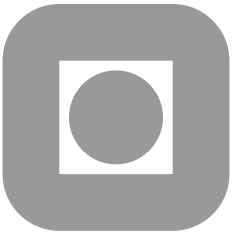
NORGES TEKNISK-NATURVITENSKAPELIGE
UNIVERSITET

EXPINT — A MATLAB[1] package for exponential integrators

by

Håvard Berland, Bård Skaflestad and Will Wright

PREPRINT
NUMERICS NO. 4/2005



NORWEGIAN UNIVERSITY OF SCIENCE AND
TECHNOLOGY
TRONDHEIM, NORWAY

This report has URL `http://www.math.ntnu.no/preprint/numerics/2005/N4-2005.ps`
Address: Department of Mathematical Sciences, Norwegian University of Science and Technology, N-7491 Trondheim, Norway.

[1] MATLAB is a registered trademark of The MathWorks, Inc.

Abstract

Recently, a great deal of attention has been focused on the construction of exponential integrators for semi-linear problems. In this paper we describe a MATLAB package which aims to facilitate the quick deployment and testing of exponential integrators, of Runge–Kutta, multistep and general linear type. A large number of integrators are included in this package along with several well known examples. The so-called φ-functions and their evaluation is crucial for stability and speed of exponential integrators, and the approach taken here is through a modification of the scaling and squaring technique; the most common approach used for computing the matrix exponential.

Contents

1 Introduction

EXPINT is a MATLAB package designed as a tool for the numerical experimentalist to facilitate easy testing of various exponential integrators. Exponential integrators are a class of numerical methods specifically designed for the numerical solution of semi-linear problems and have recently had a renewed interest in their development, generally tailored to providing efficient time integration technology for semi-discretized PDEs. Exponential integrators are essentially an alternative to implicit methods for the numerical solution of stiff or highly oscillatory differential equations. They aim to solve exactly the linear part of the problem, which is assumed to contain the substantial part of the dynamics of the problem and then numerically solve the remaining part. The integrators are designed to be explicit and the coefficients of the method are functions of the linear term. The aim of the EXPINT package is threefold: Create a uniform environment which enables the comparison of various integrators; Provide tools for easy visualizing of numerical behaviour; To be easily modified so that users can include problems and integrators of their own. We set out to provide as an efficient implementation as possible given that we do not compromise generality and ease of comparison and usability.

The outline of the paper is as follows. In the next section we outline the system requirements for the EXPINT package. Section 2 describes what exponential integrators are and how they are formatted for our context. Exponential integrators of Runge–Kutta type and multistep-type are generalized using general linear methods, and our way of writing the integrators is tailored for this. In Section 3 we outline installation of this package and describe the package structure. Equations and boundary conditions are represented in a problem structure described in Section 4, where also some details regarding some of the examples included in this package can be found. Formulae for the diagonal Padé approximations and a generalization of the squaring process for the φ-functions is outlined in Section 5. Functions essential to the package are briefly described in a user-oriented way in Section 6. In the last two sections some numerical experiments

are recorded and we address future work. Integrators included in this package are listed for user reference in Appendix A, we also briefly comment on their origin.

1.1 System requirements for the EXPINT package

In terms of computing power, the system requirements are quite modest. Any computer capable of running a moderately up-to-date version of MATLAB is sufficient for experimenting with exponential integrators by means of the EXPINT package. On the other hand, faster central processing units and memory systems decrease actual runtime.

However, the package places more stringent demands on the actual MATLAB environment. In particular, some of the package's fundamental routines use MATLAB language features and core functions which were introduced relatively recently. As a consequence, the package will not run without modification in MATLAB environments preceding the release 13 series (ie. MATLAB 6.5). The following features, introduced in MATLAB 6.5, are needed

- dynamic structure field-names
- short-circuiting logical operators
- regular expression support
- the `mat2cell` function which, although present in the neural networks toolbox prior to MATLAB 6.5, was not introduced into the core language until version 6.5.
- multiple argument version of functions `warning` and `error`

We note that it is possible to work around most of these requirements, at least at the expense of a slight reduction of the package's functionality, but the authors have elected to support only recent versions of MATLAB.

Unfortunately, a backwards incompatible change in the way dynamic function resolution is performed, introduced in the release 14 (ie. MATLAB 7) series, forces some important changes to the package. Particularly, the `integrator` function described in Section 6.1 of this manual, is heavily dependent upon dynamic function binding and resolution (ie. callbacks), by means of the core MATLAB function `feval`. The `integrator` function must be rewritten in terms of calls through function handles to function properly in MATLAB 7 and later. We note that callbacks through function handles are potentially more efficient than using the `feval` function. This is due to function handles eliminating the path search inherent in `feval`.

Fortuitously such a rewrite allows a reduction in the number of files, though not necessarily the number of statements, needed to implement a new semi-linear problem. This is due to MATLAB 7's increasing the capabilities of function handles and introducing anonymous functions. Currently, however, due to external limitations, the EXPINT package will only run without modification in MATLAB 6.5. A later release of the package may choose to target the release 14 series.

2 Exponential integrators

Exponential integrators are numerical schemes specifically designed for solving differential equations where it is possible to split the problem into a linear and a nonlinear part

$$\dot{y} = Ly + N(y,t), \qquad y(t_{n-1}) = y_{n-1}, \tag{1}$$

where $y \in \mathbf{C}^d$, $L \in \mathbf{C}^{d\times d}$ and $N\colon \mathbf{C}^d \times \mathbf{R} \to \mathbf{C}^d$. An overview of the history of exponential integrators is given in [22]. Typically in the applications of interest (discretizations of PDEs) the matrix L is unbounded. Generally solving such problems requires an implicit scheme; the aim of exponential integrators is to treat the linear term exactly and allow the remaining part of the integration to integrated numerically using an explicit scheme.

An exponential integrator has two main features:

1. If $L = 0$, then the scheme reduces to a standard general linear scheme. This is often called the underlying general linear scheme.

2. If $N(y,t) = 0$ for all y and t, then the scheme reproduces the exact solution of (1).

To ensure the second point the exponential function must be used within the numerical scheme. Despite the fact that L is unbounded, typically the coefficients of the scheme will be bounded. Generally it is not possible to obtain efficient schemes by just using the exponential function within the scheme, functions closely related to the exponential, known as φ-functions, are also needed.

The way exponential integrators are implemented in this MATLAB package allows for a relaxation of feature 2, in which some approximation of the exponential can be used. An example in this package is the Crank–Nicolson scheme. In the rest of this section we will first describe how to implement exponential integrators of the Runge–Kutta type. Then we will go directly to general linear schemes which generalises both Runge–Kutta and multistep schemes.

2.1 The format of an exponential Runge–Kutta scheme

For an s-stage exponential integrator of Runge–Kutta-type, we define the internal stages and output approximation as follows:

$$\begin{aligned} Y_i &= h\sum_{j=1}^{s} a_{ij}(hL)\, N(Y_j, t_{n-1} + c_j h) + u_{i1}(hL) y_{n-1}, \quad i = 1,\dots,s, \\ y_n &= h\sum_{i=1}^{s} b_i(hL) N(Y_j, t_{n-1} + c_j h) + v_1(hL) y_{n-1}. \end{aligned} \tag{2}$$

It is worthwhile to point out the lack of the collocation points c_i in the coefficient functions arguments, as opposed to the formats presented in [10, 31]. This allows for more generality within the schemes.

In order to fulfill feature 1 above, we require $u_{i1}(0) = 1$, $a_{ij}(0) = a_{ij}$, $v_1(0) = 1$, and $b_i(0) = b_i$, where the real numbers a_{ij} and b_i are the coefficients of the underlying Runge–Kutta scheme. The functions used in (2) are conveniently represented in an extended Butcher tableau

$$\begin{array}{c|ccc|c} c_1 & a_{11}(z) & \cdots & a_{1s}(z) & u_{11}(z) \\ \vdots & \vdots & & \vdots & \vdots \\ c_s & a_{s1}(z) & \cdots & a_{ss}(z) & u_{s1}(z) \\ \hline & b_1(z) & \cdots & b_s(z) & v_1(z) \end{array} \tag{3}$$

The MATLAB-functions defining an exponential integrator in the `schemes`-directory returns exactly these functions given in this tableau.

The extension from a traditional integrator to an exponential integrator is not unique. We give two well known examples of how the forward Euler scheme can be extended to the exponential setting. The first scheme is most commonly known as Lawson–Euler

$$y_n = \exp(hL)y_{n-1} + \exp(hL)N(y_{n-1}, t_{n-1}), \qquad \begin{array}{c|c|c} 0 & 0 & 1 \\ \hline & \mathrm{e}^z & \mathrm{e}^z \end{array} \tag{4}$$

The second scheme is Nørsett–Euler

$$y_n = \exp(hL)y_{n-1} + \varphi_1(hL)N(y_{n-1}, t_{n-1}), \qquad \begin{array}{c|c|c} 0 & 0 & 1 \\ \hline & \varphi_1(z) & \mathrm{e}^z \end{array} \tag{5}$$

The later scheme has been reinvented several times and is also known as the ETD Euler, filtered Euler, Lie–Euler and exponentially fitted Euler. Source code for the Nørsett–Euler is given in Listing 2. The function φ_1 will be defined later.

2.2 Extension to general linear schemes

We explain the notation necessary for constructing exponential general linear schemes, which were introduced in [22]. The notation follows very closely with that developed for the standard schemes in [3].

To perform a step of length h in an exponential general linear scheme, we need to import r approximations into the step, denoted as $y_i^{[n-1]}$, $i = 1, \ldots, r$. These quantities can be of a very general nature. The internal stages (as in the Runge–Kutta case) are written as Y_i, $i = 1, \ldots, s$. After the step is completed, r updated approximations are computed. These are then used in the next step.

