Add color bar for the contour plot
  plt.colorbar()
  # Add labels and title (optional)
  plt.xlabel('lambda')
  plt.ylabel('h')
  plt.title('errors for the '+method_name+' method ')
  # Show the plot
  plt.show()
```

Convergence tables

Classic Euler

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])
errors_2x_vector, domain = errors_2x(n0, k, classic_euler, t0, tf, x0, A, g, sol,
vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	0.2391072699739873	-
256	0.00390625	0.08650412059872986	1.466817233501749
512	0.001953125	0.039214210532948934	1.1413923006132296
1024	0.0009765625	0.018739566082401515	1.0652890085799935

Exponential Euler

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])
errors_2x_vector, domain = errors_2x(n0, k, exponential_euler, t0, tf, x0, A, g,
sol, vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	4.398075514689716e-05	-
256	0.00390625	2.074422525626487e-05	1.0841625981445133
512	0.001953125	1.0056221183126109e-05	1.0446214904461004
1024	0.0009765625	4.948885884282876e-06	1.0229126060177947

rk2

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])

errors_2x_vector, domain = errors_2x(n0, k, rk2, t0, tf, x0, A, g, sol,
vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	0.06606851127601271	-
256	0.00390625	0.01256096444797522	2.395015596211044
512	0.001953125	0.0027104154026279526	2.212361357172686
1024	0.0009765625	0.0006264383048139033	2.1132696413977325

etd2rk (trapezoidal)

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])

errors_2x_vector, domain = errors_2x(n0, k, etd2rk, t0, tf, x0, A, g, sol,
vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	4.186569175362864e-08	-
256	0.00390625	1.0575183428604418e-08	1.985085775819591
512	0.001953125	2.652380943352073e-09	1.9953227875115886
1024	0.0009765625	6.638462730912398e-10	1.9983668943519293

Naive version of etd2rk (trapezoidal)

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])

errors_2x_vector, domain = errors_2x(n0, k, etd2rk_trapezoidal_naive, t0, tf, x0,
A, g, sol, vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	0.0004242643044311458	-
256	0.00390625	0.00010714498082271644	1.9853990333325726
512	0.001953125	2.6871031228085582e-05	1.9954406751889993
1024	0.0009765625	6.725136514989377e-06	1.9984162299862431

rk4

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])

errors_2x_vector, domain = errors_2x(n0, k, rk4, t0, tf, x0, A, g, sol,
vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	0.002141816843239275	-
256	0.00390625	9.770249694801558e-05	4.4542958704375835
512	0.001953125	5.250705130854794e-06	4.21781234893684
1024	0.0009765625	3.024340525237257e-07	4.1178186851779905

Deduced like etd3rk

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])

errors_2x_vector, domain = errors_2x(n0, k, etd3rk_similar, t0, tf, x0, A, g, sol,
vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

n	h = $\frac{1}{h}$	$\tau(0,h)$	q = $\frac{tau(0,h)}{tau(0, 2h)}$
128	0.0078125	5.08530240486693l5e-12	-
256	0.00390625	3.212833644961055e-13	3.9844153810116354
512	0.001953125	2.0132983821752326e-14	3.996213373698299
1024	0.0009765625	1.2602766052971504e-15	3.997748687591092

Naive version of etd3rk

```
n0 = 128
k = 4
t0 = 0
tf = 1
x0 = np.array([1])
A = np.array([[100]])

errors_2x_vector, domain = errors_2x(n0, k, etd3rk_naive, t0, tf, x0, A, g, sol,
vectorize_sol, error_sup)
convergence_table(errors_2x_vector, n0, k, t0, tf)
```

```
| n | h = $\frac{1}{h}$ | $\tau(0,h)$ | q = $\frac{tau(0,h)}{tau(0, 2h)}$ |
|---|-----------------|-----------|---------------------------------|
 | 128 | 0.0078125 | 1.083876968009309e-06 | - |
 | 256 | 0.00390625 | 6.883813637344194e-08 | 3.9768491535433466 |
 | 512 | 0.001953125 | 4.322307012305515e-09 | 3.9933345852265947 |
 | 1024 | 0.0009765625 | 2.705360744453822e-10 | 3.9979086629155343 |
```

Some graphics

The following notation is used

\begin{cases} u'(t) + A u(t) = g(u(t), t)\ u(0) = u_0. \end{cases}

A Stiff problem shown in [1] is

\begin{cases} u'(t) + 100 u(t) = \sin(t)\ u(0) = u_0, \end{cases}

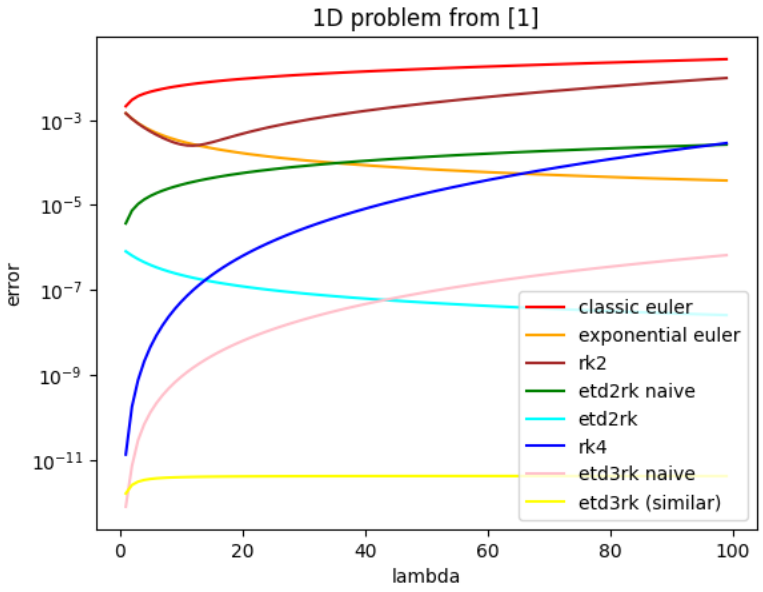
with solution

$$u(t) = u_0 \exp(-100t) + \frac{\exp(-100t) + 100\sin(t) - \cos(t)}{1 + 100^2}.$$

```
n = 128
lmba0 = 1
lmbaf = 100
t0 = 0.0
tf = 1.0
x0 = np.array([1])
lmba_1D_classic, domain = errors_for_lambdas_array(n, classic_euler, t0, tf, x0,
lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_exponential, domain = errors_for_lambdas_array(n, exponential_euler, t0,
tf, x0, lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_etd2rk, domain = errors_for_lambdas_array(n, etd2rk, t0, tf, x0, lmba0,
lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_etd2rk_trapezoidal_naive, domain = errors_for_lambdas_array(n,
etd2rk_trapezoidal_naive, t0, tf, x0, lmba0, lmbaf, A_1D, g, sol_given_lmba,
vectorize_sol_given_lmba, error_2)
lmba_1D_etd3rk_similar, domain = errors_for_lambdas_array(n, etd3rk_similar, t0,
tf, x0, lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_etd3rk_naive, domain = errors_for_lambdas_array(n, etd3rk_naive, t0, tf,
