Efficient Convolution Based Impedance Boundary Conditions Master's Thesis

Author: Alberto David Maria Paganini
Supervisor: Prof. Dr. Ralf Hiptmair
ETHZ

August 17, 2011

Abstract

When formulating impedance boundary conditions in time domain, the Dirichlet-to-Neumann map of the interior of a good conductor involves convolutions. A. Schädle, M. Lopéz-Fernández and C. Lubich have developed a fast and memory efficient algorithm based on Runge-Kutta methods for computing convolutions when only the Laplace transform of the kernel is known. In this work we discuss how to couple the algorithm with BDF methods when the convolution is differentiated and we study the coupling of fast convolution quadrature with FEM for solving parabolic PDE with impedance boundary conditions involving convolutions.

Contents

1 Introduction

Alternating electromagnetic fields decay exponentially when penetrating a good conductor (skin effect). Therefore, a reasonable approximation of the electromagnetic Dirichlet-to-Neumann map of the interior of a good conductor is provided by the impedance boundary conditions

$$\boldsymbol{E_t} = \frac{1}{2}\sqrt{2}(1-i)\sqrt{\frac{\mu}{\sigma\omega}}(\boldsymbol{H} \times \boldsymbol{n}), \tag{1}$$

where $w > 0$ is a fixed angular frequency characterizing the temporal variation of all electromagnetic fields. The conductivity σ and permeability μ are known material parameters.

The relationship (1) is valid in the frequency domain only. However, often a sinusoidal temporal variation of the fields cannot be assumed, which forces us to resort to time domain methods. When formulating impedance boundary conditions in the time domain, we encounter temporal convolutions of the form

$$\boldsymbol{E_t}(\boldsymbol{x}, t) = \int_{t_0}^{t} k(\mu, \sigma, \tau - t)(\boldsymbol{H} \times \boldsymbol{n})(\boldsymbol{x}, \tau)d\tau. \tag{2}$$

In words, the boundary conditions become non-local in time. This renders a straightforward discretization of (2) prohibitively expensive, if many timesteps are to be carried out.

Instead of the eddy current problem we will consider a second-order parabolic problem with a boundary condition involving convolution in time, in which just the Laplace transform of the kernel is known. In order to approximate this kind of convolution we will discuss the algorithm developed by A. Schädle, M. Lopéz-Fernández and C. Lubich [15] and see how it applies. We will conclude with concrete numerical experiments based on the library LehrFEM.

2 Simple model problem

The general Maxwell's equation reads

$$\begin{cases} \boldsymbol{\nabla} \cdot \boldsymbol{D} = \rho \\ \boldsymbol{\nabla} \times \boldsymbol{H} - \frac{\partial \boldsymbol{D}}{\partial t} = \boldsymbol{J_f}, \\ \boldsymbol{\nabla} \cdot \boldsymbol{B} = 0, \\ \boldsymbol{\nabla} \times \boldsymbol{E} + \frac{\partial \boldsymbol{B}}{\partial t} = 0, \end{cases}$$

where $\boldsymbol{D}$ is the electric displacement, $\boldsymbol{H}$ is the magnetic displacement, $\boldsymbol{J_f}$ is the free current density, $\boldsymbol{B}$ is the magnetic field and $\boldsymbol{E}$ is the electric field. In our model we consider a good conductor with the form of an infinitely long cylinder parallel to the z-axis, with a Lipschitz section parallel to the xy-plane. The region inside the conductor is denoted as Ω_0 and the region outside the conductor as Ω (see figure 1).

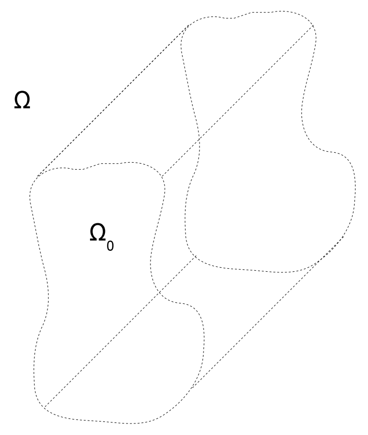


Figure 1: The regions Ω and Ω_0.

In order to derive the equations in Ω_0, it is assumed that the conductor is isotropic and homogeneous. This implies that

$$\boldsymbol{B} = \mu \boldsymbol{H},$$

where the positive constant μ is the permeability and is a known material parameter. As for the eddy current problem, a simplified version of the Maxwell's equation is considered by dropping the electric displacement $\boldsymbol{D}$. Furthermore the free current density in a conductor is given by Ohm's law

$$\boldsymbol{J_f} = \sigma \boldsymbol{E},$$

where the positive constant σ is the conductivity of the conductor (and is, in general, very large). Thus in Ω_0 we reduce the Maxwell's equations to

$$\left\{\begin{array}{l} \boldsymbol{\nabla} \times \boldsymbol{H} = \sigma \boldsymbol{E}, \\ \boldsymbol{\nabla} \times \boldsymbol{E} + \mu \frac{\partial \boldsymbol{H}}{\partial t} = 0 \end{array}\right. \tag{3}$$

Substituting the first equation of (3) in the second one gives

$$\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \boldsymbol{H} + \mu\sigma \frac{\partial \boldsymbol{H}}{\partial t} = 0 \qquad \text{in } \ \Omega_0. \tag{4}$$

The geometry of the region suggests the assumption of a z-symmetry of the fields. It is also assumed that the electric field is transversal to the conductor[1], id est

$$\boldsymbol{E} = (E_x(x,y,t), E_y(x,y,t), 0)^T.$$

This implies that the magnetic displacement is of the form

$$\boldsymbol{H} = (0, 0, H_z(x,y,t))^T.$$

Thus equation (4) becomes

$$\mu\sigma \frac{\partial H_z}{\partial t} - \Delta H_z = 0 \qquad \text{in } \ \Omega_0. \tag{5}$$

In order to derive the equation for H_z in the outer region (which is assumed to be empty), the electric displacement D is dropped from the Maxwell's equations and is added a source current density $\boldsymbol{J_s}$, which lies outside the conductor, is parallel to the conductor section and constant in the z-direction. Then we have

$$\boldsymbol{\nabla} \times \boldsymbol{H} = \boldsymbol{J_s}.$$

Taking the curl of the equation above gives

$$-\Delta H_z = f \qquad \text{in } \Omega, \tag{6}$$

where $f := \frac{\partial J_y}{\partial x} - \frac{\partial J_x}{\partial y}$. Setting $\mu = 1$ and defining $u = H_z$ (in order to simplify the notation) allows collecting equations (5) and (6) in the following parabolic partial differential equation

$$\left\{\begin{array}{rcl} \frac{\partial}{\partial t}(\tilde{\sigma} u) - \Delta u &=& f \ \text{ in } \Omega\times]0;\text{T}], \\ u &=& 0 \ \text{ in } \Omega \times \{t = 0\}, \end{array}\right. \tag{7}$$

[1]This approach is similar to that of section 1 of [3]

with $supp(f) \subset \Omega \setminus \Omega_0$ and where

$$\tilde{\sigma} = \begin{cases} \sigma >> 1 & \text{in } \Omega_0 \subset\subset \Omega, \\ 0 & \text{in } \Omega \setminus \Omega_0. \end{cases}$$

In general there's an additional radiation condition on u at infinity, which can be approximated by considering a large enough boundary of Ω, denoted $\partial\Omega$, and by imposing a homogeneous Dirichlet boundary condition on it.

Our goal is to compute the solution u in $\Omega \setminus \Omega_0$. This requires an additional boundary condition on $\partial\Omega_0$. Since σ is assumed to be very large, an impedance boundary condition provides a good approximation. In order to derive it we consider a 1D-problem by restricting the dimension of equation (7) and by moving to Laplace domain[2] (we assume that $Re(s) > 0$). By adding a radiation condition (which approximates the skin effect), equation (7) becomes

$$\begin{cases} s\sigma\mathcal{L}(u) - \partial_\xi^2\mathcal{L}(u) = & 0 \quad \text{for } \xi > 0, \\ u(\xi) \to & 0 \quad \text{for } \xi \to \infty. \end{cases} \tag{8}$$

A solution of (8) can be found with the Ansatz

$$\mathcal{L}(u)(\xi) = e^{\lambda\xi}.$$

With the decay condition at infinity we find

$$\mathcal{L}(u)(\xi) = e^{-\sqrt{s\sigma}\xi}.$$

Differentiating $\mathcal{L}(u)$ one time gives

$$\partial_\xi\mathcal{L}(u)(\xi) = -\sqrt{s\sigma}\mathcal{L}(u)(\xi)$$

and thus, by taking the limit $\xi \to 0$, we obtain the relation

$$\partial_\xi\mathcal{L}(u)(0) = -\sqrt{s\sigma}\mathcal{L}(u)(0). \tag{9}$$

Due to its smoothness, the conductor boundary can be approximated as locally flat. Applying relation (9) in each point with the ξ direction given by the normal motivates the following impedance boundary condition

$$\boldsymbol{\nabla}\mathcal{L}(u) \cdot \mathbf{n} = -\sqrt{s\sigma}\mathcal{L}(u) \quad \text{on } \partial\Omega_0.$$

[2]$\mathcal{L}(u)$ denotes the Laplace transform of u, see definition 8 on page 68 for the definition of the Laplace transform.

The square root is not Laplace invertible, thus in the time domain this boundary condition is interpreted as

$$\boldsymbol{\nabla} u \cdot \boldsymbol{n} \;=\; \frac{d}{dt}\int_0^t k(t-\tau)\cdot u(\tau)d\tau,$$

where $\mathcal{L}(k) = -\sqrt{\frac{\sigma}{s}}$. Note that the inner integral is the convolution of the kernel k and the function u. Then equation (7) becomes

$$\begin{cases} -\Delta u &= f & \text{in } \Omega \setminus \Omega_0 \times]0;\mathrm{T}], \\ u &= 0 & \text{in } \Omega \setminus \Omega_0 \times \{t=0\}, \\ u &= 0 & \text{on } \partial\Omega \times]0;\mathrm{T}], \\ \boldsymbol{\nabla} u \cdot \boldsymbol{n} &= \frac{d}{dt}\int_0^t k(t-\tau)\cdot u(\tau)d\tau & \text{on } \partial\Omega_0 \times]0;\mathrm{T}]. \end{cases}$$

The corresponding variational problem is

$$\int_{\Omega\setminus\Omega_0} \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v \; dx - \int_{\partial\Omega_0} \boldsymbol{\nabla} u \cdot \boldsymbol{n} \cdot v \; dS = \int_{\Omega\setminus\Omega_0} f \cdot v \; dx$$

for every $v \in H^1(\Omega \setminus \Omega_0)$ with $v = 0$ on $\partial\Omega$ and for every $t \in]0;\mathrm{T}]$.

Substituting $\boldsymbol{\nabla} u \cdot \boldsymbol{n}$ in the variational form gives

$$\int_{\Omega\setminus\Omega_0} \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v \; dx - \int_{\partial\Omega_0} \Big(\frac{d}{dt}\int_0^t k(t-\tau)\cdot u(\tau)d\tau\Big)\cdot v \; dS = \int_{\Omega\setminus\Omega_0} f \cdot v \; dx \quad (10)$$

with $v \in H^1(\Omega \setminus \Omega_0), v = 0$ on $\partial\Omega$ and for every $t \in]0, T]$.

3 Runge-Kutta based convolution quadrature

In the variational formulation (10) we have to compute a convolution in which only the Laplace transform of the kernel is known. In [12] C. Lubich and A. Ostermann have developed a Runge-Kutta based convolution quadrature for the approximation of this kind of convolution. In this chapter we recall the derivation of the method and its convergence theorem.

3.1 Definitions

The following definitions set the frame of convolution quadrature. We follow section 1 of [12].

Definition 1. *Let $h > 0$ and*

$$\begin{array}{c|c} \boldsymbol{c} & \mathfrak{A} \\ \hline & \boldsymbol{b^T} \end{array}$$

be a Butcher tableau with $\mathfrak{A} = (a_{ij})_{i,j=1}^m$, $\boldsymbol{b^T} = (b_1, \cdots, b_m)$ and $\boldsymbol{c} = (c_1, \cdots, c_m)^{\boldsymbol{T}}$. The related Runge-Kutta approximation y_{n+1} at time $t = (n+1) \cdot h$ of the solution $y(t)$ of an initial value problem

$$y' = f(t, y), \qquad y(0) = y_0,$$

is given by the following scheme

$$\left\{ \begin{array}{rcll} Y_{ni} & = & y_n + h \sum_{j=1}^m a_{ij} f(t_n + c_j h, Y_{nj}) & (i = 1, \cdots, m), \\ y_{n+1} & = & y_n + h \sum_{j=1}^m b_j f(t_n + c_j h, Y_{nj}). & \end{array} \right. \tag{11}$$

Definition 2 (Order p)**.** *We say that the Runge-Kutta method has order p if the error of the method after one step satisfies*

$$y_1 - y(h) = \mathcal{O}(h^{p+1})$$

as h tend to 0 when $f(t, y)$ is smooth enough.

Definition 3 (Stage order q)**.** *A Runge-Kutta method has stage order q if the error of the internal stages is*

$$Y_{0i} - y(c_i h) = \mathcal{O}(h^{q+1}) \qquad \text{for } i = 1, \cdots, m$$

as h tend to 0 (again $f(t, y)$ must be smooth enough).

Lemma 1 (Formula 1.4 on page 107 of [12]). *A Runge-Kutta method has stage order q if and only if*

$$\sum_{j=1}^{m} a_{ij} c_j^{k-1} = \frac{c_i^k}{k} \qquad \text{for } k = 1, \cdots, q.$$

Definition 4 (Stability function). *Let* $\mathbb{1} := (1, \cdots, 1)^{\boldsymbol{T}}$ *and* $\boldsymbol{I}$ *be the identity matrix. The stability function of a Runge-Kutta scheme is defined by*

$$R(z) := 1 + z\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}. \tag{12}$$

Definition 5 (A(θ)-stability). *A Runge-Kutta method is called* $A(\theta)-stable$ *if* $\boldsymbol{I} - z\mathfrak{A}$ *is nonsingular in the sector* $|arg(-z)| \leq \theta$*, and if*

$$|R(z)| \leq 1 \qquad \text{for } |arg(-z)| \leq \theta.$$

Definition 6 (Strong A(θ)-stability). *A Runge-Kutta method is called strongly* $A(\theta) - stable$ *if: it is* $A(\theta) - stable$*, it has an invertible Runge-Kutta matrix* $\mathfrak{A}$ *and the limit of the stability function at infinity,*

$$\begin{aligned} R(\infty) &:= \lim_{Re(z)\to\infty} |R(z)| \\ &= 1 - \boldsymbol{b}^{\boldsymbol{T}}\mathfrak{A}^{-1}\mathbb{1}, \end{aligned}$$

has absolute value strictly smaller then 1.