Each step in an exponential general linear scheme can be written as

$$\begin{aligned} Y_i &= \sum_{j=1}^{s} a_{ij}(hL)hN(Y_j, t_{n-1} + c_j h) + \sum_{j=1}^{r} u_{ij}(hL)y_j^{[n-1]}, \qquad i = 1, \ldots, s, \\ y_i^{[n]} &= \sum_{j=1}^{s} b_{ij}(hL)hN(Y_j, t_{n-1} + c_j h) + \sum_{j=1}^{r} v_{ij}(hL)y_j^{[n-1]}, \qquad i = 1, \ldots, r. \end{aligned} \tag{6}$$

The coefficient functions are grouped into matrices,

$$\begin{array}{c|ccc|ccc} c_1 & a_{11}(z) & \cdots & a_{1s}(z) & u_{11}(z) & \cdots & u_{1r}(z) \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ c_s & a_{s1}(z) & \cdots & a_{ss}(z) & u_{s1}(z) & \cdots & u_{sr}(z) \\ \hline & b_{11}(z) & \cdots & b_{1s}(z) & v_{11}(z) & \cdots & v_{1r}(z) \\ & \vdots & & \vdots & \vdots & & \vdots \\ & b_{r1}(z) & \cdots & b_{rs}(z) & v_{r1}(z) & \cdots & v_{rr}(z) \end{array}. \tag{7}$$

Exponential integrators of Runge–Kutta-type are easily seen to be a special case when $r = 1$ with $u_{i1}(z) = a_{i0}(z)$, $v_{11}(z) = b_0(z)$ and $b_{1j}(z) = b_j(z)$.

The way exponential general linear schemes are implemented in this package assumes a special structure of the vector $y^{[n-1]}$, the quantities which are passed from step to step

$$y^{[n-1]} = \left[\begin{array}{cccc} y_{n-1} & hN_{n-2} & hN_{n-3} & \cdots & hN_{n-r} \end{array} \right]^T,$$

where $N_{n-i} = N(y_{n-i}, t_{n-i})$. This choice enables both the ETD Adams–Bashforth [2, 8, 26] and generalized Lawson schemes [19, 28] to be conveniently represented in a single framework. Restricting ourselves to schemes of this form removes the need for complicated starting procedures. Rather an exponential Runge–Kutta scheme can be used for the first $r - 1$ steps, recording the approximation to the nonlinear term. To our knowledge no explicit exponential integrators exist which cannot be represented in this framework. If integrators are developed with, for example, a Nordsieck vector being passed from step to step it is not difficult to change the source code to allow for this option. In the supplied MATLAB-code, the scheme `hochost4` is used for the first $r - 1$ steps, but this is easily changed in the file `integrator.m`. In this paper we will use various terms of order. First the observed order is the order the numerical scheme achieves in a certain numerical example. Secondly the non-stiff order refers to the case when the operator L is bounded, such conditions were derived in [1, 22]. Finally, the stiff order refers to the case when L is unbounded, see [5, 16, 28] for various schemes. We note here that the stiff order convergence analysis is for parabolic problems only.

2.3 Construction of exponential integrators

This paper is not about how to construct new exponential integrators, but we will mention here a few of the main pathways taken in constructing such schemes. The paper [1] has some results and thoughts on how to construct exponential integrators of Runge–Kutta type.

2.3.1 Lawson schemes

The first class of exponential integrators we will consider are known after there inventor Lawson [20]. The aim of the Lawson schemes is to use a change of variables $v(t) =$

$e^{(t-t_{n-1})L}y(t)$. Then a modified differential equation is found in the v variable. This modified differential equation is then integrated using a standard explicit solver. The result is then back transformed to provide an approximation in the y variable. The overall scheme in the original variables was pointed out in [20]. These schemes unfortunately have stiff order one for all schemes.

2.3.2 ETD schemes

These are the most common class of exponential integrators and they date back to 1960 [7]. ETD schemes which are based on Adams–Bashforth schemes were first discovered by Nørsett [26]. ETD schemes based on Runge–Kutta schemes were independently discovered by several authors, we cite the papers [10, 30, 33]. The main idea behind the ETD schemes is to approximate the nonlinear term in the variation of constants formula by a suitable algebraic polynomial. The ETD Adams–Bashforth schemes obtain the same stiff order as the classical order, whereas the ETD Runge–Kutta schemes are required to satisfy complicated stiff order conditions. These were recently constructed up to order four in [16].

2.3.3 Affine Lie group schemes

Munthe-Kaas [25] pointed out that the affine Lie group could be used to solve semi-linear problems using Lie group schemes. Unfortunately the RKMK schemes were shown to exhibit instabilities due to the use of commutators. The commutator free schemes [6] overcome this problem but the overall stiff order achieved is limited to two.

2.3.4 Generalized Lawson schemes

Krogstad [19] proposed a more detailed transformation than that of Lawson. The resulting schemes are exponential general linear schemes, which achieve stiff order $r + 1$ for a transformation of order r. These schemes and modifications are discussed in detail in [28].

3 Installation and quick-start

This section describes how you can install the EXPINT package and be performing numerical experiments as soon as possible.

3.1 Installation

Fetch the archived file

`http://www.math.ntnu.no/num/expint/matlab/expint.tar.gz`

and unpack the file somewhere on your computer. In a UNIX-shell you would typically write

Listing 1: An example script to do a local error computation with plotting.

```
% Define problem to be solved:
problem = nls('ND', 256, 'IC', 'smooth', 'Potential', 'zero', 'lambda', 1);

% We should make a suitable vector of timesteps. The smallest timestep
% must divide all other timesteps (if not, the computation would be
% unnecessarily slower, so it is not supported)
steplist = unique(round(10.^(3:-0.1:0)));
dt = 10^-3 .* steplist;

% Choose schemes to be used:
schemes = setupschemes('etd4rk', 'lawson4', 'genlawson43');

% Do computational test with lawson4 as a reference solver (with
% smaller timestep):
schemes = localorder(problem, dt, schemes, 'lawson4');

% Make plot:
orderplot(schemes, 'Timestep_h', 'Local_error', problem.problemname);
```

Listing 2: MATLAB-code for the Nørsett Euler scheme

```
function [u, v, a, b, c] = norsetteuler(z, problem)
% NØRSETTEULER - The Nørsett Euler method in unified method format.

[one, ez2, ez, z] = oneez2(z); % oneez2 is a utility function in 'common'.

phi_1 = phipade(z, 1); % Compute phi_1 using a Pade-approximant.

v = { ez };
u = { one };
b = { phi_1 };
a = { [] };
c = [0];
```

```
$ tar zxvf expint.tar.gz
```

This will unpack the archive and will give you a directory named `expint` containing the directories `problems`, `schemes`, `common` and `tests` and the file `startup.m`.

3.2 A quick test, local error plot

In the main directory of the package, that is in the directory where the mentioned directories are present, launch MATLAB from the command line. This ensures that the `startup.m` file is being read. The commands in Listing 1 may be typed directly in, or run via some script (`m`-file). For the moment we regard this example code to be self-explanatory, but details will be elaborated on later.

3.3 Directory structure

`problems` Each equation or problem you wish to solve needs to be defined by a "problem"-structure. Files in this directory are helper functions for generating this structure (a MATLAB-structure). Each problem is really a system of ODEs, so typically the semi-discretization is what is provided by the "problem" when solving PDEs. The problem-structure is dealt with in Section 4.

`schemes` Files in this directory represents a specific scheme or integrator in the format for exponential integrators which were introduced in Section 2.

`common` All kinds of utility functions are put in here. Important functions are described in Section 6.

`tests` In here we have put example scripts to facilitate testing, you could also put your own scripts here if you want to.

3.4 Choosing schemes to test

It is a typical situation to test many schemes at once in order to do comparisons. For this, one defines an array of structures describing the schemes to be used. Let `schemes` be the array, each element is a structure describing one scheme. The structure elements are

`coffunc` A string naming a MATLAB-function which is able to return the coefficient functions in the format of (3) or (7).

`name` A describing name for the scheme in question. If possible, it is advisable to have this field and the `coffunc`-field equal. This field is used for legends in plots.

`linestyle` This is a string that will be used by the plotting routines to determine the line style for the scheme in question, in accordance with the MATLAB plot function.

A function called `setupschemes` is added for convenience in order to easily build up this structure, where you only list the names of the schemes, and the structure will be returned. See Listing 1 for an example. In addition, the scripts `localorder` and `globalorder` add computational results to this structure for use within the script `orderplot`.

An optional field in the scheme structure is

`relstages` A positive integer which can be used as a weight. If `relstages` $= 4$ for a scheme, the actual timestep used by `integrator` will be one fourth of your supplied timestep. This could be useful in some circumstances for producing fairer comparisons.

4 Defining equations to solve

The approach taken to tell MATLAB about the current differential equation to solve is done using a data structure, which must be defined first in every computation. Assembling this structure is best done in its own function-file, and it will also contain references to other functions representing the problem or equation in question.

4.1 The problem structure

The following fields are mandatory for defining a problem representing a differential equation.

`ND` A discretization parameter. It is up to the problem to interpret this parameter in a suitable way. It could be the number of Fourier modes or the number of inner points in a finite difference discretization.

`y0` The initial condition for the corresponding system of ODEs. A standing vector.

`L` The matrix L in (1).

`N` A string naming a MATLAB-function implementing the function $N(y,t)$ in (1). The calling syntax is assumed to be `N(y, t, problem)` where `y` is the phase value, `t` is the time point, and `problem` is the entire problem structure.

`problemname` A textual description of the problem.

In addition, we have found it useful to include these fields as well, but these should be treated as optional by all code:

`postprocessing` Function to be used for post-processing the data, for example inverse fourier transform if the problem is spectrally discretized. If the problem is of multidimensional nature, it is appropriate to do a conversion from vector to matrix form of the data more suited for plotting.

`LplusN` A string naming a MATLAB-function implementing $Ly + N(y,t)$. This is needed in order to be able to use MATLAB's `ode15s` as a reference solver.