x0, lmba0, lmbaf, A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_rk2, domain = errors_for_lambdas_array(n, rk2, t0, tf, x0, lmba0, lmbaf,
A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
lmba_1D_rk4, domain = errors_for_lambdas_array(n, rk4, t0, tf, x0, lmba0, lmbaf,
A_1D, g, sol_given_lmba, vectorize_sol_given_lmba, error_2)
```

```
/tmp/ipykernel_11908/3543048019.py:247: ComplexWarning: Casting complex values to
real discards the imaginary part
  return np.sqrt(float(np.sum(v)/x_approx.size)) #normalized
```

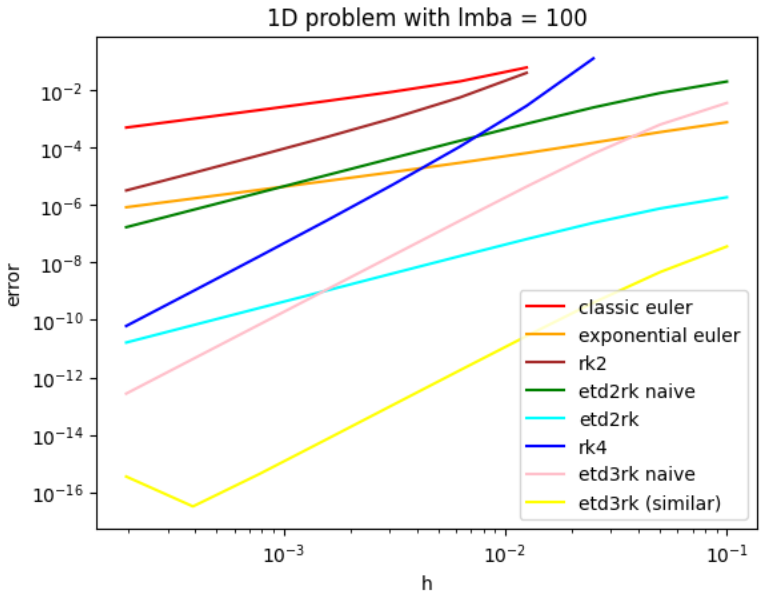
```
matrix_1D = [lmba_1D_classic, lmba_1D_exponential, lmba_1D_rk2,
lmba_1D_etd2rk_trapezoidal_naive, lmba_1D_etd2rk, lmba_1D_rk4,
lmba_1D_etd3rk_naive, lmba_1D_etd3rk_similar]
names = ['classic euler', 'exponential euler', 'rk2', 'etd2rk naive', 'etd2rk',
'rk4', 'etd3rk naive', "etd3rk (similar)"]
fig, ax = graphic_2D(8*[domain], matrix_1D, names, "lambda", "error", "1D problem
from [1]", False, True)
```




```
n0 = 10
k = 10
lmba = 100
A = lmba * np.array([[1]])
t0 = 0.0
tf = 1.0
x0 = np.array([1])
n_1D_classic, domain = errors_2x(n0, k, classic_euler, t0, tf, x0, A, g, sol,
vectorize_sol, error_2)
n_1D_exponential, domain = errors_2x(n0, k, exponential_euler, t0, tf, x0, A, g,
sol, vectorize_sol, error_2)
n_1D_etd2rk, domain = errors_2x(n0, k, etd2rk, t0, tf, x0, A, g, sol,
vectorize_sol, error_2)
n_1D_etd2rk_trapezoidal_naive, domain = errors_2x(n0, k, etd2rk_trapezoidal_naive,
t0, tf, x0, A, g, sol, vectorize_sol, error_2)
n_1D_etd3rk_similar, domain = errors_2x(n0, k, etd3rk_similar, t0, tf, x0, A, g,
sol, vectorize_sol, error_2)
n_1D_etd3rk_naive, domain = errors_2x(n0, k, etd3rk_naive, t0, tf, x0, A, g, sol,
vectorize_sol, error_2)
n_1D_rk2, domain = errors_2x(n0, k, rk2, t0, tf, x0, A, g, sol, vectorize_sol,
error_2)
n_1D_rk4, domain = errors_2x(n0, k, rk4, t0, tf, x0, A, g, sol, vectorize_sol,
error_2)
```

```
/tmp/ipykernel_11908/3543048019.py:247: ComplexWarning: Casting complex values to
real discards the imaginary part
  return np.sqrt(float(np.sum(v)/x_approx.size)) #normalized
```

```
matrix_2D = [n_1D_classic, n_1D_exponential, n_1D_rk2,
n_1D_etd2rk_trapezoidal_naive, n_1D_etd2rk, n_1D_rk4, n_1D_etd3rk_naive,
n_1D_etd3rk_similar]
names = ['classic euler', 'exponential euler', 'rk2', 'etd2rk naive', 'etd2rk',
'rk4', 'etd3rk naive', "etd3rk (similar)"]
fig_2D, ax_2D = graphic_2D(8*[1/domain], matrix_2D, names, "h", "error", "1D
problem with lmba = "+str(lmba), False, True)
plt.xscale('log')
```



Some deductions

Here is used informations from [1], [6], [7].

Exponential Euler method

For

$$\begin{cases} y'(t) + \lambda y(t) = g(y(t), t), t \in (t_0, T) \\ y(0) = y_0 \end{cases}$$

the domain is evenly discretized:

$$N \in \mathbb{N}; h = \frac{T - t_0}{N}; \text{Domain: } \{t_k = t_0 + kh : k = 0, 1, \dots\}.$$

The discretization of the ODE takes the exact solution of the Cauchy problem, given by the variation of constants formula

$$y(t) = e^{-(t-t_0)\lambda} y_0 + \int_{t_0}^{t} [e^{-\lambda(t-\tau)} g(y(\tau), \tau)] d\tau$$

and, by Taylor expansion on g:

$\tau \in (t_k, t_{k+1})$

$$g(y(\tau), \tau) = g(y(t_k), t_k) + (\tau - t_k) \frac{dg}{dt}(y(\theta_k), \theta_k)$$

for a $\theta_k \in (t_k, t_{k+1})$,

$$y(t_{k+1}) = e^{-(t_{k+1}-t_k)\lambda} y(t_k) + \int_{t_k}^{t_{k+1}} [e^{-\lambda(t_{k+1}-\tau)} g(y(\tau), \tau)] d\tau$$

$$= e^{-h\lambda} y(t_k) + \int_{t_k}^{t_{k+1}} \left[e^{-\lambda(t_{k+1}-\tau)} \left(g(y(t_k), t_k) + (\tau - t_k) \frac{dg}{dt}(y(\theta_k), \theta_k) \right) \right] d\tau$$

$$= e^{-h\lambda} y(t_k) + g(y(t_k), t_k) \int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)} d\tau + \frac{dg}{dt}(y(\theta_k), \theta_k) \int_{t_k}^{t_{k+1}} (\tau - t_k) e^{-\lambda(t_{k+1}-\tau)} d\tau.$$

Since

$$\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)} d\tau = h\phi_1(-\lambda h) = \frac{1 - e^{-h\lambda}}{\lambda}$$

and, by the Taylor expansion of $e^{-\lambda h}$ in the point zero

$$e^{-\lambda h} = 1 - \lambda h + \frac{1}{2}\lambda^2 h^2 - \frac{1}{3!}\lambda^3 h^3 + \cdots + \frac{1}{n!}(-\lambda h)^n + \cdots, n \in \mathbb{N}$$

$$\int_{t_k}^{t_{k+1}} (\tau - t_k) e^{-\lambda(t_{k+1}-\tau)} d\tau = h^2 \phi_2(-\lambda h) = h\frac{\phi_1(0) - \phi_1(-\lambda h)}{\lambda} = \frac{h}{\lambda} - \frac{1 - e^{-h\lambda}}{\lambda^2} =$$
$$\frac{h}{\lambda} - \frac{1 - (1 - \lambda h + \frac{1}{2}\lambda^2 h^2 - \frac{1}{3!}\lambda^3 h^3 + \cdots + \frac{1}{n!}(-\lambda h)^n + \cdots)}{\lambda^2} =$$
$$\frac{h^2}{2} - \frac{h^3}{3!}\lambda + \cdots + \frac{h^n}{n!}(-\lambda)^{n-2} + \cdots = O(h^2),$$

$$y(t_{k+1}) = e^{-h\lambda} y(t_k) + g(y(t_k), t_k)\frac{1 - e^{-h\lambda}}{\lambda} + \frac{dg}{dt}(y(\theta_k), \theta_k) O(h^2),$$

$$y(t_{k+1}) = e^{-h\lambda} y(t_k) + g(y(t_k), t_k)\frac{1 - e^{-h\lambda}}{\lambda} + O(h^2).$$

That inspires the **Exponential Euler method** :

$$\begin{aligned} y_0 &= y(t_0) \\ \textbf{for } k &= 0, 1, 2, \ldots, N-1: \\ y_{k+1} &= e^{-h\lambda} y_k + g(y_k, t_k)\frac{1 - e^{-h\lambda}}{\lambda} \\ t_{k+1} &= t_k + h \end{aligned}$$

with $y_k \approx y(t_k)$.