For the derivation of the method it is also convenient to assume that

$$b_i = a_{mi} \qquad \text{for } i = 1, \cdots, m, \tag{13}$$

which implies $R(\infty) = 0$ because in this case

$$\boldsymbol{b}^{\boldsymbol{T}}\mathfrak{A}^{-1} = (0, \cdots, 0, 1)\mathfrak{A}\mathfrak{A}^{-1} = (0, \cdots, 0, 1)\boldsymbol{I} = (0, \cdots, 0, 1).$$

The most important example are the m-stage Radau IIA methods, which are strong $A(\theta) - stable$ with $\theta = \frac{\pi}{2}$ and have order $p = 2m - 1$ and stage order $q = m$ (see page 72 in section IV.5 of [5]).

3.2 Derivation of the method

Now we can start working on convolutions. We want to approximate

$$u(t) = \int_0^t k(t - \tau)g(\tau)d\tau \qquad t > 0, \tag{14}$$

only knowing $g(t)$ and the Laplace transform $K(s) := \mathcal{L}(k)(s)$ of the kernel (instead of the kernel itself). First of all the Laplace transform K is assumed to satisfy the following sectorial condition.

Assumption 1 (Sectorial Laplace transform)**.**

- *$K(s)$ is analytic in a sector $|arg(s-c)| < \pi - \varphi$ with $\varphi < \frac{1}{2}\pi$ and real c (see Figure 2),*

- *$|K(s)| \leq M|s|^{-\nu}$ for some real positive ν and M.*

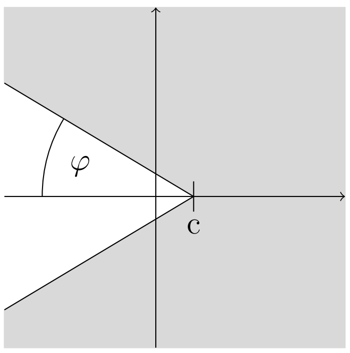


Figure 2: Domain of analyticity of $K(s)$.

Assumption 1 allows us to apply the Laplace inversion formula[3]

$$k(t) = \frac{1}{2\pi i} \int_\Gamma K(\lambda) e^{\lambda t} d\lambda, \tag{15}$$

where Γ is a contour parallel to the boundary and inside the domain of analiticity of $K(s)$, oriented with increasing imaginary part (see figure 3). By inserting (15) in (14) and applying Fubini's theorem, we obtain

$$u(t) = \frac{1}{2\pi i} \int_\Gamma K(\lambda) \int_0^t e^{\lambda\tau} g(t-\tau) d\tau \; d\lambda. \tag{16}$$

[3]See theorem 5 on page 69.

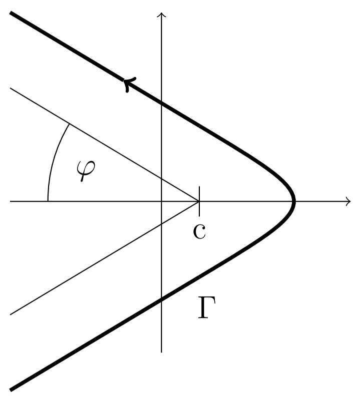


Figure 3: Contour Γ for the Laplace inversion formula (15).

Proposition 1. *The inner integral is the solution at time t of the initial value problem*

$$y' = \lambda y + g \qquad y(0) = 0. \tag{17}$$

Proof. The homogeneous solution of (17) is

$$y_h(x) = C \cdot e^{\lambda x}.$$

The variation of constants Ansatz reads

$$y_p(x) = C(x) \cdot e^{\lambda x}.$$

Inserting the Ansatz in the ODE (17) implies

$$C(x)' \cdot e^{\lambda x} + \lambda C(x) \cdot e^{\lambda x} = \lambda C(x) \cdot e^{\lambda x} + g(x)$$

thus

$$C(x) = \int_0^x e^{-\lambda a} g(a)\ da.$$

Substituting $a = x - t$ gives

$$C(x) \ = \ \int_x^0 e^{-\lambda(x-t)} g(x-t)\ dt = e^{-\lambda x} \int_0^x e^{\lambda t} g(x-t)\ dt.$$

Thus

$$y(x) = e^{\lambda x} e^{-\lambda x} \int_0^x e^{\lambda t} g(x-t)\ dt + D \cdot e^{\lambda x}.$$

We now insert the initial condition and we conclude

$$y(x) = \int_0^x e^{\lambda t} g(x-t)\ dt.$$

$\square$

The idea of the algorithm is to compute an approximation of (17), substitute it in (16) and recover an approximation of the convolution (14) by applying the Cauchy's integral formula for functions of matrices[4]. The approximation of the solution of (17) is performed by using the Runge-Kutta scheme (11) for a strong A(θ)-stable Runge-Kutta method with $\theta > \varphi$ and which further satisfies (13). In order to derive the algorithm we substitute $f(t,y) = \lambda y + g(t)$ in (11) and define the vectors (in order to reduce the amount of indexes)

$$\boldsymbol{Y_n} := (Y_{n1}, \cdots, Y_{nm})^{\boldsymbol{T}},$$
$$\boldsymbol{G_n} := (g(t_n + c_1 h), \cdots, g(t_n + c_m h))^{\boldsymbol{T}}.$$

Then the scheme (11) applied to the initial value problem (17) can be written as

$$\begin{cases} \boldsymbol{Y_n} &= \mathbb{1} y_n + h\lambda \mathfrak{A} \boldsymbol{Y_n} + h \mathfrak{A} \boldsymbol{G_n}, \\ y_{n+1} &= y_n + h\lambda \boldsymbol{b^T} \boldsymbol{Y_n} + h \boldsymbol{b^T} \boldsymbol{G_n}. \end{cases}$$

We point out that the first equation is a vector one, while the second is a scalar equation. Now we consider the generating functions of the sequences[5] $(y_n)_{n\in\mathbb{N}}$, $(\boldsymbol{Y_n})_{\boldsymbol{n}\in\mathbb{N}}$, $(\boldsymbol{G_n})_{\boldsymbol{n}\in\mathbb{N}}$ and we set

$$y(\zeta) := \mathcal{G}[(y_n)_{n\in\mathbb{N}}](\zeta) = \sum_{n=0}^{\infty} y_n \zeta^n,$$

$$\boldsymbol{Y}(\zeta) := \mathcal{G}[(\boldsymbol{Y_n})_{\boldsymbol{n}\in\mathbb{N}}](\zeta) = \sum_{n=0}^{\infty} \boldsymbol{Y_n} \zeta^n,$$

$$\boldsymbol{G}(\zeta) := \mathcal{G}[(\boldsymbol{G_n})_{\boldsymbol{n}\in\mathbb{N}}](\zeta) = \sum_{n=0}^{\infty} \boldsymbol{G_n} \zeta^n.$$

Then, since $y_0 = 0$, the scheme for (17) can be written as

$$\begin{cases} \boldsymbol{Y}(\zeta) &= \mathbb{1} y(\zeta) + h\lambda \mathfrak{A} \boldsymbol{Y}(\zeta) + h \mathfrak{A} \boldsymbol{G}(\zeta), \\ (\zeta^{-1} - 1) y(\zeta) &= h\lambda \boldsymbol{b^T} \boldsymbol{Y}(\zeta) + h \boldsymbol{b^T} \boldsymbol{G}(\zeta). \end{cases}$$

Substituting $y(\zeta)$ in the first equation from the second one gives

$$\begin{aligned} \boldsymbol{Y}(\zeta) &= \mathbb{1} \frac{1}{\zeta^{-1} - 1} \big(h\lambda \boldsymbol{b^T} \boldsymbol{Y}(\zeta) + h \boldsymbol{b^T} \boldsymbol{G}(\zeta) \big) + h\lambda \mathfrak{A} \boldsymbol{Y}(\zeta) + h \mathfrak{A} \boldsymbol{G}(\zeta) \\ &= \big(h\lambda \mathbb{1} \boldsymbol{b^T} \frac{\zeta}{1-\zeta} + h\lambda \mathfrak{A} \big) \boldsymbol{Y}(\zeta) + h \big(\mathbb{1} \boldsymbol{b^T} \frac{\zeta}{1-\zeta} + \mathfrak{A} \big) \boldsymbol{G}(\zeta). \end{aligned}$$

[4]See definition 11 on page 75 for the definition of the Cauchy's integral formula for functions of matrices.

[5]See definition 9 on page 70 for the definition of the generating function of a sequence.

We notice that $\mathbb{1}\boldsymbol{b^T}$ is a matrix which has every row equal to $\boldsymbol{b^T}$. With the definition

$$\boldsymbol{\Delta}(\zeta) := \big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\big)^{-1}.$$

and since

$$\begin{aligned}\frac{\boldsymbol{\Delta}(\zeta)}{h}\big(\boldsymbol{I} - h\lambda\mathbb{1}\boldsymbol{b^T}\frac{\zeta}{1-\zeta} - h\lambda\boldsymbol{\mathfrak{A}}\big) &= \frac{\boldsymbol{\Delta}(\zeta)}{h}\big(\boldsymbol{I} - h\lambda\big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\big)\big)\\ &= \frac{\boldsymbol{\Delta}(\zeta)}{h} - \lambda\boldsymbol{I},\end{aligned}$$

we have

$$\boldsymbol{Y}(\zeta) = \big(\frac{\boldsymbol{\Delta}(\zeta)}{h} - \lambda\boldsymbol{I}\big)^{-1}\boldsymbol{G}(\zeta). \tag{18}$$

Relation (13) implies that $y_{n+1} = Y_{nm}$ and thus $y(\zeta)$ is the last component of $\boldsymbol{Y}(\zeta)$. Let $\boldsymbol{U_n}$ be the approximated convolution vector

$$\boldsymbol{U_n} \approx (u(t_n + c_1h), \cdots, u(t_n + c_mh))^T,$$

where $u(t)$ is the exact convolution (14). Its generating function is

$$\boldsymbol{U}(\zeta) := \mathcal{G}[(\boldsymbol{U_n})_{\boldsymbol{n}\in\mathbb{N}}](\zeta) = \sum_{n=0}^{\infty}\boldsymbol{U_n}\zeta^n.$$

Substituting the numerical solution (18) in (16) implies

$$\boldsymbol{U}(\zeta) = \frac{1}{2\pi i}\int_\Gamma K(\lambda)\big(\frac{\boldsymbol{\Delta}(\zeta)}{h} - \lambda\boldsymbol{I}\big)^{-1}\boldsymbol{G}(\zeta)\ d\lambda. \tag{19}$$

To allow the next steps, we recall lemma 2.4 of [12] and lemma 2.6 of [2].

Lemma 2 (Lemma 2.4 on page 112 of [12])**.** *Under the foregoing assumptions we have*

$$\big(\boldsymbol{\Delta}(\zeta) - z\boldsymbol{I}\big)^{-1} = \boldsymbol{\mathfrak{A}}(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1} + \sum_{n=1}^{\infty} R(z)^{n-1}(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}\mathbb{1}\boldsymbol{b^T}(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}\zeta^n,$$

where $R(z)$ is the stability function defined at (12)*.*

Proof. By the definition of $\boldsymbol{\Delta}(\zeta)$ we have

$$\begin{aligned}\boldsymbol{\Delta}(\zeta) - z &= \big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\big)^{-1} - z\\ &= \Big(\boldsymbol{I} - z\big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\big)\Big)\big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\big)^{-1},\end{aligned}$$

thus

$$\big(\boldsymbol{\Delta}(\zeta) - z\big)^{-1} = \big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big)\Big(\boldsymbol{I} - z\big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big)\Big)^{-1}.$$

Regarding the second bracket on the right hand side it holds that

$$\begin{aligned}
\boldsymbol{I} - z\big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big) &= \big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big) - z\frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} \\
&= \big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big)\big(\boldsymbol{I} - z(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}\frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big) \\
&= \frac{\big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big)}{1-\zeta}\big((1-\zeta)\boldsymbol{I} - z(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}\zeta\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big) \\
&= \frac{\big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big)}{1-\zeta}\big(\boldsymbol{I} - \zeta(\boldsymbol{I} + (\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}})\big).
\end{aligned}$$

By defining $\boldsymbol{E} := \boldsymbol{I} + (\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}$ we have

$$\Big(\boldsymbol{I} - z\big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big)\Big)^{-1} = (1-\zeta)\big(\boldsymbol{I} - \zeta\boldsymbol{E}\big)^{-1}\big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big)^{-1}$$

and thus

$$\big(\boldsymbol{\Delta}(\zeta) - z\big)^{-1} = \big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big)(1-\zeta)\big(\boldsymbol{I} - \zeta\boldsymbol{E}\big)^{-1}\big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big)^{-1}.$$

For ζ small enough it holds that

$$\begin{aligned}
(1-\zeta)\big(\boldsymbol{I} - \zeta\boldsymbol{E}\big)^{-1} &= (1-\zeta)\sum_{n=0}^{\infty}\boldsymbol{E}^n\zeta^n \\
&= \sum_{n=0}^{\infty}\boldsymbol{E}^n\zeta^n - \sum_{n=0}^{\infty}\boldsymbol{E}^n\zeta^{n+1} \\
&= \boldsymbol{I} + \sum_{n=1}^{\infty}(\boldsymbol{E}^n - \boldsymbol{E}^{n-1})\zeta^n,
\end{aligned}$$

therefore

$$\big(\boldsymbol{\Delta}(\zeta) - z\big)^{-1} = \big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big)\big(\boldsymbol{I} + \sum_{n=1}^{\infty}(\boldsymbol{E}^n - \boldsymbol{E}^{n-1})\zeta^n\big)\big(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}}\big)^{-1}.$$

Now we show per induction that

$$\boldsymbol{E}^n = \boldsymbol{I} + (\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big(\sum_{m=0}^{n-1}R(z)^m\big) \qquad \text{for } n \geq 2. \tag{20}$$