4.2 Included problems

In the `problems`-directory a bundle of example problems is included. The included problems serve mainly as examples on how to implement flexible problem structures. Beyond returning a structure fulfilling the requirements of Section 4.1, the actual implementation is entirely up to the user. The name of the problem-file in the directory `problems` is included in the title in a `typed` font.

4.2.1 The 1D nonlinear Schrödinger equation — `nls.m`

This problem was introduced in Listing 1 and takes the form

$$\mathrm{i}y_t = -y_{xx} + (V(x) + \lambda|y|^2)y, \qquad x \in [-\pi, \pi].$$

with some initial condition and with periodic boundary conditions. The problem file includes the (spectral) semi-discretization of the problem. Upon initialising of the problem in MATLAB, different choices of the potential, the initial condition and λ may be chosen. A typical example of a calling sequence is the one in Listing 1,

```
problem = nls('ND', 64, 'IC', 'smooth', 'Potential', 'zero', 'lambda', 1);
```

The way arguments are supplied to the problem functions is by no means mandatory for this package, it is merely for convenience of the user. What exactly is meant by a "*smooth*" initial condition as in this example is easily found in the source code, the user is encouraged to make changes for him/herself.

4.2.2 The 1D KdV–equation — `kdv.m`

The Korteweg–de Vries equation with periodic boundary conditions in 1D is

$$y_t = -y_{xxx} - yy_x, \qquad x \in [-\pi, \pi].$$

This is semi-discretized spectrally and integrated from $t = 0$ to $t = 2\pi/625$ by default, the linear part is the diagonal matrix with entries $L_{kk} = \mathrm{i}k^3$, $k = -\mathtt{ND}/2 + 1, \ldots, \mathtt{ND}/2$ and the nonlinear function is

$$N(y(k), t) = -\frac{\mathrm{i}}{2}k\mathcal{F}(\mathcal{F}^{-1}(y)^2).$$

The default value of `ND` is 128. The eigenvalues of L are complex and therefore, the problem exhibits rapid oscillations for high wave number modes. Various initial conditions are supported, a well known choice, taken from [19] is

$$y(x, 0) = 3\lambda \operatorname{sech}^2\left(\sqrt{\lambda}x/2\right).$$

4.2.3 The Kuramoto–Sivashinsky equation — `kursiv.m`

The Kuramoto–Sivashinsky equation has been used to study many reaction-diffusion systems, in 1D it is written as

$$y_t = -y_{xx} - y_{xxxx} - yy_x, \qquad x \in [0, 32\pi].$$

We use a spectral discretization with periodic boundary conditions and `ND` equal to 128. The problem is integrated from $t = 0$ to $t = 65$ by default. The linear term is diagonal with elements $L_{kk} = k^2 - k^4$, $k = -\mathtt{ND}/2 + 1, \ldots, \mathtt{ND}/2$, which results in rapid decay for high wave numbers. The nonlinear function is

$$N(y(k), t) = -\frac{\mathrm{i}}{2}k\mathcal{F}(\mathcal{F}^{-1}(y)^2).$$

Various choices of initial condition are supported, the choice of smooth initial condition is taken from [18]

$$y(x,0) = \cos(x/16)\,(1+\sin(x/16))\,.$$

4.2.4 Burgers' equation — `burgers.m`

This equation, dating back to 1915, has been used in the study of turbulence and shock formation, it reads

$$y_t = \lambda y_{xx} - \frac{1}{2}(y^2)_x, \qquad x \in [-\pi, \pi].$$

We use a spectral discretisation with `ND` equal to 128 as default spatial resolution and various choices of initial condition are supported. The most commonly used for this equation, see [18], is

$$y(x,0) = \exp(-10\sin^2(x/2)).$$

The rapid oscillations apparent in this problem come from the λy_{xx} term, where $\lambda = 0.03$ is the default value.

4.2.5 The Hochbruck–Ostermann equation — `hochost.m`

A semi-linear parabolic problem with homogeneous Dirichlet boundary conditions from [16] is

$$y_t = y_{xx} + \frac{1}{1+y^2} + \Phi, \qquad x \in [0,1],$$

where Φ is chosen so that the exact solution is $y(x,t) = x(1-x)\mathrm{e}^t$. The problem is discretized in space using a standard finite difference scheme, with `ND` by default set to 200. The resulting ODE is integrated from $t = 0$ to $t = 1$, with various initial conditions supported. This problem results in a reduction in order for almost all schemes.

4.2.6 The Allen–Cahn equation — `allencahn.m`

The Allen–Cahn equation a parabolic problem, which reads

$$\begin{aligned} y_t &= \lambda y_{xx} + y - y^3, \qquad x \in [-1,1], \\ y(x,0) &= 0.53x + 0.47\sin(-1.5\pi x), \end{aligned}$$

we choose to implement this equation with the Dirichlet boundary conditions $y(-1,t) = -1$ and $y(1,t) = 1$. The linear part λy_{xx} is discretized using a Chebyshev differentiation matrix, resulting in a full matrix L. Further details may be found in [18]. In order to deal with the boundary conditions, one defines $y = w + x$ and works with the variable w which has homogeneous boundary values. The default parameter values are `ND` $= 64$ and $\lambda = 0.001$.

4.2.7 The 2D complex Ginzburg–Landau equations — `cginzland2d.m`

This equation describes reaction-diffusion equations close to a Hopf bifurcation and generates spiral wave fronts

$$y_t = y + (1 + \mathrm{i}\alpha)\nabla^2 y - (1 + \mathrm{i}\beta)y|y|^2, \qquad x, y \in [0, 200].$$

Both smooth (a series of Gaussian pulses) and random initial conditions are supported, see [17] or the source code for more details. We implement this equation using a Fourier spectral discretisation in 2D, this is

$$\widehat{y}_t = (1 - (1 + \mathrm{i}\alpha)(k^2 + l^2))\widehat{y} - \mathcal{F}((1 + \mathrm{i}\beta)y|y|^2).$$

By default we use `ND` equal to 128 and integrate from $t = 0$ to $t = 150$, with bifurcation parameters $\alpha = 0$ and $\beta = 1.3$.

4.2.8 The Gray–Scott equations in 2D — `grayscott2d.m`

The Gray–Scott equation is a reaction-diffusion equation which exhibits a wide variety of interesting patterns. In non-dimensional form the system is

$$u_t = D_u\nabla^2 u - uv^2 + \alpha(1 - u),$$
$$v_t = D_v\nabla^2 v + uv^2 - (\alpha + \beta)v,$$

where the positive diffusion parameters D_u, D_v are generally chosen so that the ratio $D_u/D_v = 2$. The default choice is $D_u = 2 \cdot 10^{-5}$ and $D_v = 1 \cdot 10^{-5}$. The constants α and β can be viewed as bifurcation parameters. As a default choice we set $\alpha = 0.065$ and $\beta = 0.035$. Defining $y = [u, v]^T$, represents the equations in the appropriate form. Note that the transformed equations in Fourier space are very similar to the original equations, they read

$$\widehat{u}_t = -D_u(k^2 + l^2)\widehat{u}(l, k) - \mathcal{F}(uv^2 - \alpha(1 - u)),$$
$$\widehat{v}_t = -D_v(k^2 + l^2)\widehat{v}(l, k) + \mathcal{F}(uv^2 - (\alpha + \beta)v).$$

The initial conditions we choose can be found in the source code, the smooth initial condition we have implemented are scaled Gaussian pulses. By default `ND` is chosen to be 128 on the grid $x, y \in [0, 2.5]$. The package also includes discretizations of this problem in one and three space dimensions. We refer the reader to the source code in `grayscott.m` and `grayscott3d.m` respectively for more details. In the directory `tests` you will find files `testgrayscott2d.m` and `testgrayscott3d.m`, which can be viewed as typical test scripts.

5 φ-functions

Central to the implementation of exponential integrators is the evaluation of exponential-like functions, commonly denoted by φ-functions in recent literature. We define the functions by the integral representation

$$\varphi_\ell(z) = \frac{1}{(\ell-1)!}\int_0^1 \mathrm{e}^{(\theta-1)z}\theta^{\ell-1}\,\mathrm{d}\theta, \qquad \ell = 1, 2, \ldots. \tag{8}$$

For small values of ℓ, these are

$$\varphi_1(z) = \frac{\mathrm{e}^z - 1}{z}, \quad \varphi_2(z) = \frac{\mathrm{e}^z - z - 1}{z^2}, \quad \varphi_3(z) = \frac{\mathrm{e}^z - z^2/2 - z - 1}{z^3}.$$

It is convenient to define $\varphi_0(z) = \mathrm{e}^z$, in which case the functions obey the recurrence relation

$$\varphi_\ell(z) = \frac{\varphi_{\ell-1}(z) - \frac{1}{(\ell-1)!}}{z^\ell}, \qquad \ell = 1, 2, \ldots.$$

Evaluating these expressions numerically is not a trivial task. The accurate computation of φ_1 is a well known problem in numerical literature see [13, 15, 21], and problems arise from small values of z giving cancellation errors. For small values of z, expanding in Taylor series leads to accurate approximations, while for large values of z, the direct formula can be used. This approach was used by Cox and Matthews [8], but they noticed in some cases that in an interval close to the switching point, neither approximation was accurate.

Kassam and Trefethen [18] proposed to approximate with a contour integral, which worked well as long as the contour of integration is suitably chosen. However, the contour is in general problem dependent and difficult to determine in advance.

We propose to use an algorithm based on first a variant of the scaling and squaring approach (method 3 in [23]). After scaling, the exponential is calculated using a Padé approximant. This is the same approach used to compute φ_1 in [15] and the accompanying C and MATLAB-software. The general form of the (d, d)-Padé approximant of φ_ℓ is

$$\varphi_\ell(z) = \frac{N_d^\ell(z)}{D_d^\ell(z)} + \mathcal{O}(z^{2d+1}),$$

where the unique polynomials N_d^ℓ and D_d^ℓ are

$$\begin{aligned} N_d^\ell(z) &= \frac{d!}{(2d+\ell)!}\sum_{i=0}^{d}\left[\sum_{j=0}^{i}\frac{(2d+\ell-j)!(-1)^j}{j!(d-j)!(\ell+i-j)!}\right]z^i, \\ D_d^\ell(z) &= \frac{d!}{(2d+\ell)!}\sum_{i=0}^{d}\frac{(2d+\ell-i)!}{i!(d-i)!}(-z)^i. \end{aligned} \tag{9}$$

When $\ell = 0$, these reduce to the well known diagonal Padé approximations of the exponential function, see for example [4].