Exponential time differencing methods (ETD)

In the same conditions as above, it is taken a general Taylor expansion of g:

$\tau \in (t_k, t_{k+1}), n \in \mathbb{N}$

$$g(y(\tau), \tau) = g(y(t_k), t_k) + (\tau - t_k)\frac{dg}{dt}(y(t_k), t_k) + \frac{(\tau - t_k)^2}{2!}\frac{d^2 g}{dt^2}(y(t_k), t_k) +$$
$$\cdots + \frac{(\tau - t_k)^{n-1}}{(n-1)!}\frac{d^{n-1} g}{dt^{n-1}}(y(t_k), t_k) + \frac{(\tau - t_k)^n}{n!}\frac{d^n g}{dt^n}(y(\theta_k), \theta_k)$$

for a $\theta_k \in (t_k, t_{k+1})$

In

$$y(t_{k+1}) = e^{-h\lambda} y(t_k) + \int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)} g(y(\tau), \tau) d\tau$$

It will now become

$$y(t_{k+1}) = e^{-h\lambda} y(t_k) + \int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)} g(y(t_k), t_k) + (\tau - t_k)\frac{dg}{dt}(y(t_k), t_k) +$$
$$\frac{(\tau - t_k)^2}{2!}\frac{d^2 g}{dt^2}(y(t_k), t_k) + \cdots +$$
$$+ \frac{(\tau - t_k)^{n-1}}{(n-1)!}\frac{d^{n-1} g}{dt^{n-1}}(y(t_k), t_k) + \frac{(\tau - t_k)^n}{n!}\frac{d^n g}{dt^n}(y(\theta_k), \theta_k) d\tau,$$

$$y(t_{k+1}) = e^{-h\lambda} y(t_k) + g(y(t_k), t_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)} d\tau +$$
$$+ \frac{dg}{dt}(y(t_k), t_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}(\tau - t_k) d\tau + \frac{d^2 g}{dt^2}(y(t_k), t_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}\frac{(\tau - t_k)^2}{2!} d\tau +$$
$$+ \cdots +$$
$$+ \frac{d^{n-1} g}{dt^{n-1}}(y(t_k), t_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}\frac{(\tau - t_k)^{n-1}}{(n-1)!} d\tau + \frac{d^n g}{dt^n}(y(\theta_k), \theta_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}\frac{(\tau - t_k}{n!}$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + h^2\phi_2(-\lambda h)\frac{dg}{dt}(y(t_k),t_k) + h^3\phi_3(-\lambda h)\frac{d^2g}{dt^2}(y(t_k),t_k) + \cdots + h^n\phi_n(-\lambda h)\frac{d^{n-1}g}{dt^{n-1}}(y(t_k),t_k) + h^{n+1}\phi_{n+1}(-\lambda h)\frac{d^ng}{dt^n}(y(\theta_k),\theta_k).$$

From the discussion about the exponential Euler, that is known that

$$h^2\phi_2(-\lambda h) = \frac{h^2}{2} - \frac{h^3}{3!}\lambda + \cdots + \frac{h^l}{l!}(-\lambda)^{l-2} + \cdots = \frac{1}{(-\lambda)^2}\sum_{i=2}^{\infty}\frac{(-\lambda h)^i}{i!}.$$

Since

$$\phi_{n+1}(-\lambda h) = \frac{\phi_n(-\lambda h) - \phi_n(0)}{-\lambda h} \text{ and}$$
$$\phi_n(0) = \frac{1}{n!},$$

$$h^3\phi_3(-\lambda h) = h^2\frac{\phi_2(0) - \phi_2(-\lambda h)}{\lambda} = \frac{\frac{h^2}{2} - (\frac{h^2}{2} - \frac{h^3}{3!}\lambda + \cdots + \frac{h^l}{l!}(-\lambda)^{l-2} + O(h^{l+1}))}{\lambda} = \frac{}{(}$$

And if

$$h^l\phi_l(-\lambda h) = \frac{1}{(-\lambda)^l}\sum_{i=l}^{\infty}\frac{(-\lambda h)^i}{i!}, \text{for a } l \in \mathbb{N},$$

$$h^{l+1}\phi_{l+1}(-\lambda h) = h^{l+1}\frac{\phi_l(-\lambda h) - \phi_l(0)}{-\lambda h} = \frac{h^l\phi_l(0) - h^l\phi_l(-\lambda h)}{\lambda} = \frac{h^l}{l!\lambda} - \frac{1}{\lambda}\frac{1}{(-\lambda)^l}\sum_{i=l}^{\infty}\frac{(-}{}$$

So, by induction,

$$h^n\phi_n(-\lambda h) = \frac{1}{(-\lambda)^n}\sum_{i=n}^{\infty}\frac{(-\lambda h)^i}{i!} = O(h^n), \forall n \geq 2.$$

Then,

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + h^2\phi_2(-\lambda h)\frac{dg}{dt}(y(t_k),t_k) + h^3\phi_3(-\lambda h)\frac{d^2g}{dt^2}($$
$$h^n\phi_n(-\lambda h)\frac{d^{n-1}g}{dt^{n-1}}(y(t_k$$

Exponential time differencing methods with Runge-Kutta time stepping - order 2 - Cox and Matthews - Trapezoidal rule

For the second order method, that is used the approximation

$$g(y(\tau),\tau) = g(y(t_k),t_k) + (\tau - t_k)\frac{dg}{dt}(y(t_k),t_k) + O(h^2),$$

$\forall \tau \in (t_k, t_{k+1})$.