For $n = 2$ we have

$$\begin{aligned}
\boldsymbol{E}^2 &= (\boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}})^2 \\
&= \boldsymbol{I} + 2(\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A}) z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} \\
&= \boldsymbol{I} + 2(\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}(R(z) - 1)\boldsymbol{b}^{\boldsymbol{T}} \\
&= \boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(R(z) + 1).
\end{aligned}$$

The induction hypothesis reads

$$\boldsymbol{E}^k = \boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m)$$

and consequently the induction step is

$$\begin{aligned}
\boldsymbol{E}^{k+1} &= \boldsymbol{E} \cdot \boldsymbol{E}^k \\
&= (\boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}})(\boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m)) \\
&= \boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m) + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} \\
&\quad + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m) \\
&= \boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m) + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} \\
&\quad + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}(R(z) - 1)\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m) \\
&= \boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k-1} R(z)^m + 1 + (R(z) - 1)\sum_{m=0}^{k-1} R(z)^m) \\
&= \boldsymbol{I} + (\boldsymbol{I} - z\mathfrak{A})^{-1} z\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\sum_{m=0}^{k} R(z)^m)
\end{aligned}$$

which shows (20). Moreover we compute

$$\begin{aligned}
\mathbb{1}\boldsymbol{b^T}\boldsymbol{E}^n &= \mathbb{1}\boldsymbol{b^T}(\boldsymbol{I} + (\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}z\mathbb{1}\boldsymbol{b^T}(\sum_{m=0}^{n-1} R(z)^m)) \\
&= \mathbb{1}\boldsymbol{b^T} + \mathbb{1}\boldsymbol{b^T}(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}z\mathbb{1}\boldsymbol{b^T}(\sum_{m=0}^{n-1} R(z)^m) \\
&= \mathbb{1}\boldsymbol{b^T} + \mathbb{1}(R(z)-1)\boldsymbol{b^T}(\sum_{m=0}^{n-1} R(z)^m) \\
&= R(z)^n\mathbb{1}\boldsymbol{b^T}
\end{aligned}$$

and

$$\begin{aligned}
\boldsymbol{\mathfrak{A}}(\boldsymbol{E}^n - \boldsymbol{E}^{n-1}) &= \boldsymbol{\mathfrak{A}}(R(z)^{n-1}(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}z\mathbb{1}\boldsymbol{b^T}) \\
&= R(z)^{n-1}z\boldsymbol{\mathfrak{A}}(\boldsymbol{I} - z\boldsymbol{\mathfrak{A}})^{-1}\mathbb{1}\boldsymbol{b^T}.
\end{aligned}$$

Collecting all the results gives the equality of the lemma.

$$
\begin{aligned}
\big(\boldsymbol{\Delta}(\zeta) - z\big)^{-1} &= \big(\mathfrak{A} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\big)\big(\boldsymbol{I} + \sum_{n=1}^{\infty}(\boldsymbol{E}^n - \boldsymbol{E}^{n-1})\zeta^n\big)\big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \mathfrak{A}\sum_{n=1}^{\infty}(\boldsymbol{E}^n - \boldsymbol{E}^{n-1})\zeta^n + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} \\
&\quad + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\sum_{n=1}^{\infty}(\boldsymbol{E}^n - \boldsymbol{E}^{n-1})\zeta^n\Big) \cdot \big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1} z\mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}} \\
&\quad + \frac{\zeta}{1-\zeta}\sum_{n=1}^{\infty}(R(z)^n - R(z)^{n-1})\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n\Big) \cdot \big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1} z\mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n \\
&\quad + \frac{1}{1-\zeta}\big(\sum_{n=0}^{\infty} R(z)^n\zeta^{n+1} - \sum_{n=1}^{\infty} R(z)^{n-1}\zeta^{n+1}\big)\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\Big) \cdot \big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1} z\mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n \\
&\quad + \frac{1}{1-\zeta}\sum_{n=1}^{\infty} R(z)^{n-1}(\zeta^n - \zeta^{n+1})\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\Big) \cdot \big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1} z\mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n + \sum_{n=1}^{\infty} R(z)^{n-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n\Big) \\
&\quad \cdot \big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1}\big(\boldsymbol{I} + z\mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1})\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n\big)\Big)\big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1}(\boldsymbol{I} - z\mathfrak{A})^{-1}((\boldsymbol{I} - z\mathfrak{A}) + z\mathfrak{A})\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n\big)\Big)\big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \Big(\mathfrak{A} + \sum_{n=1}^{\infty} R(z)^{n-1}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\zeta^n\big)\Big)\big(\boldsymbol{I} - z\mathfrak{A}\big)^{-1} \\
&= \mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1} + \sum_{n=1}^{\infty} R(z)^{n-1}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A})^{-1}\zeta^n.
\end{aligned}
$$

□

Lemma 3 (Lemma 2.6 on page 5 of [2]). *For strong $A(\theta)$-stable Runge-Kutta methods, and for $|\zeta| < 1$, $\sigma(\Delta(\zeta))$ is contained in the open right sector $\{z \in \mathbb{C}|\ |arg(z)| < \pi - \theta\}$, in particular*

$$\sigma(\boldsymbol{\Delta}(\zeta)) = \sigma(\mathfrak{A}^{-1}) \cup \{z \in \mathbb{C} : R(z)\zeta = 1\}. \tag{21}$$

Proof. With lemma 2 we have

$$\begin{aligned}\big(\boldsymbol{\Delta}(\zeta) - z\boldsymbol{I}\big)^{-1} &= \mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1} + \sum_{n=1}^{\infty} R(z)^{n-1}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A})^{-1}\zeta^n \\ &= \mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1} + \zeta(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A})^{-1}\sum_{n=0}^{\infty} R(z)^n\zeta^n \\ &= \mathfrak{A}(\boldsymbol{I} - z\mathfrak{A})^{-1} + \frac{\zeta}{1 - R(z)\zeta}(\boldsymbol{I} - z\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}(\boldsymbol{I} - z\mathfrak{A})^{-1}\end{aligned}$$

Thus for ζ fixed this implies (21). Definition 5 provides the geometric description of $\sigma(\boldsymbol{\Delta}(\zeta))$. □

Recall that we have chosen a Runge-Kutta method with $\theta > \varphi$. Thus, for h small enough, $\sigma(\frac{\boldsymbol{\Delta}(\zeta)}{h})$ is in the sector of analyticity of K. Then definition 11 on page 75 can be applied, giving

$$K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big) = \frac{1}{2\pi i}\int_{\Gamma} K(\lambda)\big(\frac{\boldsymbol{\Delta}(\zeta)}{h} - \lambda\boldsymbol{I}\big)^{-1}\, d\lambda. \tag{22}$$

By inserting (22) in (19) we obtain

$$\boldsymbol{U}(\zeta) = K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big)\boldsymbol{G}(\zeta).$$

By defining $\boldsymbol{W_n}$ by

$$K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big) = \sum_{n=0}^{\infty}\boldsymbol{W_n}\zeta^n,$$

we finally find

$$\boldsymbol{U_n} = \sum_{j=0}^{n}\boldsymbol{W_{n-j}}\boldsymbol{G_j}. \tag{23}$$

The approximated convolution u_{n+1} is the last component of $\boldsymbol{U_n}$. The next theorem states the error bound for this kind of convolution quadrature.

Theorem 1 (Theorem 2.2 on page 109 of [12]). *Let u_n be computed as above, h be sufficiently small and u be sufficiently smooth, then*

$$|u_n - u(t_n)| = \mathcal{O}(h^p + h^{q+1+\nu}), \tag{24}$$

where ν is the same as in Assumption 1

Regarding the complexity of the algorithm, a naive implementation requires $\mathcal{O}(n)$ evaluations of the Laplace transform K, $\mathcal{O}(n^2)$ multiplications and $\mathcal{O}(n)$ active memory for the values of the function g and for the convolution weights $\boldsymbol{W_n}$. Similarly as described in [11], the multiplications can be reduced to $\mathcal{O}(n\log(n))$ by using the fast Fourier transform. Unfortunately this technique doesn't reduce the amount of evaluations of K and the memory requirement.

3.3 Summary

Here is a brief summary of how the algorithm applies.

1. The goal is to solve

$$u(t) = \int_0^t k(t-\tau)g(\tau)d\tau \qquad t>0,$$

   for $t=(n+1)h$ and only knowing $K(s)$ and $g(t)$.

2. Choose a strong A(θ)-stable Runge-Kutta method with $\theta > \varphi$ and which further satisfies (13) and compute

$$\boldsymbol{\Delta}(\zeta) = \big(\boldsymbol{\mathfrak{A}} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\big)^{-1}.$$

3. The weight matrices $\boldsymbol{W_n}$ are computed from the Taylor expansion of

$$K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big) = \sum_{n=0}^{\infty}\boldsymbol{W_n}\zeta^n.$$

4. The approximated solution u_{n+1} is the last component of

$$\sum_{j=0}^{n}\boldsymbol{W_{n-j}G_j}.$$

Remark 1. *If $R(\infty)=0$, on page 5 of [2] is given a simpler representation of $\boldsymbol{\Delta}(\zeta)$:*

$$\boldsymbol{\Delta}(\zeta) = \boldsymbol{\mathfrak{A}}^{-1} - \zeta\boldsymbol{\mathfrak{A}}^{-1}\mathbb{1}\boldsymbol{b^T}\boldsymbol{\mathfrak{A}}^{-1}.$$

Proof. The Sherman-Morrison formula states that for an invertible square matrix $\boldsymbol{A}$ and for two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, for which $1+\boldsymbol{v^T}\boldsymbol{A}^{-1}\boldsymbol{u}\neq 0$, it holds

$$\big(\boldsymbol{A}+\boldsymbol{uv^T}\big)^{-1} = \boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1}\boldsymbol{uv^T}\boldsymbol{A}^{-1}}{1+\boldsymbol{v^T}\boldsymbol{A}^{-1}\boldsymbol{u}}.$$

Then, since

$$\begin{aligned}1+\boldsymbol{b^T}\mathfrak{A}^{-1}\frac{\zeta}{1-\zeta}\mathbb{1} &= 1+\frac{\zeta}{1-\zeta}\boldsymbol{b^T}\mathfrak{A}^{-1}\mathbb{1}\\ &= 1+\frac{\zeta}{1-\zeta}(1-R(\infty))\\ &= 1+\frac{\zeta}{1-\zeta}\\ &= \frac{1}{1-\zeta},\end{aligned}$$

we have

$$\begin{aligned}\boldsymbol{\Delta}(\zeta) &= \left(\mathfrak{A}+\frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\right)^{-1}\\ &= \mathfrak{A}^{-1}-\frac{\mathfrak{A}^{-1}\frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b^T}\mathfrak{A}^{-1}}{\frac{1}{1-\zeta}}\\ &= \mathfrak{A}^{-1}-\zeta\mathfrak{A}^{-1}\mathbb{1}\boldsymbol{b^T}\mathfrak{A}^{-1}.\end{aligned}$$

□

Remark 2. *If it is too complicated to compute point* 3 *analytically, on page 11 of [12] it is suggested to compute the weight matrices by approximating the Cauchy integral*

$$\boldsymbol{W_n}=\frac{1}{2\pi i}\int_{|\zeta|=\rho}\frac{K(\boldsymbol{\Delta}(\zeta)/h)}{\zeta^{n+1}}d\zeta.$$

With the substitution $\zeta=\rho e^{i\phi}$, this is equal to

$$\begin{aligned}\boldsymbol{W_n} &= \frac{1}{2\pi\rho^n}\int_0^{2\pi}K(\frac{\boldsymbol{\Delta}(\rho e^{i\phi})}{h})e^{-ni\phi}d\phi\\ &= \frac{1}{2\pi\rho^n}\sum_{l=0}^{L-1}\int_{l\cdot h}^{(l+1)h}K(\frac{\boldsymbol{\Delta}(\rho e^{i\phi})}{h})e^{-ni\phi}d\phi,\end{aligned}$$

where $L\cdot h=2\pi$. Applying the trapezoidal rule to each integral we obtain the following approximation

$$\boldsymbol{W_n}\approx\frac{\rho^{-n}}{L}\sum_{l=0}^{L-1}K(\frac{\boldsymbol{\Delta}(\rho e^{2\pi il/L})}{h})e^{-2\pi inl/L}.$$

If we assume, that we can compute the values of K with precision ε, the first N $\boldsymbol{W_n}$s will have an error of $\mathcal{O}(\sqrt{\varepsilon})$ choosing $L=N$ and $\rho^N=\sqrt{\varepsilon}$.

Moreover, choosing $L \geq N|\log(\varepsilon)|$ and $\rho = e^{-\gamma h}$ with $\gamma > c$ of assumption 1, the error becomes $\mathcal{O}(\varepsilon)$. The weight matrices can be computed in $\mathcal{O}(L \log L)$ operations using fast Fourier transform.

Remark 3. *Computing the convolution weights $\boldsymbol{W_n}$ is much simpler if the matrix $\boldsymbol{\Delta}(\zeta)$ is diagonalizable. In his work [1], L. Banjai investigates the diagonalizability of this matrix for the 2-stage and for the 3-stage RadauIIA methods. In proposition 3.4 on page 2971 of [1] he states that in the 2-stage RadauIIA method case, $\boldsymbol{\Delta}(\zeta)$ is diagonalizable for all $|\zeta| < 1$, except for $\zeta = 3\sqrt{3} - 5$. In the following remark 3.5 on the same page he considers the 3-stage RadauIIA method case and finds that the matrix is diagonalizable for all $|\zeta| < 1$, with an exception for $|\zeta| = 0.069366077\ldots$. He concludes the remark by saying that in practice it is highly unlikely that during the computations $\boldsymbol{\Delta}(\zeta)$ is evaluated in a ζ for which it is not diagonalizable and therefore its diagonalizability has not to be verified.*

Remark 4. *If we have an integral equation instead of a convolution in point 1 (let's say that $g = g(t, u)$), we can still apply the algorithm above. The only difference is that, instead of computing directly u_{n+1}, we have to solve n implicit system*

$$\boldsymbol{U_k} = \sum_{j=0}^{k} \boldsymbol{W_{k-j}} \boldsymbol{G_j^u} \qquad \text{for } k = 1, \cdots, n,$$

where $\boldsymbol{G_j^u} = (g(t_j + c_1 h, u_{1j}), \cdots, g(t_j + c_m h, u_{mj})^{\boldsymbol{T}}$ and $u_{ij} \approx u(t_j + c_i h)$. In this case the number of multiplications required is $\mathcal{O}(n^2)$ for a naive implementation and $\mathcal{O}(n(\log n)^2)$ by using the fast Fourier transform technique, as described in [4]. Again the amount of evaluations of the Laplace transform K as well as the memory requirement are not reduced by using FFT and remain $\mathcal{O}(n)$.