To reverse the scaling for $\ell > 0$ is not trivial, in [2] a list of scaling relations for $\ell = 1, \ldots, 6$, is given. In general one can find that the φ-functions obey scaling relations

of the form

$$\begin{aligned}\varphi_{2\ell}(2z) &= \frac{1}{2^{2\ell}}\left[\varphi_\ell(z)\varphi_\ell(z) + \sum_{j=\ell+1}^{2\ell}\frac{2}{(2\ell-j)!}\varphi_j(z)\right], \\ \varphi_{2\ell+1}(2z) &= \frac{1}{2^{2\ell+1}}\left[\varphi_\ell(z)\varphi_{\ell+1}(z) + \sum_{j=\ell+2}^{2\ell+1}\frac{2}{(2\ell+1-j)!}\varphi_j(z) + \frac{1}{\ell!}\varphi_{\ell+1}(z)\right].\end{aligned} \tag{10}$$

The overall algorithm for computing φ-functions may be summarized as follows.

1. Let p be the smallest integer such that $2^p \geq \|z\|_\infty$.
2. Set $z_{\text{scaled}} = z/2^{\max(0,p+1)}$.
3. Compute $\varphi_\ell(z_{\text{scaled}})$ using a (6,6)-Padé approximant from (9).
4. Undo scaling by (10).

In step 3, we choose the (6,6)-Padé approximation, we choose this case because it was promoted in [15] for φ_1. Note that the evaluation of φ_ℓ by scaling and modified squaring requires φ_j, for $j < \ell$, so the function implemented in this package will only take in ranges of φ-functions to compute, starting with φ_1. A detailed analysis of the φ-functions is a topic we intend to pursue in the near future.

6 Included functions and scripts

We discuss here selected functions included in the package, and their usage. The header documentation in each script is usually more in-depth than present here, but may be of a more technical nature.

6.1 `integrator.m`

The basic stepper routine for exponential integrators.

[t, u, varargout] = **integrator**(problem, tspan, h, cof)
[t, u, varargout] = **integrator**(problem, tspan, h, cof, timepoints);

This is the key interface function in this package. It is the function that integrates the differential equation (or the "problem") throughout a timespan. If an addition vector of timepoints is supplied, the numerical solution at those points will be returned. Each element in `timepoints` must be an integer multiple of `h`. `integrator.m` has a callback functionality in case one wants to calculate a function, or plot some function (energy) of the solution as `integrator` integrates through time. This uses the field `outputfcn` in the problem-structure, Section 4.1. The `nls`-problem contains two examples on usage of this callback functionality.

6.2 `phipade.m`

Computes approximations to φ-functions.

```
phi_k = phipade(z, k);
[phi_1, phi_2, ..., phi_k] = phipade(z, k);
```

This is a function crucial to all schemes using φ-functions. It implements the algorithm sketched in Section 5. For efficiency, especially in order-tests, additional memory resources are expended in caching recently computed function values. The function value cache is organized in an LRU (least recently used) fashion and, unfortunately, increases the code complexity of `phipade`. Caching is turned on by default, but may be toggled with the use of the `wantcache`-function or by setting a variable `wantcache` within `phipade.m`. The function is entirely self-contained with respect to use of other functions. It may be pulled out of the EXPINT-package and used independently if necessary.

6.3 `globalorder.m`

Performs a global order test.

```
schemes = globalorder(problem, tspan, dt, schemes, varargin)
```

Input arguments are the problem in question, the timespan, a vector of timesteps to test and the schemes to test. Optionally, one may provide a string with a scheme-name which will be used for the reference solution. The default is to use `ode15s`. It is important to build up a sensible vector of timesteps to be used, *all timesteps used should divide the length of the timespan so that all stepsizes are constant throughout each test.* An example is

```
tspan = [0, 1];
steplist = round(10.^(4:-0.1:0));
dt = unique(diff(tspan)./steplist);
```

The function returns the cell-structure of schemes, but with error-data added (and also timings, measured using `cputime`). This data is used by `orderplot` and `timingplot`.

6.4 `localorder.m`

Performs a local order test.

```
schemes = localorder(problem, dt, schemes, varargin)
```

This does not make any sense for multistep-schemes. Arguments are as for `globalorder`, but now the vector of timesteps must be built up differently. *The smallest timestep must divide all other timesteps used*, or else the computation would take an unnecessarily long time. An example is

```
steplist = round(10.^(4:-0.1:1));
dt = unique(10^-4 .* steplist);
```

Error and timing data is added as for `globalorder`.

6.5 `orderplot.m`

Makes a plot of experimental order.

varargout = **orderplot**(schemes)
varargout = **orderplot**(schemes, thexlabel, theylabel, thetitle)

This function produces a plot of local or global error versus timestep. Both error and timesteps are read from the `schemes`-array.

6.6 `timingplot.m`

Makes a plot of timing results.

varargout = **timingplot**(schemes)
varargout = **timingplot**(schemes, thexlabel, theylabel, thetitle)

This function works almost the same as `orderplot.m`, although it plots (global) error as a function of time spent for each time step and for each scheme.

6.7 `orderline.m`

Plots a dotted straight line to show order.

orderline(**gradient**)
orderline(**gradient**, [h, err])

This is a small utility function for plotting a dotted straight line in a loglog-plot along with a number representing the gradient of the line. This is useful to add in an orderplot to visualize the observered order of the schemes. With only one argument, the function is interactive, waiting for the user to click with the mouse in a plot specifying one point the line is to go through. The function prints out a vector that can be used as `[h, err]` the next time the function is used. The function is non-interactive when `[h, err]` is supplied.

6.8 `surfplot.m`

Surface plot of numerical solution.

varargout = **surfplot**(data, timepoints, problem, varargin)

Produces a surface plot of the numerical solution. In this case, one should use the "processed" solution of the problem if applicable, see `integrator`, Section 6.1. A typical usage could be:

```
tspan = [0 1]; h = 0.1; timepoints = [tspan(1):h:tspan(end)];
[t, U, UF] = integrator(problem, tspan, h, 'lawson4', timepoints);
surfplot(UF, timepoints, problem);
```

6.9 `randdecayfcn.m`

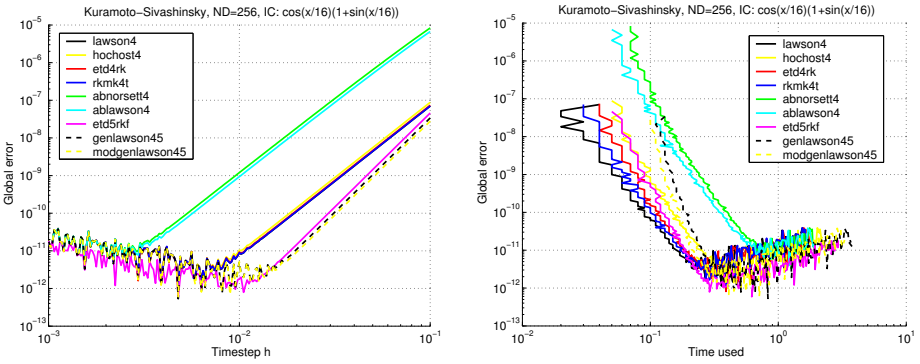
This is a function used internally in some problem-files. It produces a vector resembling a discrete Fourier transform of a random function with a prescribed regularity. The function is used for example in `nls.m` to be able to choose arbitrary regularity for the initial condition and the potential.

7 Illustrative examples

In this section we show some typical uses of the EXPINT package.

7.1 Global order with timing plot

The global order test provides a simple way of producing plots which can be used for comparing the quality of certain schemes on a particular problem. We show here one representative example of a global error plot, that is the Kuramoto–Sivashinsky equation in 1D. Both global error as a function of timestep and error as a function of time spend are shown in Figure 1.



(a) Global error

(b) Error as a function of computational time

Figure 1: Example testrun for the Kuramoto–Sivashinsky equation. Code used is shown in Listing 3

7.2 Surface plot

The following sequence of commands generates the plot in Figure 2.

```
problem = kursiv('ND', 256);
[t, U, UF] = integrator(problem, [0 100], 0.1, 'lawson4',  [0:0.2:100]);
surfplot(UF, t, problem);
```

Here we use the third optional output argument from `integrator`, which returnes post-processed output determined by the problem, in this case for the Kuramoto–Sivashinsky

Listing 3: Code used to reproduce Figure 1

```
problem = kursiv('ND', 256);
timespan = [0, 1];
 steplist  = unique(round(10.^(3:-0.01:1)));
dt = diff(timespan)./ steplist ;

% Set up choice of schemes and placeholder for results:
M = setupschemes('lawson4', 'hochost4', 'etd4rk', 'rkmk4t', ...
                 'abnorsett4', 'ablawson4', 'etd5rkf', ...
                 'genlawson45', 'modgenlawson45' );

% Do the global order test. The M structure is returned augmented.
M = globalorder(problem, timespan, dt, M);

% Plot the results
orderplot(M, 'Timestep_h', 'Global_error', problem.problemname);
timingplot(M, 'Time_used', 'Global_error', problem.problemname);
```

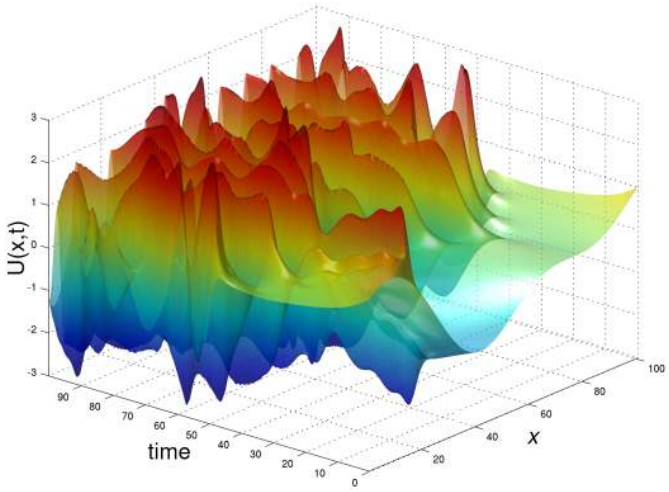


Figure 2: Example surface plot, Kuramoto–Sivashinsky 1D, $ND = 256$, Initial condition: $\cos(x/16)(1 + \sin(x/16))$.

equation it does the inverse fourier transform and picks out only the real part. We also specifically instruct `integrator` to return the solution values at specific points.