The first derivative is discretized with the Taylor expansion

$$g(y(t_{k+1}),t_{k+1}) = g(y(t_k),t_k) + h\frac{dg}{dt}(y(t_k),t_k) + O(h^2)$$

and the exponential Euler expression

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda} + O(h^2),$$

so that

$$\frac{dg}{dt}(y(t_k),t_k) = \frac{g(a_k,t_{k+1}) - g(y(t_k),t_k)}{h} + O(h),$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda},$$

which results in the expression

$$g(y(\tau),\tau) = g(y(t_k),t_k) + (\tau - t_k)\frac{g(a_k,t_{k+1}) - g(y(t_k),t_k)}{h} + (\tau - t_k)O(h)$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda}.$$

Putting in the variation of constants formula

$$y(t) = e^{-(t-t_0)\lambda}y_0 + \int_{t_0}^{t} e^{-\lambda(t-\tau)}g(y(\tau),\tau)d\tau,$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) +$$
$$+\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}\left[g(y(t_k),t_k) + (\tau - t_k)\frac{g(a_k,t_{k+1}) - g(y(t_k),t_k)}{h} + (\tau - t_k)O(h)\right]d\tau$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}d\tau + \frac{g(a_k,t_{k+1}) - g(y(t_k),t_k)}{h}\int_{t_k}^{t_{k+1}}(\tau -$$
$$+O(h)\int_{t_k}^{t_{k+1}}(\tau$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y($$

Then,

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + \frac{g(a_k,t_{k+1}) - g(y(t_k),t_k)}{h}h^2\phi_2(-\lambda h) +$$
$$+O(h)h^2\phi_2(-\lambda h)$$
$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + [g(a_k,t_{k+1}) - g(y(t_k),t_k)]h\phi_2(-\lambda h) +$$
$$+O(h^3)$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda}.$$

Butcher tableau:

$$\begin{array}{c|cc} 0 & & \\ 1 & \phi_1(-\lambda h) & \\ \hline & \phi_1(-\lambda h) - \phi_2(-\lambda h) & \phi_2(-\lambda h) \end{array}$$

Exponential time differencing methods with Runge-Kutta time stepping - order 2 - Cox and Matthews - Midpoint rule

From the same expression:

$$g(y(\tau),\tau) = g(y(t_k),t_k) + (\tau - t_k)\frac{dg}{dt}(y(t_k),t_k) + O(h^2),$$

$\forall \tau \in (t_k, t_{k+1})$.

The first derivative is now discretized with the Taylor expansion

$$g\left(y\left(t_k + \frac{h}{2}\right), t_k + \frac{h}{2}\right) = g(y(t_k),t_k) + \frac{h}{2}\frac{dg}{dt}(y(t_k),t_k) + O(h^2)$$

and the exponential Euler expression taken is with time step $\frac{h}{2}$

$$y\left(t_k + \frac{h}{2}\right) = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right) + O(h^2),$$

so that

$$\frac{dg}{dt}(y(t_k),t_k) = 2\frac{g\left(b_k, t_k+\frac{h}{2}\right) - g(y(t_k),t_k)}{h} + O(h),$$
$$\text{with } b_k = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right),$$

which results in the expression

$$g(y(\tau),\tau) = g(y(t_k),t_k) + 2(\tau-t_k)\frac{g\left(b_k, t_k+\frac{h}{2}\right) - g(y(t_k),t_k)}{h} + (\tau-t_k)O(h)$$
$$\text{with } b_k = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right).$$

Putting in the variation of constants formula

$$y(t) = e^{-(t-t_0)\lambda}y_0 + \int_{t_0}^{t} e^{-\lambda(t-\tau)}g(y(\tau),\tau)d\tau,$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) +$$
$$+\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}\left[g(y(t_k),t_k) + 2(\tau-t_k)\frac{g\left(b_k, t_k+\frac{h}{2}\right) - g(y(t_k),t_k)}{h} + (\tau-t_k)O(h)\right]d\tau$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}d\tau -$$
$$+2\frac{g\left(b_k, t_k+\frac{h}{2}\right) - g(y(t_k),t_k)}{h}\int_{t_k}^{t_{k+1}}(\tau-t_k)e^{-\lambda(t_{k+1}-\tau)}d\tau + O(h)\int_{t_k}^{t_{k+1}}(\tau-t_k)e^{-\lambda(t_{k+1}-\tau)}d$$
$$\text{with } b_k = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right)$$

Then,

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + 2\frac{g\left(b_k, t_k+\frac{h}{2}\right) - g(y(t_k),t_k)}{h}h^2\phi_2(-\lambda h) -$$
$$+O(h)h^2\phi_2(-\lambda h$$
$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + h\phi_1(-\lambda h)g(y(t_k),t_k) + 2\left[g\left(b_k, t_k+\frac{h}{2}\right) - g(y(t_k),t_k)\right]h\phi_2(-\lambda h) -$$
$$+O(h^3$$
$$\text{with } b_k = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right)$$

Butcher tableau:

\begin{array} {c|cc} 0\ \frac{1}{2} & \frac{1}{2}\phi_1\left(-\frac{\lambda h}{2}\right)\ \hline & \phi_1 (-\lambda h) - 2 \phi_2 (-\lambda h) & -2 \phi_2 (-\lambda h) \end{array}

Exponential time differencing methods with Runge-Kutta time stepping - order 2 - Classical approach - Trapezoidal rule

It is also possible to think the exponential time differencing methods with Runge-Kutta time stepping using the numerical integration, for example, for the one with second order, it starts with the trapezoidal rule (which was taken from [2]) on the variation of constants formula:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + \frac{h}{2}\left[e^{-\lambda(t_{k+1}-t_k)}g(y(t_k),t_k) + e^{-\lambda(t_{k+1}-t_{k+1})}g(y(t_{k+1}),t_{k+1})\right] + O(h^3),$$
$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + \frac{h}{2}\left[e^{-\lambda h}g(y(t_k),t_k) + g(y(t_{k+1}),t_{k+1})\right] + O(h^3).$$

And then, from the expression seen before:

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda} + O(h^2),$$

$$g(y(t_{k+1}),t_{k+1}) = g(a_k,t_{k+1}) + O(h^2), \text{ with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)\frac{1-e^{-h\lambda}}{\lambda}.$$

So,

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + \frac{h}{2}\left[e^{-\lambda h}g(y(t_k),t_k) + g(a_k,t_{k+1}) + O(h^2)\right] + O(h^3),$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + \frac{h}{2}\left[e^{-\lambda h}g(y(t_k),t_k) + g(a_k,t_{k+1})\right] + O(h^3)$$
$$\text{with } a_k = e^{-h\lambda}y(t_k) + g(y(t_k),t_k)h\phi_1(-\lambda h).$$

Butcher tableau:

$$\begin{array}{c|cc} 0 & & \\ 1 & \phi_1(-\lambda h) & \\ \hline & \frac{1}{2}e^{-h\lambda} & \frac{1}{2} \end{array}$$

Exponential time differencing methods with Runge-Kutta time stepping - order 2 - Classical approach - Midpoint rule

Besides that, using the midpoint rule, also known as rectangle rule, again taken from [2],

$$y(t_{k+1}) = e^{-(t_{k+1}-t_k)\lambda}y(t_k) + \int_{t_k}^{t_{k+1}} [e^{-\lambda(t_{k+1}-\tau)}g(y(\tau),\tau)]d\tau,$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + he^{-\lambda\left(t_{k+1}-\frac{t_{k+1}+t_k}{2}\right)}g\left(y\left(\frac{t_{k+1}+t_k}{2}\right),\frac{t_{k+1}+t_k}{2}\right) + O(h^3),$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + he^{-\frac{h\lambda}{2}}g\left(y\left(t_k+\frac{h}{2}\right),t_k+\frac{h}{2}\right) + O(h^3),$$

and Exponential Euler with time step $\frac{h}{2}$

$$y\left(t_k+\frac{h}{2}\right) = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right) + O(h^2),$$

results in

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + he^{-\frac{h\lambda}{2}}g\left(b_k + O(h^2), t_k+\frac{h}{2}\right) + O(h^3)$$
$$\text{with } b_k = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right),$$

$$y(t_{k+1}) = e^{-h\lambda}y(t_k) + he^{-\frac{h\lambda}{2}}g\left(b_k, t_k+\frac{h}{2}\right) + O(h^3)$$
$$\text{with } b_k = e^{-\frac{h\lambda}{2}}y(t_k) + g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right),$$

Butcher tableau:

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2}\phi_1(-\frac{\lambda h}{2}) & \\ \hline & 0 & e^{-\frac{h\lambda}{2}} \end{array}$$

Third order exponential time differencing methods with Runge-Kutta time stepping (ETDRK-3)

$$g(y(\tau),\tau) = g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \left(\tau - t_{k+\frac{1}{2}}\right)\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) +$$
$$+\frac{\left(\tau - t_{k+\frac{1}{2}}\right)^2}{2!}\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) +$$
$$+\frac{\left(\tau - t_{k+\frac{1}{2}}\right)^3}{3!}\frac{d^3g}{dt^3}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + O((\tau-t_k)^4),$$

$\forall \tau \in \mathbb{R}$.