3.4 Numerical examples of a convolution

Here are some numerical examples of the application of the Runge-Kutta convolution quadrature to the the continuous convolution

$$\int_0^2 \frac{1}{\sqrt{\pi(2-\tau)}} e^{\tau} d\tau.$$

The following analytic computations provide the reference solution

$$\begin{aligned}
\int_0^2 \frac{1}{\sqrt{\pi(2-\tau)}} e^{\tau} d\tau &= \frac{e^2}{\sqrt{\pi}} \int_0^2 \frac{e^{\tau-2}}{\sqrt{2-\tau}} d\tau \\
&= \frac{e^2}{\sqrt{\pi}} \int_0^2 \frac{e^{-(2-\tau)}}{\sqrt{2-\tau}} d\tau \\
&= \frac{e^2}{\sqrt{\pi}} \int_{\sqrt{2}}^0 \frac{e^{-x^2}}{x} (-2x) dx \\
&= e^2 \frac{2}{\sqrt{\pi}} \int_0^{\sqrt{2}} e^{-x^2} dx \\
&= e^2 \mathrm{erf}(\sqrt{2}),
\end{aligned}$$

where $\mathrm{erf}(t)$ is the error function. In order to apply the convolution quadrature we choose $k(t) = \frac{1}{\sqrt{\pi t}}$, whose Laplace transform is $K(s) = \frac{1}{\sqrt{s}}$, as the kernel of this convolution. As Runge-Kutta method, we consider the RadauIIA methods with one and two stages. The codes can be found in the appendix $B.1$.

Example 1 (1-stage RadauIIA)**.** *Well known as implicit Euler's method, it is the most simple case and its weights can be computed analytically without too much effort. We start by its Butcher tableau, which is*

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

Then we compute

$$\begin{aligned}
\mathbf{\Delta}(\zeta) &= \big(\mathfrak{A} + \frac{\zeta}{1-\zeta} \mathbb{1} \boldsymbol{b^T}\big)^{-1} \\
&= (1 + \frac{\zeta}{1-\zeta})^{-1} \\
&= 1 - \zeta.
\end{aligned}$$

From

$$\begin{aligned}
K\big(\frac{\mathbf{\Delta}(\zeta)}{h}\big) &= K\big(\frac{1-\zeta}{h}\big) \\
&= \sqrt{\frac{h}{1-\zeta}} \\
&= \sqrt{h} \sum_{n=0}^{\infty} (-1)^n \binom{-\frac{1}{2}}{n} \zeta^n
\end{aligned}$$

we can immediately read

$$\boldsymbol{W_n} = \sqrt{h}(-1)^n \binom{-\frac{1}{2}}{n},$$

where

$$\begin{aligned}\binom{-\frac{1}{2}}{n} &:= \frac{\Gamma(-\frac{1}{2}+1)}{\Gamma(n+1)\Gamma(-\frac{1}{2}-n+1)}, \\ &= \frac{\Gamma(\frac{1}{2})}{\Gamma(n+1)\Gamma(\frac{1}{2}-n)}.\end{aligned}$$

The convolution quadrature is now computed as in point 4 *of the summary on page 20. The results can be read in table* 1. *The algebraic convergence*

time step	Absolute error
2^{-1}	1.6953
2^{-2}	0.8416
2^{-3}	0.4186
2^{-4}	0.2086
2^{-5}	0.1041

Table 1: Absolute error of implicit euler method at time $T = 2$ versus time step h.

of order one (see theorem 1*) is confirmed by the numerical experiment (the approximate order of convergence is* 1.006185*, see figure* 4 *on page* 26 *for a plot of the absolute error versus the time step).*

Example 2 (2-stage RadauIIA)**.** *The Butcher tableau of the 2-stage Radau IIA method is*

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

Similarly to example 1 we compute

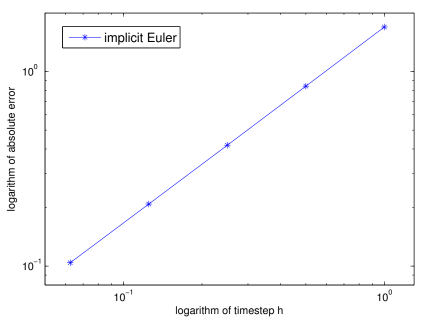
$$\begin{aligned}
\mathbf{\Delta}(\zeta) &= \left(\mathfrak{A} + \frac{\zeta}{1-\zeta}\mathbb{1}\boldsymbol{b}^{\boldsymbol{T}}\right)^{-1} \\
&= \left(\begin{pmatrix} \frac{5}{12} & -\frac{1}{12} \\ \frac{3}{4} & \frac{1}{4} \end{pmatrix} + \frac{\zeta}{1-\zeta}\begin{pmatrix} 1 \\ 1 \end{pmatrix}\begin{pmatrix} \frac{3}{4} & \frac{1}{4} \end{pmatrix}\right)^{-1} \\
&= \left(\begin{pmatrix} \frac{5}{12} & -\frac{1}{12} \\ \frac{3}{4} & \frac{1}{4} \end{pmatrix} + \frac{\zeta}{1-\zeta}\begin{pmatrix} \frac{3}{4} & \frac{1}{4} \\ \frac{3}{4} & \frac{1}{4} \end{pmatrix}\right)^{-1} \\
&= \left(\frac{1}{12(\zeta-1)}\begin{pmatrix} -4\zeta-5 & 1-4\zeta \\ -9 & -3 \end{pmatrix}\right)^{-1} \\
&= \frac{1}{2}\begin{pmatrix} 3 & 1-4\zeta \\ -9 & 4\zeta+5 \end{pmatrix}.
\end{aligned}$$

The analytic computation of the convolution weights is more complicated than in example 1 because the function K acts on the matrix $\frac{\mathbf{\Delta}(\zeta)}{h}$. The latter has to be considered as an operator and therefore the function K acts on the eigenvalues of $\frac{\mathbf{\Delta}(\zeta)}{h}$ and not on its components. Thus it is more convenient to compute the convolution weights as in remark 2*, where, in order to compute the term $K(\frac{\mathbf{\Delta}(\rho e^{2\pi i l/L})}{h})$, we must first compute the eigenvalue decomposition of $\frac{\mathbf{\Delta}(\rho e^{2\pi i l/L})}{h}$. The results can be read in table* 2 *(and seen in figure* 4*). The*

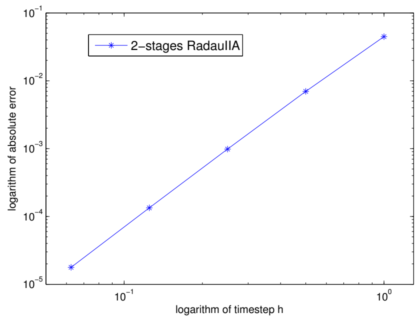
time step	Absolute error
2^{-1}	0.0448
2^{-2}	0.0070
2^{-3}	$9.8455 \cdot 10^{-4}$
2^{-4}	$1.3388 \cdot 10^{-4}$
2^{-5}	$1.7772 \cdot 10^{-5}$

Table 2: Absolute error of 2-stage RadauIIA method at time $T = 2$ versus time step h.

experiment shows an approximated convergence order of 2.829789*, while its theoretic rate is* 3*. The difference is mainly due to a rough choice of the parameters for the approximation of the convolution weights.*



(a) Implicit Euler



(b) 2-stage RadauIIA

Figure 4: Absolute error of convolution quadrature at time $T = 2$ versus time step h in double-logarithmic scale. We observe algebraic convergence.

4 Fast and oblivious convolution quadrature (FCQ)

In paper [15] A. Schädle, M. Lopéz-Fernández and C. Lubich apply the idea of paper [13] to the algorithm described in chapter 3 in order to make it faster and more memory efficient. They reduce the number of multiplications to $\mathcal{O}(n \log n)$, to $\mathcal{O}(\log n)$ the evaluations of the Laplace transform K and to $\mathcal{O}(\log n)$ the active memory requirement. These values hold both for computing a convolution and for solving an integral equation. Here we repeat the derivation of this method.

4.1 A contour integral representation for weight matrices

The fast algorithm is based on a different representation of the weight matrices. So far we know

$$K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big) = \sum_{n=0}^{\infty} \boldsymbol{W_n} \zeta^n.$$

On the other hand, by the Cauchy's integral formula it holds that

$$K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big) = \frac{1}{2\pi i} \int_{\Gamma} K(\lambda) \big(\frac{\boldsymbol{\Delta}(\zeta)}{h} - \lambda \boldsymbol{I}\big)^{-1} \, d\lambda.$$

A new representation for the weight matrices can be derived in terms of a new sequence of matrices $\boldsymbol{E_n}(z)$ defined by

$$\big(\boldsymbol{\Delta}(\zeta) - z\boldsymbol{I}\big)^{-1} = \sum_{n=0}^{\infty} \boldsymbol{E_n}(z) \zeta^n.$$

Since

$$\big(\frac{\boldsymbol{\Delta}(\zeta)}{h} - \lambda \boldsymbol{I}\big)^{-1} = h \sum_{n=0}^{\infty} \boldsymbol{E_n}(h\lambda) \zeta^n,$$

it follows that

$$\begin{aligned}\sum_{n=0}^{\infty} \boldsymbol{W_n}\zeta^n &= K\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}\big)\\ &= \frac{1}{2\pi i}\int_\Gamma K(\lambda)\big(\frac{\boldsymbol{\Delta}(\zeta)}{h}-\lambda\boldsymbol{I}\big)^{-1}\,d\lambda\\ &= \frac{1}{2\pi i}\int_\Gamma K(\lambda)h\sum_{n=0}^{\infty}\boldsymbol{E_n}(h\lambda)\zeta^n\,d\lambda\\ &= \sum_{n=0}^{\infty}\Big(\frac{h}{2\pi i}\int_\Gamma K(\lambda)\boldsymbol{E_n}(h\lambda)\,d\lambda\Big)\zeta^n.\end{aligned}$$

Thus the new representation reads

$$\boldsymbol{W_n} = \frac{h}{2\pi i}\int_\Gamma K(\lambda)\boldsymbol{E_n}(h\lambda)\,d\lambda. \tag{25}$$

4.2 Approximation of the contour integral

The task now is to find an optimal way to compute (25). In paper [9] M. Lopéz-Fernández, C. Palencia and A. Schädle have found that an effective approximation is obtained by applying the trapezoidal rule to a parametrization of a hyperbola, which plays the role of Γ and depends on n. We discuss now their procedure.

For a fixed integer value B a sequence of overlapping intervals is defined by

$$I_\ell := [B^{\ell-1}, 2B^\ell) \qquad \text{for } \ell \geq 1.$$

Each weight matrice $\boldsymbol{W_n}$ with $n \in I_\ell$ is computed as in (25) by choosing the contour $\Gamma = -\Gamma_\ell$, where Γ_ℓ is the left branch of a hyperbola associated to the interval I_ℓ. This hyperbola is parametrized by

$$\begin{array}{rcl}\mathbb{R} & \to & \Gamma_\ell\\ \theta & \mapsto & \gamma_\ell(\theta) := \mu_\ell(1-\sin(\alpha_\ell + i\theta)) + \sigma\end{array} \tag{26}$$

for a suitable choice of the parameter $\mu_\ell > 0$, α_ℓ and σ, so that the singularities of K lie to the left of it while the singularities of $\boldsymbol{E_n}(h\lambda)$ lie to the right of it[6]. Regarding the parametrisation (26), we notice that the asymptotes have slope $\pm\text{cotan}(\alpha_\ell)$, the hyperbola center is $\mu_\ell + \sigma$ and for $\mu_\ell > 0$ the hyperbola is oriented with decreasing imaginary part (see figure 5).

[6]The precise discussion about the parameters is done on page 31.

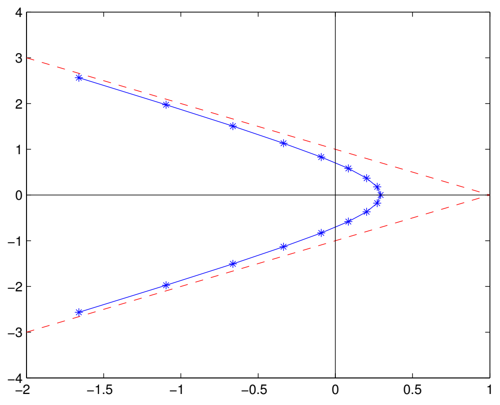


Figure 5: Left branch of a hyperbola with $\mu_\ell = 1$, $\sigma = 0$ and $\alpha_\ell = \frac{\pi}{4}$.

The contour integral is finally discretized with the trapezoidal rule (with $2N+1$ points). This gives

$$\begin{aligned} \boldsymbol{W_n} &= \frac{h}{2\pi i}\int_\Gamma K(\lambda)\boldsymbol{E_n}(h\lambda)\,d\lambda \\ &= \frac{hi}{2\pi}\int_{\Gamma_\ell} K(\lambda)\boldsymbol{E_n}(h\lambda)\,d\lambda \qquad (27) \\ &\approx h\sum_{k=-N}^{N} \omega_k^{(\ell)} K(\lambda_k^{(\ell)})\boldsymbol{E_n}(h\lambda_k^{(\ell)}), \qquad (28) \end{aligned}$$

where, for a step τ,

$$\omega_k^{(\ell)} = \frac{i\tau}{2\pi}\gamma_\ell'(\theta_k), \quad \lambda_k^{(\ell)} = \gamma_\ell(\theta_k) \quad \text{and} \quad \theta_k = k\tau.$$

Theorem 3.1[7] on page 426 of [15] provides an error bound for the approximation (28). Here we repeat the statement.

[7]This theorem is proven similarly to theorem 3 on page 289 of [8].