A similar procedure produces the plot in Figure 3. However, as this procedure requires a bit of extra manipulation to generate the desired graphic effects, we refer the reader to the `testgrayscott3d` script implemented in `tests/testgrayscott3d.m` for additional detail.

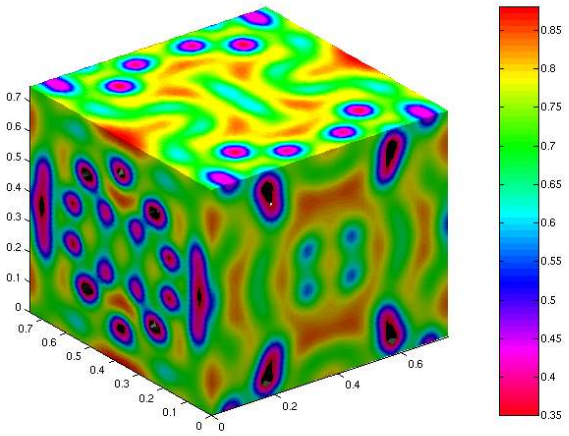


Figure 3: Example 3D plot, Gray–Scott, $ND = 48$.

8 Discussion

The EXPINT package was designed as a tool for facilitating easy testing and comparison of exponential integrators (and approximations to them) applied to semi-linear problems. The function which performs the integration known as `integrator.m` implements the exponential general linear method described in Section 2. This generality enables all known exponential integrators to be included in our framework, which allows the user to easily add their own integrators for comparison. This generality comes to some extent at the cost of efficiency, however we believe this to be justified given our aims of providing usability. The package includes many well known exponential integrators (and approximations to them) and some not so well known ones. Also several semi-discretizations of well known PDEs have been included. We encourage the user to add their own integrators and problems.

An important part of the EXPINT package is the evaluation of the φ-functions. There are various problems associated with the alternative approaches used to evaluate these functions. Combining scaling and squaring with Padé-approximations has become the standard approach in numerical software like MATLAB for computing the matrix exponential. In the self contained function `phipade.m` a scaling and corrected squaring combined

with diagonal Padé approximations enables efficient evaluation of the φ-functions. The Padé-coefficients in Section 5 have not been published elsewhere as far as we know.

The implementation uses fixed stepsizes, the reason being twofold. First the coefficients of the exponential integrator are functions of the stepsize h and the linear term L. Which means that if the stepsize is changed the coefficients of the method need to be recomputed before integrating over the new step. Secondly, semi-linear problems are generally integrated using low order methods such as Crank–Nicolson with fixed stepsizes. This enables us to compare exponential integrators with the standard approach for such problems. We do intend to extend this package so that variable stepsizes are possible, a way to overcome the higher computational cost arising from the change of the stepsize, is to take advantage of the fact that when implementing any exponential integrator, we do not really need to compute the φ-functions. What we need, is just their action to a given state vector v. Krylov subspace approximations to the exponential and some related functions, have been studied by many authors, see for example [12, 14, 29]. The main idea is to approximately project the action of the function $\varphi_\ell(hL)$ on a state vector v, to a smaller Krylov subspace. The computations are then performed in this subspace typically of much smaller dimension than the original problem. This technique has been used effectively for exponential integrators, we cite the papers [11, 15].

We hope that this package is in some way useful in testing and comparing exponential integrators and will encourage the use of these novel integrators in the future.

A Coefficient functions for included schemes

In this section we list all schemes implemented in this package, with reference to their origin. For space and æstethic reasons we adopt the notation

$$\varphi_{i,j} = \varphi_i(c_j z) \qquad i = 0, 1, \ldots, \text{ and } j = 1, \ldots, s \tag{11}$$

where $\varphi_0(z) = \mathrm{e}^z$. Also we use 1 to represent the identity matrix. All schemes with relevant figures describing their performance is listed in Table 1.

A.1 Lawson schemes

Lawson schemes are constructed by applying the Lawson transformation [20] to the semi-linear problem, then solving the transformed equation by a standard numerical scheme then back transforming. This whole process can be written in the original variables see [9], and results in the coefficients of the method involving exponentials. Below we include some Adam–Bashforth–Lawson and Runge–Kutta–Lawson methods schemes of low order.

A.1.1 Lawson–Euler — `lawsoneuler.m`

This is the simplest example of a Lawson scheme, which we choose to call Lawson–Euler. It has also occasionally been called the "exponential Euler scheme", but this is confusing

Name	Nonstiff p	Stiff p	Stages s	Output r	$\#\varphi$	matvecs
Lawson–Euler	1	1	1	1	1	1
ABLawson2	2	1	1	2	2	2
ABLawson3	3	1	1	3	3	3
ABLawson4	4	1	1	4	4	4
Lawson2a	2	1	2	1	2	3
Lawson2b	2	1	2	1	1	2
Lawson4	4	1	4	1	2	6
Nørsett–Euler	1	1	1	1	2	2
ABNørsett2	2	2	1	2	3	3
ABNørsett3	3	3	1	3	4	4
ABNørsett4	4	4	1	4	5	5
ETD2RK	2	2	2	1	3	4
ETD3RK	3	2	3	1	6	7
ETD4RK	4	2	4	1	6	10
Krogstad	4	3	4	1	7	11
Strehmel–Weiner	4	3	4	1	7	10
Friedli	4	3	4	1	7	11
Hochbruck–Ostermann	4	4	5	1	8	13
ETD5RKF	5	1	6	1	9	26
Ehle–Lawson	2	2	4	1	6	9
RKMK2e	2	2	2	1	2	3
ETD2CF3	3	2	3	1	9	8
Cfree4	4	2	4	1	4	9
RKMK4t	4	2	4	1	4	9
GenLawson41	4	2	4	1	4	8
GenLawson42	4	3	4	2	6	12
GenLawson43	4	4	4	3	8	16
GenLawson44	5	4	4	4	10	20
GenLawson45	6	5	4	5	12	24
ModGenLawson41	4	2	4	1	5	9
ModGenLawson42	4	3	4	2	7	13
ModGenLawson43	4	4	4	3	9	17
ModGenLawson44	5	5	4	4	11	21
ModGenLawson45	6	6	4	5	13	25

Table 1: Integrators included in the package, along with relevant figures describing their properties. $\#\varphi$ is the number of distinct φ-functions needed to be evaluated for each scheme. Beware that this counting of φ-functions and matrix-vector products do not tell the full story for all schemes.

given that the Nørsett–Euler scheme also goes by this name. This scheme has stiff order one.

$$\begin{array}{c|c|c} 0 & & 1 \\ \hline & \varphi_0 & \varphi_0 \end{array}$$

A.1.2 ABLawson2 — `ablawson2.m`

This scheme is based on the Adams–Bashforth scheme of order two, we represent the scheme in this form so that the incoming approximations are $y^{[n-1]} = [y_{n-1}, hN_{n-2}]$, in accordance with the starting procedure implemented in `integrator.m`. This scheme has stiff order one.

$$\begin{array}{c|cc|cc} 0 & & & 1 & 0 \\ 1 & \frac{3}{2}\varphi_0 & & \varphi_0 & -\frac{1}{2}\varphi_0^2 \\ \hline & \frac{3}{2}\varphi_0 & 0 & \varphi_0 & -\frac{1}{2}\varphi_0^2 \\ & 1 & 0 & 0 & 0 \end{array}$$

A.1.3 ABLawson3 — `ablawson3.m`

This scheme has stiff order one and is based on the Adams–Bashforth scheme of order three and is represented in this way so that the incoming approximation has the form $y^{[n-1]} = [y_{n-1}, hN_{n-2}, hN_{n-3}]^T$.

$$\begin{array}{c|cc|ccc} 0 & & & 1 & 0 & 0 \\ 1 & \frac{23}{12}\varphi_0 & & \varphi_0 & -\frac{4}{3}\varphi_0^2 & \frac{5}{12}\varphi_0^3 \\ \hline & \frac{23}{12}\varphi_0 & 0 & \varphi_0 & -\frac{4}{3}\varphi_0^2 & \frac{5}{12}\varphi_0^3 \\ & 1 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 1 & 0 \end{array}$$

A.1.4 ABLawson4 — `ablawson4.m`

This scheme has stiff order one and is based on the Adams–Bashforth scheme of order four and is represented in this way so that the incoming approximation has the form $y^{[n-1]} = [y_{n-1}, hN_{n-2}, hN_{n-3}, hN_{n-4}]^T$.

$$\begin{array}{c|cc|cccc} 0 & & & 1 & 0 & 0 & 0 \\ 1 & \frac{55}{12}\varphi_0 & & \varphi_0 & -\frac{59}{24}\varphi_0^2 & \frac{37}{24}\varphi_0^3 & -\frac{3}{8}\varphi_0^4 \\ \hline & \frac{55}{12}\varphi_0 & 0 & \varphi_0 & -\frac{59}{24}\varphi_0^2 & \frac{37}{24}\varphi_0^3 & -\frac{3}{8}\varphi_0^4 \\ & 1 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