$$g(y(t_{k+1}),t_{k+1}) = g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \left(t_{k+1}-t_{k+\frac{1}{2}}\right)\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+ \\ +\frac{\left(t_{k+1}-t_{k+\frac{1}{2}}\right)^2}{2!}\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+ \\ +\frac{\left(t_{k+1}-t_{k+\frac{1}{2}}\right)^3}{3!}\frac{d^3g}{dt^3}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+O(h^4),$$

$$g(y(t_{k+1}),t_{k+1}) = g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \frac{h}{2}\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+ \\ +\frac{h^2}{8}\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \frac{h^3}{48}\frac{d^3g}{dt^3}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+O(h^4),$$

and

$$g(y(t_k),t_k) = g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \left(t_{k}-t_{k+\frac{1}{2}}\right)\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+ \\ +\frac{\left(t_{k}-t_{k+\frac{1}{2}}\right)^2}{2!}\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+ \\ +\frac{\left(t_{k}-t_{k+\frac{1}{2}}\right)^3}{3!}\frac{d^3g}{dt^3}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+O(h^4),$$

$$g(y(t_k),t_k) = g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) - \frac{h}{2}\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+ \\ +\frac{h^2}{8}\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) - \frac{h^3}{48}\frac{d^3g}{dt^3}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+O(h^4).$$

Subtracting the two expressions,

$$g(y(t_{k+1}),t_{k+1}) - g(y(t_k),t_k) = h\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+O(h^3).$$

So,

$$\frac{dg}{dt}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) = \frac{g(y(t_{k+1}),t_{k+1}) - g(y(t_k),t_k)}{h}+O(h^2).$$

And summing them

$$g(y(t_{k+1}),t_{k+1}) + g(y(t_k),t_k) = 2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \frac{h^2}{4}\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+O(h^4).$$

So,

$$\frac{d^2g}{dt^2}\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) = 4\frac{g(y(t_{k+1}),t_{k+1}) + g(y(t_k),t_k) - 2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)}{h^2}+O(h^2).$$

This results in the expression

$$g(y(\tau),\tau) = g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right) + \left(\tau-t_{k+\frac{1}{2}}\right)\frac{g(y(t_{k+1}),t_{k+1}) - g(y(t_k),t_k)}{h}+ \\ +\frac{\left(\tau-t_{k+\frac{1}{2}}\right)^2}{2!}\frac{4\left[g(y(t_{k+1}),t_{k+1}) + g(y(t_k),t_k) - 2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}+ \\ +O((\tau-t_k)^3),$$

$\forall \tau \in \mathbb{R}$.

Putting in the variation of constants formula

$$y(t) = e^{-(t-t_0)\lambda}y_0 + \int_{t_0}^{t} e^{-\lambda(t-\tau)}g(y(\tau),\tau)d\tau,$$

$$+\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}\left[g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)+\left(\tau-t_{k+\frac{1}{2}}\right)\frac{g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)}{h}++\frac{\left(\tau\right.}{}\right.$$

$$\begin{aligned}
y(t_{k+1}) = e^{-h\lambda}y(t_k)+\\
g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
\frac{g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)}{h}\int_{t_k}^{t_{k+1}}\left(\tau-t_{k+\frac{1}{2}}\right)e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
+\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}\int_{t_k}^{t_{k+1}}\frac{\left(\tau-t_{k+\frac{1}{2}}\right)^2}{2!}e^{-\lambda(t_{k+1}-\tau)}d\tau\\
+\int_{t_k}^{t_{k+1}} O((\tau-t_k)^3)e^{-\lambda(t_{k+1}-\tau)}d\tau,
\end{aligned}$$

Since $\tau-t_{k+\frac{1}{2}}=\tau-t_k-\frac{h}{2}$ and $\left(\tau-t_{k+\frac{1}{2}}\right)^2=(\tau-t_k)^2+\frac{h^2}{4}-h(\tau-t_k)$,

$$\begin{aligned}
y(t_{k+1}) = e^{-h\lambda}y(t_k)+g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
\frac{g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)}{h}\int_{t_k}^{t_{k+1}}(\tau-t_k)e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
+\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}\int_{t_k}^{t_{k+1}}\frac{(\tau-t_k)^2}{2!}e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
-\frac{g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)}{h}\int_{t_k}^{t_{k+1}}\frac{h}{2}e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
+\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}\int_{t_k}^{t_{k+1}}\frac{h^2}{4\cdot 2!}e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
-\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}\int_{t_k}^{t_{k+1}}\frac{h(\tau-t_k)}{2!}e^{-\lambda(t_{k+1}-\tau)}d\tau+\\
+\int_{t_k}^{t_{k+1}} O((\tau-t_k)^3)e^{-\lambda(t_{k+1}-\tau)}d\tau.
\end{aligned}$$

Then,

$$\begin{aligned}
y(t_{k+1}) = e^{-h\lambda}y(t_k)+\\
g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)h\phi_1(-h\lambda)+\\
\frac{g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)}{h}h^2\phi_2(-h\lambda)+\\
+\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}h^3\phi_3(-h\lambda)+\\
-\frac{g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)}{h}\frac{h^2\phi_1(-h\lambda)}{2}+\\
+\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}\frac{h^3\phi_1(-h\lambda)}{8}+\\
-\frac{4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]}{h^2}\frac{h^3\phi_2(-h\lambda)}{2}+\\
+O(h^4\phi_4(-h\lambda)).
\end{aligned}$$

i.e.