Theorem 2 (Theorem 3.1 on page 426 of [15])**.** *There are positive constants $C, d, c_0, \cdots, c_4$, and c such that at $t = nh \leq T$ the quadrature error in* (28) *for a hyperbola with* $1 \leq c\mu t \leq n$ *is bounded by*

$$\|E(\tau, N, h, n)\| \leq Cht^{\nu-1}(\mu t)^{1-\nu}\Bigg(\frac{e^{c_0\mu t}}{e^{2\pi d/\tau} - 1} + e^{(c_1 - c_2\cosh(N\tau))\mu t} + e^{c_3\mu t}\big(1 + \frac{c_4\cosh(N\tau)\mu t}{n/2}\big)^{-n/2}\Bigg),$$

where ν is the exponent of assumption 1 *on page 11.*

The most important thing is that this theorem shows an exponential convergence. Moreover, assuming an error tolerance ε, and with an exception for the first few weights, an accurate choice of the parameters leads to theorem 3.2 on page 427 of [15], which we repeat.

Theorem 3 (Theorem 3.2 on page 427 of [15])**.** *In* (28)*, a quadrature error bounded in norm by $\varepsilon ht^{\nu-1}$ for $nh \in I_\ell$ is obtained with $N = \mathcal{O}(\log \frac{1}{\varepsilon})$. This holds for $n \geq c \log \frac{1}{\varepsilon}$ (with some constant $c > 0$) with N independent of ℓ and of n and h with $nh \leq T$.*

When choosing the parameters of the hyperbola, is important to note that each term of $\boldsymbol{E_n}(h\lambda)$ is an approximation to $e^{nh\lambda}$. By exchanging them, equation (27) becomes the inverse Laplace transform of K at time $t = nh$. In paper [9] M. López-Fernández, C. Palencia and A. Schädle developed a spectral order method for computing the inverse Laplace transform of a sectorial function along the hyperbola contour. In that paper the error constants are given explicitly and it contains also a section entirely dedicated to the optimal choice of the parameters of the method. Moreover the error bound of theorem 2 for large n and small h becomes similar to the error bound given in [9] for inverting the Laplace transform. Therefore the authors of [15] have decided to follow this strategy used in [9] for the choice of the parameters, which reads

1. choose two fixed integer values N and B (for the latter, in the literature is many times said that $B = 10$ is a good choice),

2. fix $\sigma = c$, $\alpha = d = (\pi/2 - \varphi)/2$, with c and φ of assumption 1 on page 11,

3. set

$$a(\rho) := \operatorname{acosh}\Big(\frac{2B}{(1-\rho)\sin\alpha}\Big) \quad \text{and} \quad \epsilon_N(\rho) := \exp\Big(-\frac{2\pi d}{a(\rho)}N\Big)$$

   and find $\rho \in \,]0,1[$ so that

$$\varepsilon \cdot \epsilon_N(\rho)^{\rho-1} + \epsilon_N(\rho)^{\rho}$$

   takes its minimum, where ε is the machine precision,

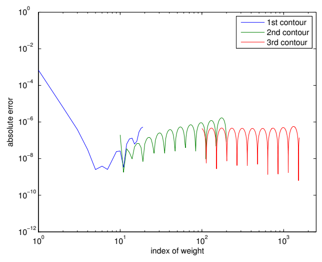
4. set

$$\tau = \frac{1}{N}a(\rho) \quad \text{and} \quad \mu = \frac{2\pi d N(1-\rho)}{(2B^{\ell}-2)ha(\rho)}.$$

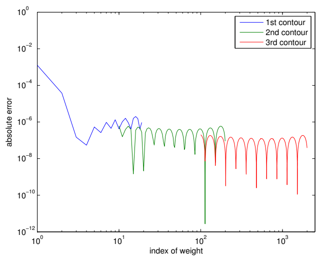
Remark 5. *Since theorem* 3 *doesn't hold for the first few weights, these are computed as in remark* 2 *on page 21.*

4.3 Numerical example of the approximation of the contour integral

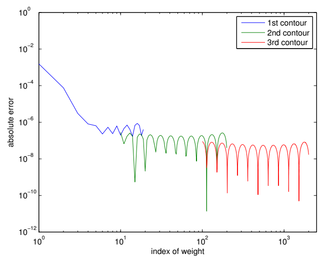
Here we give a numerical example for the approximation of the convolution weights through the contour approximation (28). The code can be found in the appendix B.2. As a complex function, we choose $K(s) = \frac{1}{\sqrt{s}}$, which is the Laplace transform of $f(t) = \frac{1}{\sqrt{\pi t}}$. As the fixed contour parameters we choose $B = 10$, $N = 15$, $\alpha = 1$ while τ and μ are computed by following the strategy discussed at the end of section 4.2. The reference solution is computed just by raising the number of quadrature points N to 50. Figure 6 contains the absolute errors of the approximation of the convolution weights versus ther indexes for the 1-stage, 2-stage and 3-stage RadauIIA methods (for the 2-stage and 3-stage RadauIIA methods we consider only the last entry of the weight matrices). We observe that, except for the first few weights, the absolute error is about 10^{-6}.



(a) Implicit Euler



(b) 2-stage RadauIIA



(c) 3-stage RadauIIA

Figure 6: Absolute error of the approximation of the convolution weights versus their indexes.

4.4 Fast convolution quadrature

As in section 3.2, we want to solve

$$u(t) = \int_0^t k(t-\tau)g(\tau)d\tau \qquad t > 0,$$

for $t = (n+1)h$ and only knowing $K(s)$ and $g(t)$. The classic approximated solution u_{n+1} is the last component of

$$\sum_{j=0}^{n} \boldsymbol{W_{n-j}G_j}. \tag{29}$$

Let L be the smallest integer such that $n < 2B^L$. The idea of FCQ is to reorganize the summation (29) in $L+1$ terms

$$\begin{aligned} \boldsymbol{U_n^{(0)}} &:= \boldsymbol{W_0 G_n}, \\ \boldsymbol{U_n^{(\ell)}} &:= \sum_{j=b_\ell}^{b_{\ell-1}-1} \boldsymbol{W_{n-j}G_j} \qquad \text{for } \ell = 1, \cdots, L, \end{aligned} \tag{30}$$

where the b_ℓs are chosen so that:

- $b_0 = n$,
- $b_L = 0$,
- $b_i > b_{i+1}$ for $i = 0, \cdots, L-1$ and
- $n - j \in [B^{\ell-1}, 2B^{\ell} - 2]$ for $j \in [b_\ell, b_{\ell-1} - 1]$.

A pseudo-code for their computation is given on page 430 of [15] . Since the implementation of the fast convolution quadrature is not trivial, the section 4.6 is entirely dedicated to the implementation aspects of the algorithm. Since

$$[n - (b_{\ell-1} - 1), n - b_\ell] \subset [B^{\ell-1}, 2B^{\ell} - 2]$$

is equivalent to

$$[(n+1) - b_{\ell-1}, (n+1) - b_\ell] \subset [B^{\ell-1}, 2B^{\ell} - 1],$$

the splitting allows using the integral representation (27) to compute

$$\begin{aligned} \boldsymbol{U}_{\boldsymbol{n}}^{(\boldsymbol{\ell})} &:= \sum_{j=b_\ell}^{b_{\ell-1}-1} \boldsymbol{W}_{\boldsymbol{n-j}} \boldsymbol{G}_{\boldsymbol{j}} \\ &= \sum_{j=b_\ell}^{b_{\ell-1}-1} \frac{hi}{2\pi} \int_{\Gamma_\ell} K(\lambda) \boldsymbol{E}_{\boldsymbol{n-j}}(h\lambda)\, d\lambda \boldsymbol{G}_{\boldsymbol{j}}, \end{aligned}$$

where ℓ indicates the index of the countour. Note that L is the number of contours involved. Lemma 2 on page 14 provides an explicit representation

$$\boldsymbol{E}_{\boldsymbol{n}}(z) = R(z)^{n-1} (\boldsymbol{I} - z\mathfrak{A})^{-1} \mathbb{1} \boldsymbol{b}^{\boldsymbol{T}} (\boldsymbol{I} - z\mathfrak{A})^{-1} \qquad \text{for } n \geq 1.$$

By defining the row vector

$$\boldsymbol{e}_{\boldsymbol{n}}(z) := R(z)^n \boldsymbol{b}^{\boldsymbol{T}} (\boldsymbol{I} - z\mathfrak{A})^{-1} \tag{31}$$

it is found that

$$\boldsymbol{E}_{\boldsymbol{n}}(z) = R(z)^{-1} (\boldsymbol{I} - z\mathfrak{A})^{-1} \mathbb{1} \boldsymbol{e}_{\boldsymbol{n}}(z).$$

Then

$$\begin{aligned}
\boldsymbol{U}_{\boldsymbol{n}}^{(\boldsymbol{\ell})} &= \sum_{j=b_\ell}^{b_{\ell-1}-1} \frac{hi}{2\pi}\int_{\Gamma_\ell} K(\lambda)\boldsymbol{E}_{\boldsymbol{n-j}}(h\lambda)\, d\lambda \boldsymbol{G}_{\boldsymbol{j}} \\
&= \frac{hi}{2\pi}\int_{\Gamma_\ell} K(\lambda) \sum_{j=b_\ell}^{b_{\ell-1}-1} \boldsymbol{E}_{\boldsymbol{n-j}}(h\lambda)\boldsymbol{G}_{\boldsymbol{j}}\, d\lambda \\
&= \frac{hi}{2\pi}\int_{\Gamma_\ell} K(\lambda) \sum_{j=b_\ell}^{b_{\ell-1}-1} R(h\lambda)^{-1}(\boldsymbol{I}-h\lambda\mathfrak{A})^{-1}\mathbb{1}\boldsymbol{e}_{\boldsymbol{n-j}}(h\lambda)\boldsymbol{G}_{\boldsymbol{j}}\, d\lambda \\
&= \frac{hi}{2\pi}\int_{\Gamma_\ell} K(\lambda) R(h\lambda)^{-1}(\boldsymbol{I}-h\lambda\mathfrak{A})^{-1}\mathbb{1} \sum_{j=b_\ell}^{b_{\ell-1}-1} \boldsymbol{e}_{\boldsymbol{n-j}}(h\lambda)\boldsymbol{G}_{\boldsymbol{j}}\, d\lambda \\
&= \frac{hi}{2\pi}\int_{\Gamma_\ell} K(\lambda) R(h\lambda)^{-1}(\boldsymbol{I}-h\lambda\mathfrak{A})^{-1}\mathbb{1} R(h\lambda)^{n-(b_{\ell-1}-1)} \sum_{j=b_\ell}^{b_{\ell-1}-1} \boldsymbol{e}_{\boldsymbol{(b_{\ell-1}-1)-j}}(h\lambda)\boldsymbol{G}_{\boldsymbol{j}}\, d\lambda \\
&= \frac{i}{2\pi}\int_{\Gamma_\ell} K(\lambda) R(h\lambda)^{n-b_{\ell-1}}(\boldsymbol{I}-h\lambda\mathfrak{A})^{-1}\mathbb{1} \sum_{j=b_\ell}^{b_{\ell-1}-1} h\boldsymbol{e}_{\boldsymbol{(b_{\ell-1}-1)-j}}(h\lambda)\boldsymbol{G}_{\boldsymbol{j}}\, d\lambda.
\end{aligned}$$

The inner summation is nothing else than the Runge-Kutta approximation at time $t = b_{\ell-1}h$ of the initial value problem

$$y' = \lambda y + g, \qquad y(b_\ell h) = 0. \tag{32}$$

Defining

$$y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)}) := \sum_{j=b_\ell}^{b_{\ell-1}-1} h\boldsymbol{e}_{\boldsymbol{(b_{\ell-1}-1)-j}}(h\lambda)\boldsymbol{G}_{\boldsymbol{j}},$$

we have

$$\boldsymbol{U}_{\boldsymbol{n}}^{(\boldsymbol{\ell})} = \frac{i}{2\pi}\int_{\Gamma_\ell} K(\lambda) R(h\lambda)^{n-b_{\ell-1}}(\boldsymbol{I}-h\lambda\mathfrak{A})^{-1}\mathbb{1} y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)})\, d\lambda.$$

Discretizing this integral by applying the trapezoidal rule as in (28) gives

$$\boldsymbol{U}_{\boldsymbol{n}}^{(\boldsymbol{\ell})} \approx \sum_{k=-N}^{N} \omega_k^{(\ell)} K(\lambda_k^{(\ell)}) R(h\lambda_k^{(\ell)})^{n-b_{\ell-1}}(\boldsymbol{I}-h\lambda_k^{(\ell)}\mathfrak{A})^{-1}\mathbb{1} y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)}). \tag{33}$$

Thus the approximated convolution u_{n+1} is the last component of

$$\boldsymbol{W_0 G_n} + \sum_{\ell=1}^{L}\sum_{k=-N}^{N} \omega_k^{(\ell)} K(\lambda_k^{(\ell)}) R(h\lambda_k^{(\ell)})^{n-b_{\ell-1}}(\boldsymbol{I}-h\lambda_k^{(\ell)}\mathfrak{A})^{-1}\mathbb{1} y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)}).$$

We see that the computation time for computing the convolution is proportional to the one necessary for computing the solutions of the L ODE's (32) at the time nh by using the Runge-Kutta method. Since L is proportional to $\log_B(n)$, we conclude that the computation time is $\mathcal{O}(n \log n)$. Moreover the Laplace transform K is computed only $(2N+1) \cdot L$ times. In the section 4.6 we see how to rearrange the computations in order to use only $\mathcal{O}(\log n)$ active memory. Since the fast convolution quadrature is nothing else than a reorganisation of the computations of the classic convolution quadrature, it inherits the convergence property of theorem 1 on page 19.

4.5 Fast convolution quadrature for integral equation

The FCQ algorithm can also be applied to integral equations. It is only necessary to note the fact that the Runge-Kutta solver requires the values of the solution at intermediate times. Consider

$$u(t) = a(t) + \int_0^t k(t-\tau) g(\tau, u(\tau)) \qquad t \geq 0 \tag{34}$$

for given functions $a(t)$, $g(t,u)$ and a kernel $k(t)$ of which only the Laplace transform is known. Because of the above mentioned problem, it is discretized as

$$\boldsymbol{U_n} = \boldsymbol{a_n} + \sum_{j=0}^{n} \boldsymbol{W_{n-j} G_j},$$

where

- $\boldsymbol{U_n} := (u_{n1}, \cdots, u_{nm})^{\boldsymbol{T}} \approx (u(t_n + c_1 h), \cdots, u(t_n + c_m h))^{\boldsymbol{T}}$,
- $\boldsymbol{a_n} := (a(t_n + c_1 h), \cdots, a(t_n + c_m h))^{\boldsymbol{T}}$ and
- $\boldsymbol{G_j} := (g(t_j + c_1 h, u_{j1}), \cdots, g(t_j + c_m h, u_{jm}))^{\boldsymbol{T}}$.