A.1.5 Lawson2a — `lawson2a.m`

Based on the midpoint rule see [4, Eq. (232b)], this scheme has stiff order one.

$$\begin{array}{c|cc|c} 0 & & & 1 \\ \frac{1}{2} & \frac{1}{2}\varphi_{0,2} & & \varphi_{0,2} \\ \hline & 0 & \varphi_{0,2} & \varphi_0 \end{array}$$

A.1.6 Lawson2b — `lawson2b.m`

Based on the trapezoidal rule see [4, Eq. (232a)], this scheme has stiff order one.

$$\begin{array}{c|cc|c} 0 & & & 1 \\ 1 & \varphi_0 & & \varphi_0 \\ \hline & \frac{1}{2}\varphi_0 & \frac{1}{2} & \varphi_0 \end{array}$$

A.1.7 Lawson4 — `lawson4.m`

Based on the classical fourth order scheme of Kutta see [4, Eq. (235i)], this scheme has stiff order one.

$$\begin{array}{c|cccc|c} 0 & & & & & 1 \\ \frac{1}{2} & \frac{1}{2}\varphi_{0,2} & & & & \varphi_{0,2} \\ \frac{1}{2} & & \frac{1}{2} & & & \varphi_{0,2} \\ 1 & & & \varphi_{0,2} & & \varphi_0 \\ \hline & \frac{1}{6}\varphi_0 & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{6} & \varphi_0 \end{array} \tag{12}$$

A.2 ETD-schemes

ETD schemes are based on algebraic approximations to the nonlinear term in the variation of constants formula. ETD means "Exponential Time Differencing" and the name stems from [8]. This is not a particularly good name for this type of scheme as the name does not indicate anything about this type of schemes distinguishing it from other exponential integrators. Nevertheless, we still adopt the term for the time being.

Nørsett [26] developed a class of schemes which reduced to the Adams–Bashforth methods when the linear part of the problem is zero. The Adam–Bashforth–Nørsett schemes have recently been reinvented by Cox and Matthews [8] and Beylkin, Keiser and Vozovoi [2]. Many ETD schemes based on Runge–Kutta methods have also been developed, below we give some of the well known schemes.

A.2.1 Nørsett–Euler — `norsetteuler.m`

The most well known exponential version of the Euler method was first found by Nørsett in [26, 27]. It is also known as the exponentially fitted Euler, ETD Euler, ETD1RK,

filtered Euler scheme, or Lie–Euler as it is the simplest Lie group integrator with the affine Lie group, it has stiff order one.

0		1
	φ_1	φ_0

A.2.2 ABNørsett2 — `abnorsett2.m`

This stiff order two scheme of Nørsett [26], is implemented in this way so that the incoming approximation has the same form as in ABLawson2.

0			1	0
1	$\varphi_1 + \varphi_2$		φ_0	$-\varphi_2$
	$\varphi_1 + \varphi_2$	0	φ_0	$-\varphi_2$
	1	0	0	0

A.2.3 ABNørsett3 — `abnorsett3.m`

This stiff order three scheme of Nørsett [26], is implemented in this way so that the incoming approximation has the same form as in ABLawson3.

0			1	0	0
1	$\varphi_1 + \frac{3}{2}\varphi_2 + \varphi_3$		φ_0	$-2\varphi_2 - 2\varphi_3$	$\frac{1}{2}\varphi_2 + \varphi_3$
	$\varphi_1 + \frac{3}{2}\varphi_2 + \varphi_3$	0	φ_0	$-2\varphi_2 - 2\varphi_3$	$\frac{1}{2}\varphi_2 + \varphi_3$
	1	0	0	0	0
	0	0	0	1	0

A.2.4 ABNørsett4 — `abnorsett4.m`

This stiff order four scheme of Nørsett [26], is implemented in this way so that the incoming approximation has the same form as in ABLawson4.

0			1	0	0	0
1	$\varphi_1 + \frac{11}{6}\varphi_2 + 2\varphi_3 + \varphi_4$		φ_0	$-3\varphi_2 - 5\varphi_3 - 3\varphi_4$	$\frac{3}{2}\varphi_2 + 4\varphi_3 + 3\varphi_4$	$-\frac{1}{3}\varphi_2 - \varphi_3 - \varphi_4$
1	$\varphi_1 + \frac{11}{6}\varphi_2 + 2\varphi_3 + \varphi_4$	0	φ_0	$-3\varphi_2 - 5\varphi_3 - 3\varphi_4$	$\frac{3}{2}\varphi_2 + 4\varphi_3 + 3\varphi_4$	$-\frac{1}{3}\varphi_2 - \varphi_3 - \varphi_4$
	1	0	0	0	0	0
	0	0	0	1	0	0
	0	0	0	0	1	0

A.2.5 ETD2RK — `etd2rk.m`

This scheme first derived by Strehmel and Weiner [31, Eq. (3.6)], has roots in chemistry [24, Section 3] and was recently derived by Cox and Matthews [8, Eq. (22)], it has stiff order two.

$$
\begin{array}{c|cc|c}
0 & & & 1 \\
1 & \varphi_1 & & \varphi_0 \\
\hline
 & \varphi_1 - \varphi_2 & \varphi_2 & \varphi_0
\end{array}
$$

A.2.6 ETD3RK — `etd3rk.m`

This scheme was first derived by Friedli in [10, Section 4], and more recently appeared in [8, Eq. (23)–(25)], it has stiff order three.

$$
\begin{array}{c|ccc|c}
0 & & & & 1 \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} & & & \varphi_{0,2} \\
1 & -\varphi_1 & 2\varphi_1 & & \varphi_0 \\
\hline
 & \varphi_1 - 3\varphi_2 + 4\varphi_3 & 4\varphi_2 - 8\varphi_3 & -\varphi_2 + 4\varphi_3 & \varphi_0
\end{array}
$$

A.2.7 Ehle–Lawson — `ehlelawson.m`

This is a scheme of Ehle and Lawson [9] made to remedy problems with the Lawson schemes. It has four stages but only stiff order two.

$$
\begin{array}{c|cccc|c}
0 & & & & & 1 \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} & & & & \varphi_{0,2} \\
\frac{1}{2} & & \frac{1}{2}\varphi_{1,2} & & & \varphi_{0,2} \\
1 & & & \varphi_1 & & \varphi_0 \\
\hline
 & \varphi_1 - 3\varphi_2 + \varphi_3 & 2\varphi_2 - \varphi_3 & 2\varphi_2 - \varphi_3 & -\varphi_2 + \varphi_3 & \varphi_0
\end{array}
$$

A.2.8 ETD4RK — `etd4rk.m`

This scheme due to Cox and Matthews in [8, Eq. (26)–(29)], was one of the schemes that kick started the recent focus on exponential integrators, unfortunately it has only stiff order two.

$$
\begin{array}{c|cccc|c}
0 & & & & & 1 \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} & & & & \varphi_{0,2} \\
\frac{1}{2} & & \frac{1}{2}\varphi_{1,2} & & & \varphi_{0,2} \\
1 & \frac{1}{2}\varphi_{1,2}(\varphi_{0,2} - 1) & & \varphi_{1,2} & & \varphi_0 \\
\hline
 & \varphi_1 - 3\varphi_2 + 4\varphi_3 & 2\varphi_2 - 4\varphi_3 & 2\varphi_2 - 4\varphi_3 & -\varphi_2 + 4\varphi_3 & \varphi_0
\end{array}
$$

A.2.9 Krogstad — `krogstad.m`

This scheme appeared in [19, Eq. (51)] as a variant of ETD4RK by adding the φ_2-function in the internal stages, it does not require stage splittings and has stiff order three.

0					1
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2}$				$\varphi_{0,2}$
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2} - \varphi_{2,2}$	$\varphi_{2,2}$			$\varphi_{0,2}$
1	$\varphi_1 - 2\varphi_2$		$2\varphi_2$		φ_0
	$\varphi_1 - 3\varphi_2 + 4\varphi_3$	$2\varphi_2 - 4\varphi_3$	$2\varphi_2 - 4\varphi_3$	$-\varphi_2 + 4\varphi_3$	φ_0

A.2.10 Strehmel–Weiner — `strehmelweiner.m`

This scheme appeared in [32, Example 4.5.5] one of the earliest exponential Runge–Kutta methods, it has stiff order three.

0					1
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2}$				$\varphi_{0,2}$
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2} - \frac{1}{2}\varphi_{2,2}$	$\frac{1}{2}\varphi_{2,2}$			$\varphi_{0,2}$
1	$\varphi_1 - 2\varphi_2$	$-2\varphi_2$	$4\varphi_2$		φ_0
	$\varphi_1 - 3\varphi_2 + 4\varphi_3$	0	$4\varphi_2 - 8\varphi_3$	$-\varphi_2 + 4\varphi_3$	φ_0

A.2.11 Friedli — `friedli.m`

This scheme appeared in [10, Section 5], it is also one of the earliest exponential Runge–Kutta methods, it has stiff order three.

0					1
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2}$				$\varphi_{0,2}$
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2} - \frac{1}{2}\varphi_{2,2}$	$\frac{1}{2}\varphi_{2,2}$			$\varphi_{0,2}$
1	$\varphi_1 - 2\varphi_2$	$-\frac{26}{25}\varphi_1 + \frac{2}{25}\varphi_2$	$\frac{26}{25}\varphi_1 + \frac{48}{25}\varphi_2$		φ_0
	$\varphi_1 - 3\varphi_2 + 4\varphi_3$	0	$4\varphi_2 - 8\varphi_3$	$-\varphi_2 + 4\varphi_3$	φ_0

A.2.12 Hochbruck–Ostermann — `hochost4.m`

This scheme developed by Hochbruck and Ostermann [16, Section 5], with five-stages is the only known exponential Runge–Kutta method with stiff order four.