$$
\begin{aligned}
y(t \\
g\left(y\left(t_{k+\frac{1}{2}}\right)\right. \\
[g(y(t_{k+1}),t_{k+1})-g(y(t_k),t_k)]\left(h\phi_2(-h\lambda\right. \\
+4\left[g(y(t_{k+1}),t_{k+1})+g(y(t_k),t_k)-2g\left(y\left(t_{k+\frac{1}{2}}\right),t_{k+\frac{1}{2}}\right)\right]\left(h\phi_3(-h\lambda)+\frac{h\phi_1(-h\lambda)}{8}-\frac{h\phi_2(-h\lambda)}{2}\right)+
\end{aligned}
$$

Using the Cox and Mathhews's ETDRK-2 expressions to approximate $y\left(t_{k+\frac{1}{2}}\right)$ and $y(t_{k+1})$, since those are of order 2, i.e., $O(h^3)$, the expression of the method is

$$
\begin{aligned}
y(t_{k+1}) = e^{-h\lambda} \\
g\left(c'_k,t_{k+\frac{1}{2}}\right)h\phi_1(\\
[g(c_k,t_{k+1})-g(y(t_k),t_k)]\left(h\phi_2(-h\lambda)-\frac{h\phi_1(-h\lambda)}{2}\right. \\
+4\left[g(c_k,t_{k+1})+g(y(t_k),t_k)-2g\left(c'_k,t_{k+\frac{1}{2}}\right)\right]\left(h\phi_3(-h\lambda)+\frac{h\phi_1(-h\lambda)}{8}-\frac{h\phi_2(-h\lambda)}{2}\right)+
\end{aligned}
$$

with

$$
\begin{aligned}
c_k = e^{-h\lambda}y(t_k)+ \\
h\phi_1(-\lambda h)g(y(t_k),t_k)+ \\
[g(a_k,t_{k+1})-g(y(t_k),t_k)]h\phi_2(-\lambda h), \\
a_k = e^{-h\lambda}y(t_k)+g(y(t_k),t_k)h\phi_1(-h\lambda), \\
c'_k = e^{-\frac{h\lambda}{2}}y(t_k)+ \\
\frac{h}{2}\phi_1\left(-\frac{\lambda h}{2}\right)g(y(t_k),t_k)+ \\
\left[g\left(a'_k,t_{k+\frac{1}{2}}\right)-g(y(t_k),t_k)\right]\frac{h}{2}\phi_2\left(-\frac{\lambda h}{2}\right), \\
a'_k = e^{-\frac{h\lambda}{2}}y(t_k)+g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\frac{h\lambda}{2}\right).
\end{aligned}
$$

Here de deducing isn't exactly in Runge Kutta form, differing on the approximations for steps in minor order, so it cannot be a Butcher tableau, but doing language abuse only on the part that would form the triangle, it would be:

\begin{array} {c|cccc} 0 \ \frac{1}{2} & \frac{1}{2} \left(\phi_1\left(- \frac{\lambda h}{2} \right) - \phi_2\left(- \frac{\lambda h}{2} \right) \right) & \frac{1}{2}\phi_2\left(- \frac{\lambda h}{2} \right) \ 1 & \phi_1\left(- \lambda h \right) - \phi_2\left(- \lambda h \right) & 0 & \phi_2\left(- \lambda h \right) \ \hline & 4 \phi_3(-h \lambda)-3\phi_2(-h\lambda)+\phi_1(-h\lambda) & -8\phi_3(-h\lambda)+4\phi_2(-h\lambda) & 4 \phi_3(-h\lambda)-\phi_2(-h\lambda) \text{ }. \end{array}

Naive etd3rk

Here, that is taken the variation of constants formula:

$$
y(t_{k+1}) = e^{-h\lambda}y(t_k) + \int_{t_k}^{t_{k+1}} e^{-\lambda(t_{k+1}-\tau)}g(y(\tau),\tau)d\tau,
$$

and applied the Simpson's rule (here was used the order of convergence from Burden) so that it will be:

$$
\begin{aligned}
y(t_{k+1}) = e^{-h\lambda}y \\
\frac{h}{6}\left[e^{-\lambda(t_{k+1}-t_k)}g(y(t_k),t_k)+4e^{-\lambda\left(t_{k+1}-t_{k+\frac{1}{2}}\right)}g\left(y\left(t_{k+\frac{1}{2}}\right),t_k+\frac{h}{2}\right)+e^{-\lambda(t_{k+1}-t_{k+1})}g(y(t_{k+1}),t\right. \\
+O \\
y(t_{k+1}) = e^{-h\lambda}y \\
\frac{h}{6}\left[e^{-\lambda h}g(y(t_k),t_k)+4e^{-\frac{\lambda h}{2}}g\left(y\left(t_k+\frac{h}{2}\right),t_k+\frac{h}{2}\right)+g(y(t_{k+1}),t\right. \\
+O
\end{aligned}
$$

To approximate $y\left(t_k+\frac{h}{2}\right)$:

$$y\left(t_k+\frac{h}{2}\right)=e^{-\frac{h\lambda}{2}}y(t_k)+\frac{h}{4}\left[e^{-\frac{h\lambda}{2}}g(y(t_k),t_k)+g\left(a'_k,t_k+\frac{h}{2}\right)\right]+O(h^3),$$
$$\text{with } a'_k=e^{-\frac{h\lambda}{2}}y(t_k)+g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\lambda\frac{h}{2}\right),$$

and, for $y\left(t_{k+1}\right)$:

$$y(t_{k+1})=e^{-h\lambda}y(t_k)+\frac{h}{2}\left[e^{-\lambda h}g(y(t_k),t_k)+g(a_k,t_{k+1})\right]+O(h^3),$$
$$\text{with } a_k=e^{-h\lambda}y(t_k)+g(y(t_k),t_k)h\phi_1(-\lambda h).$$

So, the expression is

$$y(t_{k+1})=e^{-h\lambda}y(t_k)+\frac{h}{6}\left[e^{-\lambda h}g(y(t_k),t_k)+4e^{-\frac{\lambda h}{2}}g\left(b'_k,t_k+\frac{h}{2}\right)+g(b_k,t_{k+1})\right]+O(h^4),$$
$$\text{with } b'_k=e^{-\frac{h\lambda}{2}}y(t_k)+\frac{h}{4}\left[e^{-\frac{h\lambda}{2}}g(y(t_k),t_k)+g\left(a'_k,t_k+\frac{h}{2}\right)\right],$$
$$b_k=e^{-h\lambda}y(t_k)+\frac{h}{2}\left[e^{-\lambda h}g(y(t_k),t_k)+g(a_k,t_{k+1})\right],$$
$$a'_k=e^{-\frac{h\lambda}{2}}y(t_k)+g(y(t_k),t_k)\frac{h}{2}\phi_1\left(-\lambda\frac{h}{2}\right),$$
$$a_k=e^{-h\lambda}y(t_k)+g(y(t_k),t_k)h\phi_1(-\lambda h).$$

Matrix exponential

This part has information from [4].

Based on the Maclaurin series of the exponential function

$$e^x=\sum_{i=0}^{\infty}\frac{x^i}{i!},$$

the **exponential of a square complex matrix** A is defined as

$$e^A\doteq\sum_{i=0}^{\infty}\frac{A^i}{i!}.$$

This is well defined because it has been proven that the sequence p_k with, $\forall k\in\mathbb{N}$:

$$p_k=\sum_{i=0}^{k}\frac{A^i}{i!},\forall A \text{ as decribed above},$$

is a Cauchy sequence, and therefore converge to a limit matrix which was denoted e^A, since the set of the square complex matrix with fixed lenght with the norm

$$||A||=\max_{||x||=1}||Ax||$$

is a Banach space.