The summation can be split as before, then

$$\boldsymbol{U_n} = \boldsymbol{a_n} + \boldsymbol{W_0 G_n} + \sum_{\ell=1}^{L} \boldsymbol{U_n^{(\ell)}}. \tag{35}$$

The terms $\boldsymbol{U_n^{(\ell)}}$ can be computed as in (33) and require only the solution of the initial value problem (32) at previous time. The resulting scheme (35) is implicit in $\boldsymbol{U_n}$ (which in the right hand side appears only in $\boldsymbol{G_n}$). The complexity of the algorithm, as well as the memory requirement (which is independent of m because $y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)})$ are scalar values) and the number of evaluations of the Laplace transform K stay the same as for computing a convolution for a given function g. This means that for an integral equation this algorithm reduces the number of required multiplication by a factor $\log n$ (with respect to the previous algorithm, which needs $\mathcal{O}(n(\log n)^2)$ multiplications).

4.6 Implementation of the FCQ

The implementation of the fast convolution quadrature algorithm is another challenge and requires some attention. Here we give a guideline for an implementation of the algorithm for solving integral equations like (34) (the algorithm for computing convolutions is very similar and therefore we omit its description).

Suppose that $t = (\tilde{n}+1)h$, then we have to solve the implicit equation (35) iteratively for $n = 0, \cdots, \tilde{n}$[8]. Once the sum on the right hand side has been computed, the implicit equation can be solved with Newton's method (for $n = 0$ no sum appears in (35) and Newton's method can be directly applied). Thus the basic pseudo-code of the algorithm is

```
solve $\boldsymbol{U_0} = \boldsymbol{a_0} + \boldsymbol{W_0}\boldsymbol{G_0}$
for $n = 1 : \tilde{n}$ do
   compute $\sum_{\ell=1}^{L} \boldsymbol{U_n^{(\ell)}}$
   solve $\boldsymbol{U_n} = \boldsymbol{a_n} + \boldsymbol{W_0}\boldsymbol{G_n} + \sum_{\ell=1}^{L} \boldsymbol{U_n^{(\ell)}}$
end for
read the solution at time $t = (\tilde{n}+1)h$ in the last component of $\boldsymbol{U_n}$.
```

The difficulties lie in the computation of the $\boldsymbol{U_n^{(\ell)}}$s. Equation (33) shows that each of these involves the solutions $y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)})$ of the initial value problem (32) at several quadrature points $\lambda_k^{(\ell)}$. Thus we have to find an economical way of computing and storing them.

First we examine the b_ℓs because they indicates the initial and the final time of the solution $y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)})$ which is required for computing $\boldsymbol{U_n^{(\ell)}}$. Their definition on page 32 indicates that the b_ℓ's depend on n (therefore we write $b_\ell(n)$, even if it complicates the notation) and are chosen so that

$$[(n+1) - b_{\ell-1}(n), (n+1) - b_\ell(n)] \subset [B^{\ell-1}, 2B^\ell - 1] = I_\ell. \qquad (36)$$

[8]Note that at the nth step we are the computing the solution at the time $t = (n+1)h$.

Moreover L too depends on n because it is defined to be the smallest integer such that $n < 2B^L$ (thus we write $L(n)$ instead of L). Let's consider an example: we choose $B = 10$ and we examine the first twenty $b_\ell(n)$s (which are given in table 3). For $n = 1, \cdots, 18$ we see that a contour (recall that $L(n)$

n	$L(n)$	$b_0(n)$	$b_1(n)$	$b_2(n)$
1	1	1	0	
2	1	2	0	
3	1	3	0	
4	1	4	0	
5	1	5	0	
6	1	6	0	
7	1	7	0	
8	1	8	0	
9	1	9	0	
10	1	10	0	
11	1	11	0	
12	1	12	0	
13	1	13	0	
14	1	14	0	
15	1	15	0	
16	1	16	0	
17	1	17	0	
18	1	18	0	
19	2	19	10	0
20	2	20	10	0

Table 3: First 20 values of $L(n)$ and $b_\ell(n)$ for B=10.

is the number of contours involved) is enough for the accurate computation of the first n weights. Indeed the first interval (for $B = 10$) is $I_1 = [1, 19]$. For $n = 19$ one contour is not enough anymore[9] and thus the second interval $I_2 = [10, 199]$ is taken into account. This is done by introducing a new value $b_2(19) = 0$ and by choosing $b_1(19)$ so that

$$(19 + 1) - b_1(19) = 10,$$

where the value 10 on the right hand side is the smallest value of I_2. The values $b_0(n) = n$, $b_1(n) = 10$ and $b_2(n) = 0$ satisfy the interval condition (36)

[9]In the 19th step we are computing the solution at time $t = 20h$.

for $n = 19, \cdots 28$. On the other hand for $n = 29$ $b_1(n)$ has to be updated because otherwise we would have an interval

$$[(n+1) - b_0(n), (n+1) - b_1(n)] = [(29+1) - 29, (29+1) - 10] = [1, 20]$$

which clearly is not contained in I_1. $b_1(29)$ is chosen with the same strategy as above, id est so that

$$(29+1) - b_1(29) = 10.$$

The strategy of augmenting $b_1(n)$ by 10 every ten n works until

$$[(n+1) - b_1(n), (n+1) - b_0(n)] = [(n+1) - b_1(n), (n+1)]$$

is contained in I_2, id est until $(n+1) < 199$. For $n = 199$ a third value $b_3(199) = 0$ and a new interval $I_3 = [100, 1999]$ have to be introduced. The strategy for $b_1(n)$ remain the same as before while similarly $b_2(199)$ is chosen so that

$$(199+1) - b_2(199) = 100,$$

where 100 is the smallest value of I_3. The values $b_1(299) = 290$ and $b_2(299) = 200$ are chosen in the same way. This strategy works until $n = 1999$, when a new value $b_4(1999) = 0$ and a new interval $I_4 = [1000, 19999]$ have to be introduced and when, for the first time, $b_3(n)$ is not equal zero ($b_3(1999) = 1000$). It should now be clear that each $b_\ell(n)$ is augmented by B^ℓ every B^ℓ steps, as we can see in the table 4 (and as pointed out on page 431 of [15]).

n	$L(n)$	$b_0(n)$	$b_1(n)$	$b_2(n)$	$b_3(n)$	$b_4(n)$
1	1	1	0			
$\vdots$	$\vdots$	$\vdots$	$\vdots$			
18	1	18	0			
19	2	19	10	0		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		
28	2	28	10	0		
29	2	29	20	0		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		
198	2	198	180	0		
199	3	199	190	100	0	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
208	3	208	190	100	0	
209	3	209	200	100	0	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
298	3	298	280	100	0	
299	3	299	290	200	0	
1998	3	1998	1980	1800	0	
1999	4	1999	1990	1900	1000	0

Table 4: Some values of $L(n)$ and $b_\ell(n)$ for B=10, note that each $b_\ell(n)$ is augmented by B^ℓ every B^ℓ steps.

In algorithm 1 we repeat the pseudo-code given on page 430 of [15] for computing the $b_\ell(n)$s.

Algorithm 1 Computation of the $b_\ell(n)$

```
$L = 1$
$q = 0$
for $m = 1 : n$ do
  if $2 \cdot B^L == m + 1$ then
    $L = L + 1$
  end if
  $k = 1$
  while $\mathrm{mod}(m + 1, B^k) == 0$ && $k < L$ do
    $q(k) = q(k) + 1$
    $k = k + 1$
  end while
  for $k = 1 : L - 1$ do
    $b(k) = q(k) \cdot B^k$
  end for
end for
```

Now we can concentrate on the values $y(b_{\ell-1}h, b_\ell h, \lambda_k^{(\ell)})$. These are necessary for the computation of $\boldsymbol{U_n^{(\ell)}}$. A first idea could be to create a struct object ***odesol*** so that ***odesol*{*n*}** contains all $y(b_{\ell-1}(n)h, b_\ell(n)h, \lambda_k^{(\ell)})$ that are necessary for computing $\boldsymbol{U_n^{(\ell)}}$. In fact it is more convenient to require that ***odesol*{*n*}** contains all $y(b_0(n)h, b_\ell(n)h, \lambda_k^{(\ell)})$ for all the contours which are and will be involved and to compute $\boldsymbol{U_n^{(\ell)}}$ with ***odesol*{$\boldsymbol{b_{\ell-1}(n)}$}**. In order to do this we set $b_\ell(n) = 0$ for all the ns for which they are not defined[10]. Now we show why and how to construct and update such a struct object.

Let's start with $n = 1$. Even if a contour would be enough for computing $\boldsymbol{U_n^{(1)}}$, we compute and store in ***odesol*{1}** the values $y(b_0(1)h, b_\ell(1)h, \lambda_k^{(\ell)})$ for $\ell = 1, \cdots, L(\tilde{n})$[11]. These values can be computed just by using $\boldsymbol{U_0}$, because

$$
\begin{aligned}
y(b_0(1)h, b_\ell(1)h, \lambda) &= \sum_{j=b_\ell(1)}^{b_0(1)-1} h\boldsymbol{e}_{\boldsymbol{(b_0(1)-1)-j}}(h\lambda)\boldsymbol{G_j} \\
&= h\boldsymbol{e_0}(h\lambda)\boldsymbol{G_0}.
\end{aligned}
$$

[10]For example we set $b_2(1) = 0$.

[11]Note that $L(\tilde{n})$ is the highest number of contours involved for computing (34) at the end time $t = (\tilde{n} + 1)h$.

The next proposition shows that, except for some particular n, , the values of the ℓ^{th} contour contained in $\boldsymbol{odesol\{n\}}$ can be computed only involving $\boldsymbol{odesol\{n-1\}}$ and $\boldsymbol{U_{n-1}}$.

Proposition 2. *Whenever $b_\ell(n) = b_\ell(n-1)$ for an $\ell > 1$, the values of $\boldsymbol{odesol\{n\}}$ related to the ℓ^{th} contour can be computed with only $\boldsymbol{odesol\{n-1\}}$ and $\boldsymbol{U_{n-1}}$.*

Proof. For all n we have $b_0(n) = b_0(n-1) + 1$, which with formula (31) implies

$$\boldsymbol{e_{(b_0(n)-1)-j}} = \boldsymbol{e_{((b_0(n-1)-1)-j)+1}} = R(h\lambda)\boldsymbol{e_{(b_0(n-1)-1)-j}}.$$

Thus

$$\begin{aligned}
y(b_0(n)h, b_\ell(n)h, \lambda) &= \sum_{j=b_\ell(n)}^{b_0(n)-1} h\boldsymbol{e_{(b_0(n)-1)-j}}(h\lambda)\boldsymbol{G_j} \\
&= \sum_{j=b_\ell(n-1)}^{(b_0(n-1)+1)-1} h\boldsymbol{e_{(b_0(n)-1)-j}}(h\lambda)\boldsymbol{G_j} \\
&= \sum_{j=b_\ell(n-1)}^{b_0(n-1)-1} h\boldsymbol{e_{(b_0(n)-1)-j}}(h\lambda)\boldsymbol{G_j} + h\boldsymbol{e_0}(h\lambda)\boldsymbol{G_{b_0(n)-1}} \\
&= R(h\lambda)y(b_0(n-1)h, b_\ell(n-1)h, \lambda) + h\boldsymbol{e_0}(h\lambda)\boldsymbol{G_{b_0(n)-1}}.
\end{aligned}$$

□

For example the values $y(b_0(n)h, b_\ell(n)h, \lambda_k^{(1)})$ can be computed as in proposition 2 for $n = 2, \cdots, 18$ or for $n = 20, \cdots, 28$ or $n = 30, \cdots, 38$ and so on, while for $y(b_0(n)h, b_\ell(n)h, \lambda_k^{(2)})$ proposition 2 can be applied for $n = 2, \cdots, 198$ and for $n = 200, \cdots, 298$ and so on. Therefore proposition 2 is the main tool to update $\boldsymbol{odesol}$.

Now we explain why, when computing $\boldsymbol{odesol\{1\}}$, we don't compute just the values of $y(b_0(1)h, b_\ell(1)h, \lambda_k^{(1)})$. Suppose we have done so. For $n = 2, \cdots, 18$, $L(n) = 1$ and thus we have to compute only $\boldsymbol{U_n^{(1)}}$, which requires only the values of the first contour. But for $n = 19$ we have $L(19) = 2$, and thus, to compute $\boldsymbol{U_{19}^{(\ell)}}$s we need the values $y(b_1(19)h, b_2(19)h, \lambda_k^{(2)})$ too (which are used to compute $\boldsymbol{U_{19}^{(2)}}$). Computing them at this time would require the employment of $\boldsymbol{U_n}$ for $n = 0, \cdots, 10$ and thus, in general, the storage of all values of $\boldsymbol{U_n}$s. Instead, computing and storing them for every step from the beginnig solves this problem. We also notice that for every ℓ the values necessary for computing $\boldsymbol{U_n^{(\ell)}}$ are almost always stored in $\boldsymbol{odesol\{b_{\ell-1}(n)\}}$ (we will discuss this on page 43 and provide a solution).