0						1
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2}$					$\varphi_{0,2}$
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2} - \varphi_{2,2}$	$\varphi_{2,2}$				$\varphi_{0,2}$
1	$\varphi_1 - 2\varphi_2$	φ_2	φ_2			φ_0
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2} - 2a_{5,2} - a_{5,4}$	$a_{5,2}$	$a_{5,2}$	$a_{5,4}$		$\varphi_{0,2}$
	$\varphi_1 - 3\varphi_2 + 4\varphi_3$	0	0	$-\varphi_2 + 4\varphi_3$	$4\varphi_2 - 8\varphi_3$	φ_0

where

$$a_{5,2} = \frac{1}{2}\varphi_{2,2} - \varphi_3 + \frac{1}{4}\varphi_2 - \frac{1}{2}\varphi_{3,2}, \qquad a_{5,4} = \frac{1}{4}\varphi_{2,2} - a_{5,2}$$

A.2.13 ETD5RKF — `etd5rkf.m`

This is a non-stiff fifth order scheme developed in [1]. It usually performs worse than other order four schemes presented here due to bad error constant. It is based on the six stage fifth order scheme of Fehlberg.

$$c = \begin{bmatrix} 0 & \frac{2}{9} & \frac{1}{3} & \frac{3}{4} & 1 & \frac{5}{6} \end{bmatrix}^T \qquad u_{i1}(z) = \varphi_{0,i}(z) \qquad v_{11}(z) = \varphi_0(z)$$

$$A(z) = \begin{bmatrix} -\frac{2}{3}\varphi_2 + \frac{10}{9}\hat{\varphi}_2 & & & & \\ \frac{569}{11544}\varphi_2 + \frac{1355}{11544}\hat{\varphi}_2 & -\frac{831}{3848}\varphi_2 + \frac{2755}{3848}\hat{\varphi}_2 & & & \\ -\frac{77157}{61568}\varphi_2 + \frac{143535}{61568}\hat{\varphi}_2 & \frac{587979}{61568}\varphi_2 - \frac{821745}{61568}\hat{\varphi}_2 & -\frac{405}{64}\varphi_2 + \frac{675}{64}\hat{\varphi}_2 & & \\ \frac{655263}{7696}\varphi_2 - \frac{2031205}{23088}\hat{\varphi}_2 & -\frac{1148769}{7696}\varphi_2 + \frac{1252665}{7696}\hat{\varphi}_2 & \frac{1593}{40}\varphi_2 - \frac{405}{8}\hat{\varphi}_2 & \frac{144}{5}\varphi_2 - \frac{80}{3}\hat{\varphi}_2 & \\ -\frac{2212835}{277056}\varphi_2 + \frac{6888625}{831168}\hat{\varphi}_2 & \frac{477285}{30784}\varphi_2 - \frac{496525}{30784}\hat{\varphi}_2 & -\frac{39}{16}\varphi_2 + \frac{65}{16}\hat{\varphi}_2 & -\frac{4}{9}\varphi_2 + \frac{20}{27}\hat{\varphi}_2 & -\frac{185}{96}\varphi_2 + \frac{575}{288}\hat{\varphi}_2 \end{bmatrix}$$

$$b_{11}(z) = \frac{47}{150}\varphi_1 - \frac{188}{75}\varphi_2 + \frac{94}{15}\varphi_3 \quad b_{12}(z) = 0$$
$$b_{13}(z) = -\frac{43}{25}\varphi_1 + \frac{132}{5}\varphi_2 - 66\varphi_3 \quad b_{14}(z) = \frac{4124}{75}\varphi_1 - \frac{6152}{15}\varphi_2 + \frac{2704}{3}\varphi_3$$
$$b_{15}(z) = \frac{189}{10}\varphi_1 - \frac{662}{5}\varphi_2 + 284\varphi_3 \quad b_{16}(z) = -\frac{1787}{25}\varphi_1 + \frac{12966}{25}\varphi_2 - \frac{5628}{5}\varphi_3$$

where $\hat{\varphi}_2(z) = \varphi_2(\frac{3}{5}z)$. Note that the coefficient in front of φ_3 is different from [1] due to differing definitions of φ_i.

A.3 Affine Lie group schemes

The construction of Lie group integrators for the solution of semi-discretized PDEs started with the paper of Munthe–Kaas [25], where the affine Lie group was used. The RKMK methods requires the computation of the dexp^{-1} operator which involves iterated commutators. To overcome the need for commutators which often result in stepsize restrictions Celledoni, Martinsen and Owren [6], constructed the commutator-free methods.

A.3.1 RKMK2e — `rkmk2e.m`

This scheme which generalizes the trapezoidal rule can be derived from [25, Ex. 4], but is also a standard scheme in the chemistry literature, known as the Pseudo-Steady-State-Approximation (PASSA) scheme, see [34, Section 2], it has stiff order one.

0			1
1	φ_1		φ_0
	$\frac{1}{2}\varphi_1$	$\frac{1}{2}\varphi_1$	φ_0

A.3.2 ETD2CF3 — `etd2cf3.m`

This is a stiff order three ETD version [16] of a commutator-free scheme in [6].

0				1
$\frac{1}{3}$	$\frac{1}{3}\varphi_{1,2}$			$\varphi_{0,2}$
$\frac{2}{3}$	$\frac{2}{3}\varphi_{1,3} - \frac{4}{3}\varphi_{2,3}$	$\frac{4}{3}\varphi_{2,3}$		$\varphi_{0,3}$
	$\varphi_1 - \frac{9}{2}\varphi_2 + 9\varphi_3$	$6\varphi_2 - 18\varphi_3$	$-\frac{3}{2}\varphi_2 + 9\varphi_3$	φ_0

A.3.3 Cfree4 — `cfree4.m`

This scheme given in [6, Eq. (7)] assuming affine Lie group action is used, is of stiff order two. Note that the internal stages are equivalent to those of ETD4RK, Section A.2.8.

0					1
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2}$				$\varphi_{0,2}$
$\frac{1}{2}$		$\frac{1}{2}\varphi_{1,2}$			$\varphi_{0,2}$
1	$\frac{1}{2}\varphi_{1,2}(\varphi_{0,2} - 1)$		$\varphi_{1,2}$		φ_0
	$\frac{1}{2}\varphi_1 - \frac{1}{3}\varphi_{1,2}$	$\frac{1}{3}\varphi_1$	$\frac{1}{3}\varphi_1$	$-\frac{1}{6}\varphi_1 + \frac{1}{3}\varphi_{1,2}$	φ_0

A.3.4 RKMK4t — `rkmk4t.m`

Using a suitable truncation of the dexp^{-1} operator leads to the method of Munthe–Kaas [25, Ex. 4], which again is of stiff order two but suffers from instabilities, especially when non-periodic boundary conditions are used.

0					1
$\frac{1}{2}$	$\frac{1}{2}\varphi_{1,2}$				$\varphi_{0,2}$
$\frac{1}{2}$	$\frac{z}{8}\varphi_{1,2}$	$\frac{1}{2}(1 - \frac{z}{4})\varphi_{1,2}$			$\varphi_{0,2}$
1			φ_1		φ_0
	$\frac{1}{6}\varphi_1(1 + \frac{z}{2})$	$\frac{1}{3}\varphi_1$	$\frac{1}{3}\varphi_1$	$\frac{1}{6}\varphi_1(1 - \frac{z}{2})$	φ_0

A.4 Generalized Lawson schemes

Krogstad [19], constructed these schemes as a means of overcoming some of the undesirable properties of the Lawson schemes. The methods boil down to using a more sophisticated transformation which better approximates the dynamics of the original differential equation see [28] for more details. The transformation involves using approximations of the nonlinear term from previous steps, resulting in an exponential general linear method.

Two schemes are included in the package but not listed here due to space reasons, that is the `genlawson44.m` and `genlawson45.m`.

A.4.1 GenLawson41 — `genlawson41.m`

This exponential Runge–Kutta scheme closely related to Lawson4 has stiff order two.

$$\begin{array}{c|cccc|c}
0 & & & & & 1 \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} & & & & \varphi_{0,2} \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} - \frac{1}{2} & \frac{1}{2} & & & \varphi_{0,2} \\
1 & \varphi_1 - \varphi_{0,2} & & \varphi_{0,2} & & \varphi_0 \\
\hline
 & \varphi_1 - \frac{2}{3}\varphi_{0,2} - \frac{1}{6} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{6} & \varphi_0
\end{array}$$

A.4.2 GenLawson42 — `genlawson42.m`

This scheme requires y_{n-1} and N_{n-2} as initial data. At least one step of an alternative method is needed to start the integration. The overall stiff order is three.

$$\begin{array}{c|cccc|cc}
0 & & & & & 1 & 0 \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} + \frac{1}{4}\varphi_{2,2} & & & & \varphi_{0,2} & -\frac{1}{4}\varphi_{2,2} \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} + \frac{1}{4}\varphi_{2,2} - \frac{3}{4} & \frac{1}{2} & & & \varphi_{0,2} & -\frac{1}{4}\varphi_{2,2} + \frac{1}{4} \\
1 & \varphi_1 + \varphi_2 - \frac{3}{2}\varphi_{0,2} & & \varphi_{0,2} & & \varphi_0 & -\varphi_2 + \frac{1}{2}\varphi_{0,2} \\
\hline
 & \varphi_1 + \varphi_2 - \varphi_{0,2} - \frac{1}{3} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{6} & \varphi_0 & -\varphi_2 + \frac{1}{3}\varphi_{0,2} + \frac{1}{6} \\
 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}$$

A.4.3 GenLawson43 — `genlawson43.m`

This scheme requires y_{n-1}, N_{n-2} and N_{n-3} as initial data. At least two steps of an alternative method are needed to start the integration. The overall stiff order is four and for æsthetic reasons the method is broken into the individual matrices.