Exponential of a zeros matrix

If $A=\begin{bmatrix}0&0&0&\cdots&0\\0&0&0&\cdots&0\\0&0&0&\cdots&0\\\vdots&\vdots&\vdots&\ddots&\vdots\\0&0&0&\cdots&0\end{bmatrix}$,

$$e^A\doteq\sum_{i=0}^{\infty}\frac{A^i}{i!}=I+A+\frac{A^2}{2}+\cdots=I+0+0+\cdots=I.$$

Exponential of a diagonal matrix

If $A = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ 0 & 0 & \lambda_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_N \end{bmatrix} = diag(\lambda_1, \lambda_2, \lambda_3, \cdots, \lambda_N),$

it is easy to note that

$$A^2 = diag\left(\lambda_1^2, \lambda_2^2, \lambda_3^2, \ldots, \lambda_N^2\right)$$

$$A^3 = diag\left(\lambda_1^3, \lambda_2^3, \lambda_3^3, \ldots, \lambda_N^3\right)$$

$$\vdots$$

$$A^j = diag\left(\lambda_1^j, \lambda_2^j, \lambda_3^j, \ldots, \lambda_N^j\right), \forall j \in \mathbb{N}$$

$$\vdots$$

so

$$e^A \doteq \sum_{i=0}^{\infty} \frac{A^i}{i!} = diag\left(\sum_{i=0}^{\infty} \frac{\lambda_1^i}{i!}, \sum_{i=0}^{\infty} \frac{\lambda_2^i}{i!}, \sum_{i=0}^{\infty} \frac{\lambda_3^i}{i!}, \ldots, \sum_{i=0}^{\infty} \frac{\lambda_N^i}{i!}\right)$$

$$= diag\left(e^{\lambda_1}, e^{\lambda_2}, e^{\lambda_3}, \ldots, e^{\lambda_N}\right).$$

In the same way, if B is a diagonal by blocks matrix:

$$B = \begin{bmatrix} B_1 & 0 & 0 & \cdots & 0 \\ 0 & B_2 & 0 & \cdots & 0 \\ 0 & 0 & B_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & B_N \end{bmatrix} = diag(B_1, B_2, B_3, \cdots, B_N),$$

then

$$e^B = diag(e^{B_1}, e^{B_2}, e^{B_3}, \cdots, e^{B_N}).$$

Exponential of a matrix of ones above the diagonal

If $A = A_{N \times N} = \begin{bmatrix} 0 & 1 & & & & & \\ & 0 & 1 & & & & \\ & & 0 & 1 & & & \\ & & & 0 & 1 & & \\ & & & & 0 & \ddots & \\ & & & & & \ddots & 1 \\ & & & & & & 0 \end{bmatrix},$

one can calculate

$$A^2 = A \cdot A = \begin{bmatrix} 0 & 1 & & & & & \\ & 0 & 1 & & & & \\ & & 0 & 1 & & & \\ & & & 0 & 1 & & \\ & & & & 0 & \ddots & \\ & & & & & \ddots & 1 \\ & & & & & & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & & & & & \\ & 0 & 1 & & & & \\ & & 0 & 1 & & & \\ & & & 0 & 1 & & \\ & & & & 0 & \ddots & \\ & & & & & \ddots & 1 \\ & & & & & & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 1 & & & & \\ & 0 & 0 & 1 & & & \\ & & 0 & 0 & 1 & & \\ & & & 0 & 0 & \ddots & \\ & & & & 0 & \ddots & 1 \\ & & & & & \ddots & 0 \\ & & & & & & 0 \end{bmatrix},$$

$$A^3 = A \cdot A^2 = \begin{bmatrix} 0 & 1 & & & & & \\ & 0 & 1 & & & & \\ & & 0 & 1 & & & \\ & & & 0 & 1 & & \\ & & & & 0 & \ddots & \\ & & & & & \ddots & 1 \\ & & & & & & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 & & & & \\ & 0 & 0 & 1 & & & \\ & & 0 & 0 & 1 & & \\ & & & 0 & 0 & \ddots & \\ & & & & 0 & \ddots & 1 \\ & & & & & \ddots & 0 \\ & & & & & & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 1 & & & \\ & 0 & 0 & 0 & 1 & & \\ & & 0 & 0 & 0 & \ddots & \\ & & & 0 & 0 & \ddots & 1 \\ & & & & 0 & \ddots & 0 \\ & & & & & \ddots & 0 \\ & & & & & & 0 \end{bmatrix},$$

$$\vdots$$

$$A^{N-2} = \begin{bmatrix} & & & 0 & 1 & 0 \\ & & & & 0 & 1 \\ & & & & & 0 \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix},$$

$$A^{N-1} = \begin{bmatrix} & & & & & 1 \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix},$$

$$A^N = 0.$$

And then, with $t \in \mathbb{R}$

$$e^{tA} \doteq \sum_{i=0}^{\infty} \frac{tA^i}{i!}$$

$$= Id + tA + \frac{t^2A^2}{2} + \frac{t^3A^3}{6} + \ldots + \frac{t^{N-2}A^{N-2}}{(N-2)!} + \frac{t^{N-1}A^{N-1}}{(N-1)!} + 0 + 0 + \ldots + 0$$

$$= \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} + \begin{bmatrix} 0 & t & & & & & \\ & 0 & t & & & & \\ & & 0 & t & & & \\ & & & 0 & t & & \\ & & & & 0 & \ddots & \\ & & & & & \ddots & t \\ & & & & & & 0 \end{bmatrix} +$$

$$+ \begin{bmatrix} 0 & 0 & \frac{t^2}{2} & & & & \\ & 0 & 0 & \frac{t^2}{2} & & & \\ & & 0 & 0 & \frac{t^2}{2} & & \\ & & & 0 & 0 & \ddots & \\ & & & & 0 & \ddots & \frac{t^2}{2} \\ & & & & & \ddots & 0 \\ & & & & & & 0 \end{bmatrix} + \ldots + \begin{bmatrix} & & & & & & \frac{t^{N-1}}{(N-1)!} \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \end{bmatrix}$$

$$= \begin{bmatrix} 1 & t & \frac{t^2}{2} & \frac{t^3}{3!} & \frac{t^4}{4!} & \cdots & \frac{t^{N-1}}{(N-1)!} \\ & 1 & t & \frac{t^2}{2} & \frac{t^3}{3!} & \ddots & \vdots \\ & & 1 & t & \frac{t^2}{2} & \ddots & \frac{t^4}{4!} \\ & & & 1 & t & \ddots & \frac{t^3}{3!} \\ & & & & 1 & \ddots & \frac{t^2}{2} \\ & & & & & \ddots & t \\ & & & & & & 1 \end{bmatrix}.$$

Exponential of a Jordan block

Proposition: $A_1, A_2 \in \mathscr{M}_{N\times N}(\mathbb{C})$. If $A_1 \cdot A_2 = A_2 \cdot A_1$, then $e^{A_1+A_2} = e^{A_1} \cdot e^{A_2}$.