The last thing to consider is how to compute $\boldsymbol{odesol\{n\}}$ when proposition 2 don't apply. For example when $n = 19$ $b_1(19) = 10$ and thus $\boldsymbol{odesol\{19\}}$ must contain the values $y(19h, 10h, \lambda_k^{(1)})$. So far, these can't be obtained directly from the ones contained in $\boldsymbol{odesol\{18\}}$. The problem can be solved by making of $\boldsymbol{odesol\{n\}}$ a nested cell, which contains $\boldsymbol{odesol\{n\}\{1\}}$ (in which we store and update $y(b_0(n)h, b_\ell(n)h, \lambda_k^{(\ell)})$ for all the contours and which will be used for computing the $\boldsymbol{U_n^{(\ell)}}$s) and $\boldsymbol{odesol\{n\}\{2\}}$ (in which we pre-compute and store the solution of the initial value problem with the new initial time[12]). Then, when computing $\boldsymbol{odesol\{19\}}$, we first overwrite the values related to the first contour in $\boldsymbol{odesol\{18\}\{1\}}$ with those contained in $\boldsymbol{odesol\{18\}\{2\}}$ and then we compute $\boldsymbol{odesol\{19\}\{1\}}$ with $\boldsymbol{odesol\{18\}\{1\}}$ and $\boldsymbol{U_{18}}$. At this point the values related to the first contour contained in $\boldsymbol{odesol\{18\}\{2\}}$ can be canceled. When $n = 21$ we start computing and storing in $\boldsymbol{odesol\{21\}\{2\}}$ the ones with initial time $20h$. Table 5 shows the contents of $\boldsymbol{odesol\{n\}}$ which are related to the first contour.

n	1	$\cdots$	10
$\boldsymbol{odesol\{n\}\{1\}}$	$y(h, 0, \lambda_k^{(1)})$	$\cdots$	$y(10h, 0, \lambda_k^{(1)})$
$\boldsymbol{odesol\{n\}\{2\}}$	empty	$\cdots$	empty

n	11	$\cdots$	18
$\boldsymbol{odesol\{n\}\{1\}}$	$y(11h, 0, \lambda_k^{(1)})$	$\cdots$	$y(18h, 0, \lambda_k^{(1)})$
$\boldsymbol{odesol\{n\}\{2\}}$	$y(11h, 10h, \lambda_k^{(1)})$	$\cdots$	$y(18h, 10h, \lambda_k^{(1)})$

n	19	20	21
$\boldsymbol{odesol\{n\}\{1\}}$	$y(19h, 10, \lambda_k^{(1)})$	$y(20h, 10, \lambda_k^{(1)})$	$y(21h, 10, \lambda_k^{(1)})$
$\boldsymbol{odesol\{n\}\{2\}}$	empty	empty	$y(21h, 20h, \lambda_k^{(1)})$

Table 5: Contents of $\boldsymbol{odesol\{n\}}$ related to the first contour for $n = 1, \cdots, 21$.

[12]For example from $n = 11$ we compute and store the values $y(b_0(n)h, 10h, \lambda_k^{(1)})$.

The second contour is handled in a similar fashion when $n = 199$. Indeed $b_2(199) = 100$, which implies that $\boldsymbol{odesol\{199\}}$ must contain the values $y(199h, 100h, \lambda_k^{(2)})$. Therefore when $n = 101$ we start storing them in $\boldsymbol{odesol\{n\}\{2\}}$. Table 6 shows the contents of $\boldsymbol{odesol\{n\}}$ which are related to the first three contours for $n = 1, \cdots, 299$.

In general the values of the ℓ^{th} contour are being computed and stored in $\boldsymbol{odesol\{n\}\{2\}}$ since $n \in \{\mathbb{N} \cdot B^{\ell} + 1\}$ and overwritten in $\boldsymbol{odesol\{n\}\{1\}}$ when $n \in \{(2 + \mathbb{N}) \cdot B^{\ell} - 1\}$.

In order to compute $\boldsymbol{U_n^{(\ell)}}$ only with the values stored in $\boldsymbol{odesol\{b_{\ell-1}(n)\}}$, we also have to overwrite $\boldsymbol{odesol\{b_{\ell}(n)\}\{1\}}$ with $\boldsymbol{odesol\{b_{\ell}(n)\}\{2\}}$ for $\ell > 1$ because, to compute $\boldsymbol{U_n^{(\ell)}}$ we need the values $y(b_{\ell-1}(n)h, b_{\ell}(n)h, \lambda_k^{(\ell)})$ while $\boldsymbol{odesol\{b_{\ell}(n)\}\{1\}}$ contains $y(b_{\ell-1}(n)h, b_{\ell}(b_{\ell}(n))h, \lambda_k^{(\ell)})$. If we consider for example $n = 199$, to compute $\boldsymbol{U_{199}^{(2)}}$ we need $y(190h, 100h, \lambda_k^{(\ell)})$ because $b_1(199) = 190$ and $b_2(199) = 100$, while, if we don't overwrite $\boldsymbol{odesol\{199\}\{1\}}$, it would contain $y(190h, 0, \lambda_k^{(\ell)})$ because $b_2(b_1(199)) = b_2(190) = 0$ (see table 6).

n	1	$\cdots$	11	$\cdots$	18	19	20	21	$\cdots$
$\boldsymbol{odesol\{n\}\{1\}}$	$y(h,0,\lambda_k^{(1)})$ $y(h,0,\lambda_k^{(2)})$ $y(h,0,\lambda_k^{(3)})$	$\cdots$ $\cdots$ $\cdots$	$y(11h,0,\lambda_k^{(1)})$ $y(11h,0,\lambda_k^{(2)})$ $y(11h,0,\lambda_k^{(2)})$	$\cdots$ $\cdots$ $\cdots$	$y(18h,0,\lambda_k^{(1)})$ $y(18h,0,\lambda_k^{(2)})$ $y(18h,0,\lambda_k^{(3)})$	$y(19h,10h,\lambda_k^{(1)})$ $y(19h,0,\lambda_k^{(2)})$ $y(19h,0,\lambda_k^{(3)})$	$\cdots$ $\cdots$ $\cdots$	$y(21h,10h,\lambda_k^{(1)})$ $y(21h,0,\lambda_k^{(2)})$ $y(21h,0,\lambda_k^{(3)})$	$\cdots$ $\cdots$ $\cdots$
$\boldsymbol{odesol\{n\}\{2\}}$	empty	$\cdots$	$y(11h,10h,\lambda_k^{(1)})$	$\cdots$	$y(18h,10h,\lambda_k^{(1)})$	empty	$\cdots$	$y(21h,20h,\lambda_k^{(1)})$	$\cdots$

n	28	$\cdots$	101	$\cdots$	198	199	200
$\boldsymbol{odesol\{n\}\{1\}}$	$y(28h,10h,\lambda_k^{(1)})$ $y(28h,0,\lambda_k^{(2)})$ $y(28h,0,\lambda_k^{(3)})$	$\cdots$ $\cdots$ $\cdots$	$y(101h,90h,\lambda_k^{(1)})$ $y(101h,0,\lambda_k^{(2)})$ $y(101h,0,\lambda_k^{(3)})$	$\cdots$ $\cdots$ $\cdots$	$y(198h,180h,\lambda_k^{(1)})$ $y(198h,0,\lambda_k^{(2)})$ $y(198h,0,\lambda_k^{(3)})$	$y(199h,190h,\lambda_k^{(1)})$ $y(199h,100,\lambda_k^{(2)})$ $y(199h,0,\lambda_k^{(3)})$	$y(200h,190h,\lambda_k^{(1)})$ $y(200h,100,\lambda_k^{(2)})$ $y(200h,0,\lambda_k^{(3)})$
$\boldsymbol{odesol\{n\}\{2\}}$	$y(28h,20h,\lambda_k^{(1)})$	$\cdots$	$y(101h,100h,\lambda_k^{(1)})$ $y(101h,100h,\lambda_k^{(2)})$	$\cdots$ $\cdots$	$y(198h,190h,\lambda_k^{(1)})$ $y(198h,100h,\lambda_k^{(2)})$	empty	empty

n	201	$\cdots$	298	299
$\boldsymbol{odesol\{n\}\{1\}}$	$y(201h,190h,\lambda_k^{(1)})$ $y(201h,100,\lambda_k^{(2)})$ $y(201h,0,\lambda_k^{(3)})$	$\cdots$ $\cdots$ $\cdots$	$y(298h,280h,\lambda_k^{(1)})$ $y(298h,100,\lambda_k^{(2)})$ $y(298h,0,\lambda_k^{(3)})$	$y(299h,290h,\lambda_k^{(1)})$ $y(299h,200,\lambda_k^{(2)})$ $y(299h,0,\lambda_k^{(3)})$
$\boldsymbol{odesol\{n\}\{2\}}$	$y(201h,200h,\lambda_k^{(1)})$ $y(201h,200h,\lambda_k^{(2)})$	$\cdots$ $\cdots$	$y(298h,290h,\lambda_k^{(1)})$ $y(298h,200h,\lambda_k^{(2)})$	empty

Table 6: Contents of $\boldsymbol{odesol\{n\}}$ related to the first three contours for $n = 1, \cdots, 299$. We omitted the overwriting of $\boldsymbol{odesol\{b_\ell(n)\}\{1\}}$ with $\boldsymbol{odesol\{b_\ell(n)\}\{2\}}$.

Now we can finally write a complete pseudo-code of the FCQ (k_1 and k_2 are used for finding when $n \in \{\mathbb{N} \cdot B^\ell + 1\}$ and when $n \in \{(2 + \mathbb{N}) \cdot B^\ell - 1\}$).

Algorithm 2 FCQ

```
 1: precompute all the $b_\ell(n)$s with algorithm 1
 2: solve $\boldsymbol{U_0} = \boldsymbol{a_0} + \boldsymbol{W_0 G_0}$
 3: compute $\boldsymbol{odesol\{1\}}$
 4: $k_1 = 0$
 5: $k_2 = 1$
 6: for $n = 1 : \tilde{n}$ do
 7:    compute $\sum_{\ell=1}^{L} \boldsymbol{U_n^{(\ell)}}$ with $\boldsymbol{odesol\{b_\ell(n)\}\{1\}}$
 8:    solve $\boldsymbol{U_n} = \boldsymbol{a_n} + \boldsymbol{W_0 G_n} + \sum_{\ell=1}^{L} \boldsymbol{U_n^{(\ell)}}$
 9:    update $\boldsymbol{odesol}$ and $k_1$, $k_2$ with algorithm 3
10: end for
11: read the solution at time $t = (\tilde{n} + 1)h$ in the last component of $\boldsymbol{U_n}$.
```

Algorithm 3 update ***odesol***

```
for $\ell = 1 : L(\tilde{n})$ do
   if $n + 1 = (2 + k_1(\ell)) \cdot B^\ell - 1$ then
      if $\ell > 1$ then
         overwrite $\boldsymbol{odesol\{b_\ell(n)\}\{1\}}$ with $\boldsymbol{odesol\{b_\ell(n)\}\{2\}}$
      end if
      overwrite $\boldsymbol{odesol\{n\}\{1\}}$ with $\boldsymbol{odesol\{n\}\{2\}}$
      $k_1(\ell) = k_1(\ell) + 1$
   end if
   compute $\boldsymbol{odesol\{n+1\}}$ with $\boldsymbol{odesol\{n\}}$ and $\boldsymbol{U_n}$
   if $n + 1 = k_2(\ell) \cdot B^\ell + 1$ then
      restart $\boldsymbol{odesol\{n+1\}\{2\}}$
      $k_2(\ell) = k_2(\ell) + 1$
   end if
end for
```

Remark 6. *The active memory requirement of the algorithm is only $\mathcal{O}(\log n)$ because* $\boldsymbol{odesol\{n\}}$ *can be computed iteratively with just* $\boldsymbol{odesol\{n-1\}}$ *and* $\boldsymbol{U_{n-1}}$.

4.7 Numerical example of an integral equation

We construct an integral equation of the form

$$y(t) = H(t) - \int_0^t k(t-\tau)y(\tau)d\tau$$

by assuming that the analytic solution is

$$y(t) := \sqrt{\pi}t^{7/2}$$

and that the convolution kernel is

$$k(t) := \frac{1}{\sqrt{\pi t}},$$

whose Laplace transform is

$$K(s) = \frac{1}{\sqrt{s}}.$$

By defining the function

$$H(t) := \int_0^t k(t-\tau)y(\tau)d\tau + y(t)$$

we find[13]

$$\begin{aligned} H(t) &= \mathcal{L}^{-1}\big(\mathcal{L}(k)\mathcal{L}(y)\big)(t) + \sqrt{\pi}t^{7/2} \\ &= \mathcal{L}^{-1}\Big(\frac{1}{\sqrt{s}}\frac{105\pi}{16s^{9/2}}\Big)(t) + \sqrt{\pi}t^{7/2} \\ &= \frac{105\pi}{16}\mathcal{L}^{-1}\Big(\frac{1}{s^5}\Big)(t) + \sqrt{\pi}t^{7/2} \\ &= \frac{35\pi}{128}t^4 + \sqrt{\pi}t^{7/2}. \end{aligned}$$

Thus $y(t) := \sqrt{\pi}t^{7/2}$ is the analytic solution of the linear Volterra integral equation of second kind

$$y(t) = \frac{35\pi}{128}t^4 + \sqrt{\pi}t^{7/2} - \int_0^t \frac{1}{\sqrt{\pi(t-\tau)}}y(\tau)d\tau.$$

[13]We use the convolution property for the Laplace transform, see appendix A.1.

This integral equation is used to test the fast algorithm described above for the implicit Euler method and the 2-stage RadauIIA method. We perform five time step refinements starting with the time step $h = 0.5$ and we compute the relative error between the approximated and the analytic solutions at time $t = 4$, by choosing as fixed parameters $B = 10$ and $K = 15$. The results can be found in table 7 and in table 8.

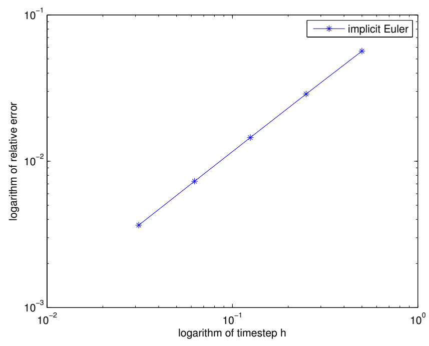
time step	Relative error
2^{-1}	0.0566
2^{-2}	0.0288
2^{-3}	0.0145
2^{-4}	0.0073
2^{-5}	0.0037

Table 7: Relative error of implicit Euler method at time $t = 4$ versus time step h.

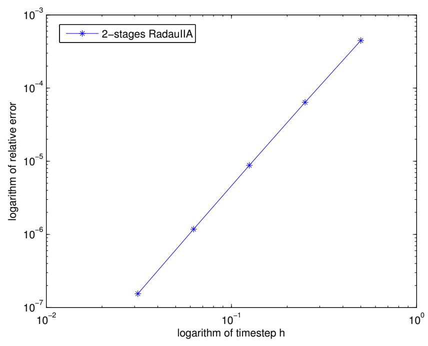
time step	Relative error
2^{-1}	$0.4471 \cdot 10^{-3}$
2^{-2}	$0.0638 \cdot 10^{-3}$
2^{-3}	$0.0088 \cdot 10^{-3}$
2^{-4}	$0.0012 \cdot 10^{-3}$
2^{-5}	$0.0002 \cdot 10^{-3}$

Table 8: Relative error of the 2-stage RadauIIA method at time $t = 4$ versus time step h.