$$c = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}^T$$

$$A(z) = \begin{bmatrix} & & \\ \frac{1}{2}\varphi_{1,2} + \frac{3}{8}\varphi_{2,2} + \frac{1}{8}\varphi_{3,2} & & \\ \frac{1}{2}\varphi_{1,2} + \frac{3}{8}\varphi_{2,2} + \frac{1}{8}\varphi_{3,2} - \frac{15}{16} & \frac{1}{2} & \\ \varphi_1 + \frac{3}{2}\varphi_2 + \varphi_3 - \frac{15}{8}\varphi_{0,2} & 0 & \varphi_{0,2} \end{bmatrix}$$

$$U(z) = \begin{bmatrix} 1 & 0 & 0 \\ \varphi_{0,2} & -\frac{1}{4}\varphi_{2,2} - \frac{1}{4}\varphi_{3,2} & \frac{1}{8}\varphi_{2,2} + \frac{1}{8}\varphi_{3,2} \\ \varphi_{0,2} & -\frac{1}{2}\varphi_{2,2} - \frac{1}{4}\varphi_{3,2} - \frac{3}{16} & \frac{1}{8}\varphi_{2,2} + \frac{1}{8}\varphi_{3,2} - \frac{3}{16} \\ \varphi_0 & -\varphi_2 - 2\varphi_3 + \frac{5}{4}\varphi_{0,2} & \frac{1}{2}\varphi_2 + \varphi_3 - \frac{3}{8}\varphi_{0,2} \end{bmatrix}$$

$$B(z) = \begin{bmatrix} \varphi_1 + \frac{3}{2}\varphi_2 + \varphi_3 - \frac{5}{4}\varphi_{0,2} - \frac{1}{2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{6} \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$V(z) = \begin{bmatrix} \varphi_0 & -\varphi_2 - 2\varphi_3 + \frac{5}{6}\varphi_{0,2} + \frac{1}{2} & \frac{1}{2}\varphi_2 + \varphi_3 - \frac{1}{4}\varphi_{0,2} - \frac{1}{6} \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

A.5 Modified generalized Lawson schemes

To overcome a loss of stability which some times occurs in the generalized Lawson schemes a modification is suggested in [28]. This modification only effects the solution approximation but significantly improves the accuracy of the schemes. Therefore, the following listing only include the $B(z)$ and $V(z)$ matrices.

Two schemes are included in the package but not listed here due to space reasons, they are `modgenlawson44.m` and `modgenlawson45.m`.

A.5.1 ModGenLawson41 — `modgenlawson41.m`

This scheme of stiff order two but performs significantly better than its counterpart GenLawson1.

$$B(z) = \begin{bmatrix} \varphi_1 - \varphi_2 - \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \varphi_2 - \frac{1}{2}\varphi_{0,2} \end{bmatrix}$$

$$V(z) = \begin{bmatrix} \varphi_0 \end{bmatrix}$$

A.5.2 ModGenLawson42 — `modgenlawson42.m`

This scheme of stiff order three but performs significantly better than its counterpart GenLawson2. It requires y_{n-1} and N_{n-2} to be passed from step to step.

$$B(z) = \begin{bmatrix} \varphi_1 - 2\varphi_3 - \frac{1}{2}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{2}\varphi_2 + \varphi_3 - \frac{1}{4}\varphi_{0,2} \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$V(z) = \begin{bmatrix} \varphi_0 & \frac{1}{2}\varphi_2 + \varphi_3 - \frac{1}{4}\varphi_{0,2} \\ 0 & 0 \end{bmatrix}$$

A.5.3 ModGenLawson43 — `modgenlawson43.m`

This scheme of stiff order four and does not have stability problems for large values of the timestep. It requires y_{n-1}, N_{n-2} and N_{n-3} to be passed from step to step.

$$B(z) = \begin{bmatrix} \varphi_1 + \frac{1}{2}\varphi_2 - 2\varphi_3 - 3\varphi_4 - \frac{5}{8}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_{0,2} & \frac{1}{3}\varphi_2 + \varphi_3 + \varphi_4 - \frac{5}{24}\varphi_{0,2} \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$V(z) = \begin{bmatrix} \varphi_0 & -\varphi_2 + \varphi_3 + 3\varphi_4 + \frac{5}{24}\varphi_{0,2} & \frac{1}{6}\varphi_2 - \varphi_4 - \frac{1}{24}\varphi_{0,2} \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

References

[1] H. Berland, B. Owren, and B. Skaflestad. *B*-series and order conditions for exponential integrators. *SIAM J. Numer. Anal.*, 2005. To appear.

[2] G. Beylkin, J. M. Keiser, and L. Vozovoi. A new class of time discretization schemes for the solution of nonlinear PDEs. *J. of Comp. Phys.*, 147:362–387, 1998.

[3] K. Burrage and J. C Butcher. Nonlinear stability for a general class of differential equation methods. *BIT*, 20:185–203, 1980.

[4] J. C. Butcher. *Numerical methods for ordinary differential equations.* John Wiley & Sons, 2003.

[5] M. P. Calvo and C. Palencia. A class of multistep exponential integrators for semilinear problems. Submitted, 2005.

[6] E. Celledoni, A. Marthinsen, and B. Owren. Commutator-free Lie group methods. *FGCS*, 19(3):341–352, 2003.

[7] J. Certaine. The solution of ordinary differential equations with large time constants. In *Mathematical methods for digital computers*, pages 128–132. Wiley, New York, 1960.

[8] S. M. Cox and P. C. Matthews. Exponential time differencing for stiff systems. *J. Comput. Phys.*, 176(2):430–455, 2002.

[9] B. L. Ehle and J. D. Lawson. Generalized Runge–Kutta processes for stiff initial-value problems. *J. Inst. Maths. Applics.*, 16:11–21, 1975.

[10] A. Friedli. Verallgemeinerte Runge–Kutta Verfahren zur Lösung steifer Differentialgleichungssysteme. In *Numerical treatment of differential equations*, pages 35–50. Lecture Notes in Math., Vol. 631. Springer, Berlin, 1978.

[11] R. A. Friesner, L. S. Tuckerman, B. C. Dornblaser, and T. V. Russo. A method for the exponential propagation of large stiff nonlinear differential equations. *J. Sci. Comput.*, 4(4):327–354, 1989.

[12] E. Gallopoulos and Y. Saad. Efficient solution of parabolic equations by Krylov approximation methods. *SIAM J. Sci. Statist. Comput.*, 13:1236–1264, 1992.

[13] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.

[14] M. Hochbruck and C. Lubich. On Krylov subspace approximations to the matrix exponential operator. *SIAM J. Numer. Anal.*, 34(5):1911–1925, 1997.

[15] M. Hochbruck, C. Lubich, and H Selhofer. Exponential integrators for large systems of differential equations. *SIAM J. Sci. Comput.*, 19(5):1552–1574, 1998.

[16] M. Hochbruck and A. Ostermann. Explicit exponential Runge–Kutta methods for semilinear parabolic problems. *SIAM J. Numer. Anal.*, To appear 2005.

[17] A.-K. Kassam. *High Order Timestepping for Stiff Semilinear Partial Differential Equations*. PhD thesis, University of Oxford, 2004.

[18] A.-K. Kassam and L. N. Trefethen. Fourth-order time-stepping for stiff PDEs. *SIAM J. Sci. Comput.*, 26(4):1214–1233, 2005.

[19] S. Krogstad. Generalized integrating factor methods for stiff PDEs. *J. of Comp. Phys.*, 203(1):72–88, 2005.

[20] D. J. Lawson. Generalized Runge–Kutta processes for stable systems with large lipschitz constants. *SIAM J. Numer. Anal.*, 4:372–380, 1967.

[21] Y. Y. Lu. Computing a matrix function for exponential integrators. *J. Comput. Appl. Math*, 161(1):203–216, 2003.

[22] B. Minchev and W. M. Wright. A review of exponential integrators for semilinear problems. Technical Report 2/05, The Norwegian University of Science and Technology, 2005. http://www.math.ntnu.no/preprint/.

[23] C. B. Moler and C. F. van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty five years later. *SIAM Review*, 45(1):3–49, 2003.

[24] D. R. Mott, E. S. Oran, and B. van Leer. A quasi-steady-state solver for stiff ordinary differential equations of reaction kinetics. *J. of Comp. Phys.*, 164:407–428, 2000.

[25] H. Munthe-Kaas. High order Runge–Kutta methods on manifolds. In *Proceedings of the NSF/CBMS Regional Conference on Numerical Analysis of Hamiltonian Differential Equations (Golden, CO, 1997)*, volume 29,1, pages 115–127, 1999.

[26] S. P. Nørsett. An A-stable modification of the Adams–Bashforth methods. In *Conf. on Numerical Solution of Differential Equations (Dundee, 1969)*, pages 214–219. Springer, Berlin, 1969.

[27] S. P. Nørsett. Numerisk integrasjon av stive likninger. Master's thesis, University of Oslo, 1969.

[28] A. Ostermann, M. Thalhammer, and W. Wright. More on generalized Lawson methods. In preperation, 2005.

[29] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Math. Comp.*, 37:105–126, 1981.

[30] K. Strehmel and R. Weiner. Behandlung steifer Anfangswertprobleme gewöhnlicher Differentialgleichungen mit adaptiven Runge–Kutta-Methoden. *Computing*, 29(2):153–165, 1982.

[31] K. Strehmel and R. Weiner. B-convergence results for linearly implicit one step methods. *BIT*, 27:264–281, 1987.

[32] K. Strehmel and R. Weiner. *Linear-implizite Runge–Kutta-Methoden und ihre Anwendung*, volume 127 of *Teubner-Texte zur Mathematik*. B. G. Teubner Verlagsgesellschaft mbH, Stuttgart, 1992.

[33] P. J. van der Houwen. *Construction of integration formulas for initial value problems.* North-Holland Publishing Co., Amsterdam, 1977. North-Holland Series in Applied Mathematics and Mechanics, Vol. 19.

[34] J. G. Verwer and M. van Loon. An evaluation of explicit pseudo-steady-state approximation schemes for stiff ODE systems from chemical kinetics. *J. of Comp. Phys.*, 113:347–352, 1994.