A Jordan block is of the form: $$J = \begin{bmatrix} \lambda_i & 1 & 0 & \cdots & 0 \\ 0 & \lambda_i & 1 & \cdots & 0 \\ 0 & 0 & \lambda_i & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 \\ 0 & 0 & 0 & \cdots & \lambda_i \end{bmatrix}$$

$$= \begin{bmatrix} \lambda_i & 0 & 0 & \cdots & 0 \\ 0 & \lambda_i & 0 & \cdots & 0 \\ 0 & 0 & \lambda_i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i \end{bmatrix} + \begin{bmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & 0 & \ddots & \\ & & & \ddots & 1 \\ & & & & 0 \end{bmatrix}$$

$$= D + N,$$

and $$\begin{bmatrix} \lambda_i & 0 & 0 & \cdots & 0 \\ 0 & \lambda_i & 0 & \cdots & 0 \\ 0 & 0 & \lambda_i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & 0 & \ddots & \\ & & & \ddots & 1 \\ & & & & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \lambda_i & & & \\ & 0 & \lambda_i & & \\ & & 0 & \ddots & \\ & & & \ddots & \lambda_i \\ & & & & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & 0 & \ddots & \\ & & & \ddots & 1 \\ & & & & 0 \end{bmatrix} \cdot \begin{bmatrix} \lambda_i & 0 & 0 & \cdots & 0 \\ 0 & \lambda_i & 0 & \cdots & 0 \\ 0 & 0 & \lambda_i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i \end{bmatrix},$$

so

$$e^{tJ} = e^{tD+tN} = e^{tD} \cdot e^{tN}$$

$$= \begin{bmatrix} e^{t\lambda_i} & 0 & 0 & \cdots & 0 \\ 0 & e^{t\lambda_i} & 0 & \cdots & 0 \\ 0 & 0 & e^{t\lambda_i} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & e^{t\lambda_i} \end{bmatrix} \cdot \begin{bmatrix} 1 & t & \frac{t^2}{2} & \cdots & \frac{t^{N-1}}{(N-1)!} \\ & 1 & t & \ddots & \vdots \\ & & 1 & \ddots & \frac{t^2}{2} \\ & & & \ddots & t \\ & & & & 1 \end{bmatrix}$$

$$= \begin{bmatrix} e^{t\lambda_i} & e^{t\lambda_i}t & \frac{e^{t\lambda_i}t^2}{2} & \cdots & \frac{e^{t\lambda_i}t^{N-1}}{(N-1)!} \\ & e^{t\lambda_i} & e^{t\lambda_i}t & \ddots & \vdots \\ & & e^{t\lambda_i} & \ddots & \frac{e^{t\lambda_i}t^2}{2} \\ & & & \ddots & e^{t\lambda_i}t \\ & & & & e^{t\lambda_i} \end{bmatrix}, t \in \mathbb{R}.$$

Exponential of any matrix

Proposition: $\forall A \in \mathscr{M}_{N\times N}(\mathbb{C}), \exists M \in \mathscr{M}_{N\times N}(\mathbb{C})$ invertible, such that $A = MJM^{-1}$, with

$$J = \begin{bmatrix} J_1 & 0 & 0 & \cdots & 0 \\ 0 & J_2 & 0 & \cdots & 0 \\ 0 & 0 & J_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & J_N \end{bmatrix}$$

and each J_i, $i = 1, 2, 3, \ldots, N$ being a Jordan block, i.e.,

$$J_i = \begin{bmatrix} \lambda_i & 0 & 0 & \cdots & 0 \\ 0 & \lambda_i & 0 & \cdots & 0 \\ 0 & 0 & \lambda_i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i \end{bmatrix}$$

for some $\lambda_i \in \mathbb{C}$.

Note that $$(MJM^{-1})^k = MJM^{-1}MJM^{-1}MJM^{-1}\ldots MJM^{-1}$$

$$= MJIJIJM^{-1}\ldots MJM^{-1} = MJJJ\ldots JM^{-1} = MJ^kM^{-1}.$$

Because of the formula of the series that defines the expansion, it implicates in $e^{MJM^{-1}} = Me^{J}M^{-1}$
.

And then, using the same notation from the last proposition, $
$e^{tA} = e^{tMJM^{-1}} = e^{MtJM^{-1}} = Me^{tJ}M^{-1}$$

$$= M \begin{bmatrix} e^{tJ_1} & 0 & 0 & \cdots & 0 \\ 0 & e^{tJ_2} & 0 & \cdots & 0 \\ 0 & 0 & e^{tJ_3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & e^{tJ_N} \end{bmatrix} M^{-1}, t \in \mathbb{R},$$

with each block as the section above indicates.

Euler method

Further detailing this explicit one-step method of

$$\phi(t_k, y_k, h) = f(t_k, y_k),$$

an analysis on stability, convergence and order of convergence is done.

Stability

For the problem $\begin{cases} y'(t) = -\lambda y(t) \, ; t \in [t_0, T] \\ y(t_0) = y_0, \end{cases}$

with known solution

$$y(t) = y_0 e^{-\lambda(t - t_0)},$$

the method turn into:

$$\begin{aligned} y_0 &= y(t_0) \\ \textbf{for } k &= 0, 1, 2, \ldots, N-1 : \\ y_{k+1} &= y_k + h\lambda y_k \\ t_{k+1} &= t_k + h. \end{aligned}$$

Then the amplification factor is: $$(1 - h\lambda).$$

If

$$|1 - h\lambda| > 1, \text{for fixed } N,$$

it will be a divergent series

$$(k \to \infty \Rightarrow y_k \to \infty),$$

so, since the computer has a limitant number that can represent, even if the number of steps is such that h is not small enought, it might have sufficient steps to reach the maximum number represented by the machine.

However, if

$$|1 - h\lambda| < 1 \text{ and } N \text{ is fixed},$$

it converges to zero

$$(k \to \infty \Rightarrow y_k \to 0).$$

Besides that,

$$|1 - h\lambda| < 1$$

is the same as

$$0 < h\lambda < 2.$$

So the interval of stability is $(0, 2)$.

That's why the method suddenly converged, it was when h got small enought to $h\lambda$ be in the interval of stability, i.e.,

$$h < 2/\lambda.$$

It is worth mentioning here that if

$$-1 < 1 - h\lambda < 0,$$

the error will converge oscillating since it takes positive values with even exponents and negative with odd ones.

Convergence

Since

$$\lim_{m \to +\infty} \left(1 + \frac{p}{m}\right)^m = e^p,$$

and h = $\frac{T - t_0}{N}$, for y_N we have

$$\lim_{N \to +\infty} y_N = \lim_{N \to +\infty} (1 - h\lambda)^N y_0 = \lim_{N \to +\infty} \left(1 - \frac{(T - t_0)\lambda}{N}\right)^N y_0.$$

It is reasonable to take $p = -(T - t_0)\lambda$ and conclude that the last point estimated by the method will converge to

$$y_0 e^{-\lambda(T - t_0)}.$$

Which is precisely $y(T)$ and proves the convergence.

Order of convergence

Being $\tau(h, t_k)$ the local truncation error.

From

$$y(t_{k+1}) = y(t_k) + hf(y(t_k), t_k) + O(h^2),$$

we have

$$h\tau(h, t_k) \doteq \frac{y(t_{k+1}) - y(t_k)}{h} - f(t_k, y(t_k)) = O(h^2),$$

so

$$\tau(h, t_k) = O(h).$$

Since for one step methods the order of convergence is the order of the local truncation error, the order is of $O(h)$, order 1.

By Isabela Miki Suzuki