As expected,the FCQ based on the implicit Euler method has an algebraic convergence of (approximated) order 0.988555, as we can see in figure 7. Regarding the 2-stage RadauIIA method an approximated algebraic convergence of order 2.875043 is obtained (we expect an algebraic convergence of order 3, see figure 7 for a plot of the absolute error versus the time step). The MATLAB implementation can be found in the appendix B.3.



(a) Implicit Euler



(b) 2-stage RadauIIA

Figure 7: Relative error of fast convolution quadrature for integral equations at time $T = 4$ versus time step h in double-logarithmic scale. We observe algebraic convergence.

4.8 Fast convolution quadrature for integral equation with derivative

Since the convolution is differentiated in the variational formulation (10), it is useful to discuss how to approach an integral equation of the form

$$y(t) = H(t) - \frac{d}{dt}\int_0^t k(t-\tau)y(\tau)d\tau. \tag{37}$$

The idea is to approximate the derivative with a backward difference method with the same order of convergence as the convolution quadrature. Backward difference methods approximate the derivative of a function u at time t using a liner combination of the values of u at previous times. By denoting the derivative of the function u as $u' := \frac{d}{dt}u$ the general formula for backward difference methods reads

$$hu'(t) \approx \sum_{j=0}^{p} \alpha_j u(t-jh), \tag{38}$$

for some $p > 0$, $h > 0$ and $\alpha_j \in \mathbb{R}$. By choosing the α_j's as the solutions of the following linear system

$$h * \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & -h & -2h & \cdots & -ph \\ 0 & \frac{(-h)^2}{2} & \frac{(-2h)^2}{2} & \cdots & \frac{(-ph)^2}{2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \frac{(-h)^p}{p!} & \frac{(-2h)^p}{p!} & \cdots & \frac{(-ph)^p}{p!} \end{pmatrix} \cdot \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_p \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{39}$$

and if u is smooth enough, formula (38) leads to an algebraic order of convergence $p-1$. Thus our strategy to approach an integral equations like (37) is to:

- choose a Runge-Kutta based fast convolution quadrature of order q (where in general $q \in \mathbb{Q}$),
- choose a backward differentiation method of integer order $p \geq q$,
- discretize the equation (37) as

$$y(t) = H(t) - \sum_{j=0}^{p} \frac{\alpha_j}{h} \int_0^{t-jh} k(t - jh - \tau) y(\tau) d\tau, \tag{40}$$

Then, by setting

$$\tilde{H}(t) := H(t) - \sum_{j=1}^{p} \frac{\alpha_j}{h} \int_0^{t-jh} k(t - jh - \tau) y(\tau) d\tau,$$

equation (40) can be written as

$$y(t) = \tilde{H}(t) - \frac{\alpha_0}{h} \int_0^{t} k(t - \tau) y(\tau) d\tau. \tag{41}$$

Equation (41) can be solved with the fast convolution quadrature algorithm explained in section 4.5, because $\tilde{H}(t)$ contains only values of the convolution in the past.

4.9 Numerical example of an integral equation with derivative

We construct an integral equation like (37) assuming that its analytic solution is

$$y(t) := \sqrt{\pi} t^{7/2}.$$

The convolution kernel is chosen as

$$k(t) := \frac{1}{\sqrt{\pi t}},$$

whose Laplace transform is

$$K(s) = \frac{1}{\sqrt{s}}.$$

$H(t)$ is computed similarly as in section 4.7.

$$\begin{aligned} H(t) &:= \frac{d}{dt}\int_0^t k(t-\tau)y(\tau)d\tau + y(t) \\ &= \frac{d}{dt}\left(\frac{35\pi}{128}t^4\right) + \sqrt{\pi}t^{7/2} \\ &= \frac{35\pi}{32}t^3 + \sqrt{\pi}t^{7/2}. \end{aligned}$$

Thus $y(t)$ is the analytic solution of

$$y(t) = \frac{35\pi}{32}t^3 + \sqrt{\pi}t^{7/2} - \frac{d}{dt}\int_0^t \frac{1}{\sqrt{\pi(t-\tau)}}y(\tau)d\tau.$$

As a first numerical example, we choose the implicit Euler based fast convolution quadrature. Its order of convergence is 1 and thus, a backward difference method with $p = 2$ should be good enough in order to preserve the rate of convergence. This is confirmed by the experiment, where an approximate order of convergence equal 0.990007 is found. The relative errors can be read in the table 9 while a plot can be found in figure 8.

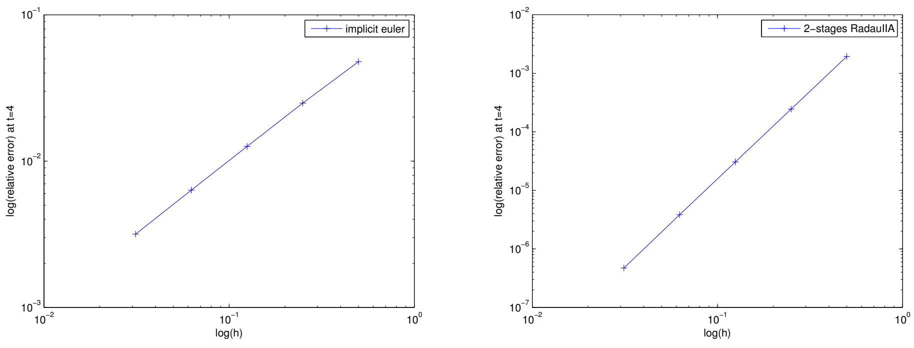
time step	Relative error
2^{-1}	0.0493
2^{-2}	0.0250
2^{-3}	0.0126
2^{-4}	0.0063
2^{-5}	0.0032

Table 9: Relative error of implicit Euler method at time $t = 4$ versus time step h.

For a second numerical example, we choose the 2-stage RadauIIA based fast convolution quadrature. Its order of convergence is 3 and thus, in order to preserve the convergence rate, a backward difference method with $p = 4$ should be chosen. This is confirmed by the actual convergence of 2.998812. The values of the relative error can be found in table 10 while a plot is contained in figure 8. The code of both examples are contained in the appendix B.4.

time step	Relative error
2^{-1}	$1.9271 \cdot 10^{-3}$
2^{-2}	$0.2438 \cdot 10^{-3}$
2^{-3}	$0.0306 \cdot 10^{-3}$
2^{-4}	$0.0038 \cdot 10^{-3}$
2^{-5}	$0.0004 \cdot 10^{-3}$

Table 10: Relative error of the 2-stage RadauIIA method at time $t = 4$ versus time step h.



(a) Implicit Euler (b) 2-stage RadauIIA

Figure 8: Relative error of fast convolution quadrature for integral equations with derivative at time $T = 4$ versus time step h in double-logarithmic scale. We observe algebraic convergence.

5 FEM - FCQ coupling

5.1 Discretization of exterior BVP

The goal is to discretize the variational form (10). We start with a spatial semi-discretization over a simplicial mesh. Let N be the number of nodes of the mesh and $\{b_N^j | j = 1, \cdots N\}$ be the set of hat functions. The solution is approximated through a linear time-dependent combination of the hat functions

$$u(x,t) \approx \sum_{i=1}^{N} \mu^i(t) b_N^i.$$

Then the semi-discrete evolution problem reads to find the (time-dependent) coefficients $\mu_i(t)$ such that

$$\sum_{i=1}^{N} \mu^i(t) \int_{\Omega\setminus\Omega_0} \boldsymbol{\nabla} b_N^i \cdot \boldsymbol{\nabla} b_N^j \, dx - \sum_{i=1}^{N} \frac{d}{dt} \int_0^t k(t-\tau) \cdot \mu^i(\tau) d\tau \int_{\partial\Omega_0} b_N^i \cdot b_N^j \, dS \\ = \int_{\Omega\setminus\Omega_0} f \cdot b_N^j \, dx \tag{42}$$

holds for every $j = 1, \cdots, N$ and for every $t \in]0, T]$. By defining

$$\boldsymbol{\mu}(t) := \left(\mu^1(t), \cdots, \mu^N(t)\right)^{\boldsymbol{T}},$$

$$\mathbf{A} := \Big(\int_{\Omega\setminus\Omega_0} \boldsymbol{\nabla} b_N^i \cdot \boldsymbol{\nabla} b_N^j \, dx\Big)_{i,j=1}^{N}, \tag{43}$$

$$\mathbf{B} := \Big(\int_{\partial\Omega_0} b_N^i \cdot b_N^j \, dS\Big)_{i,j=1}^{N}, \tag{44}$$

$$\boldsymbol{\varphi}(t) := \Big(\int_{\Omega\setminus\Omega_0} f \cdot b_N^1 \, dx, \cdots, \int_{\Omega\setminus\Omega_0} f \cdot b_N^N \, dx\Big)^{\boldsymbol{T}},$$

equation (42) becomes

$$\mathbf{A} \cdot \boldsymbol{\mu}(t) - \mathbf{B} \cdot \frac{d}{dt} \int_0^t k(t-\tau) \cdot \boldsymbol{\mu}(\tau) d\tau = \boldsymbol{\varphi}(t) \qquad \text{for } t \in]0, T]. \tag{45}$$

The derivative is approximated by using a backward difference method[14]. Let h_t be the time step (so that $t = h_t(n+1)$), then

$$\frac{d}{dt} \int_0^t k(t-\tau) \cdot \boldsymbol{\mu}(\tau) d\tau \approx \frac{1}{h_t} \sum_{l=0}^{p} \alpha_l \int_0^{t-lh_t} k(t - lh_t - \tau) \cdot \boldsymbol{\mu}(\tau) d\tau. \tag{46}$$

[14]BDF methods were introduced on page 48.

The next step is to compute the summands with the convolution quadrature algorithm (23). In order to do this let us introduce some notation. We define $\tilde{\boldsymbol{\mu}}(t)$ to be an $(N \cdot m)$-vector (where m is the number of stages of the Runge-Kutta scheme)

$$\tilde{\boldsymbol{\mu}}(t) := \big(\boldsymbol{\mu}(t + c_1 h_t), \cdots, \boldsymbol{\mu}(t + c_m h_t)\big)^{\boldsymbol{T}}.$$

In order to apply the convolution quadrature, we recall the definition of the Kronecker product.

Definition 7 (Kronecker product)**.** *Let $\boldsymbol{A}$ be an $m \times n$-matrix and $\boldsymbol{B}$ be a $q \times p$-matrix, then the Kronecker product between $\boldsymbol{A}$ and $\boldsymbol{B}$ is defined as*

$$\begin{aligned}
\boldsymbol{A} \otimes \boldsymbol{B} &= \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & & \vdots \\ b_{q1} & \cdots & b_{qp} \end{pmatrix} \\
&= \begin{pmatrix} a_{11}\boldsymbol{B} & \cdots & a_{1n}\boldsymbol{B} \\ \vdots & & \vdots \\ a_{m1}\boldsymbol{B} & \cdots & a_{mn}\boldsymbol{B} \end{pmatrix} \\
&= \begin{pmatrix} a_{11}b_{11} & \cdots & a_{11}b_{1p} & \cdots & \cdots & a_{1n}b_{11} & \cdots & a_{1n}b_{1p} \\ \vdots & & \vdots & & & \vdots & & \vdots \\ a_{11}b_{q1} & \cdots & a_{11}b_{qp} & \cdots & \cdots & a_{1n}b_{q1} & \cdots & a_{1n}b_{qp} \\ \vdots & & \vdots & & & \vdots & & \vdots \\ \vdots & & \vdots & & & \vdots & & \vdots \\ a_{m1}b_{11} & \cdots & a_{m1}b_{1p} & \cdots & \cdots & a_{mn}b_{11} & \cdots & a_{mn}b_{1p} \\ \vdots & & \vdots & & & \vdots & & \vdots \\ a_{m1}b_{q1} & \cdots & a_{m1}b_{qp} & \cdots & \cdots & a_{mn}b_{q1} & \cdots & a_{mn}b_{qp} \end{pmatrix}
\end{aligned}$$

With $\tilde{\boldsymbol{\mu}}$ and the Kronecker product, equation (45) can be rewritten as

$$\big(\mathbb{1}(m) \otimes \mathbf{A}\big) \cdot \tilde{\boldsymbol{\mu}}(t) - \big(\mathbb{1}(m) \otimes \mathbf{B}\big) \frac{d}{dt} \int_0^t k(t-\tau) \cdot \tilde{\boldsymbol{\mu}}(\tau) d\tau = \tilde{\boldsymbol{\varphi}}(t) \qquad \text{for } t \in [0, T - h_t], \tag{47}$$

where $\mathbb{1}(m)$ is the $m \times m$ identity matrix and

$$\tilde{\boldsymbol{\varphi}}(t) := \big(\boldsymbol{\varphi}(t + c_1 h_t), \cdots, \boldsymbol{\varphi}(t + c_m h_t)\big)^{\boldsymbol{T}}.$$

Remark 7. *There is an abuse of notation in equation* (47)*. With the convolution*

$$\int_0^t k(t - \tau) \cdot \tilde{\boldsymbol{\mu}}(\tau) d\tau \tag{48}$$

we mean that the convolution is computed on each component of the vector $\tilde{\boldsymbol{\mu}}(\tau)$ *and for an adjusted time* t. *To be more precise, instead of* (48) *we shoud write the vector*

$$\left(\int_0^{t+c_1h_t} k(t+c_1h_t-\tau)\cdot\boldsymbol{\mu}(\tau)d\tau,\cdots,\int_0^{t+c_mh_t} k(t+c_mh_t-\tau)\cdot\boldsymbol{\mu}(\tau)d\tau\right)^T,$$

where

$$\int_0^{t+c_ih_t} k(t+c_ih_t-\tau)\cdot\boldsymbol{\mu}(\tau)d\tau$$

$$:=\left(\int_0^{t+c_ih_t} k(t+c_ih_t-\tau)\cdot\mu^1(\tau)d\tau,\cdots,\int_0^{t+c_ih_t} k(t+c_ih_t-\tau)\cdot\mu^N(\tau)d\tau\right)^T$$

for $i=1,\cdots,m$.

Now we can apply the convolution quadrature to (46); by substituting $\boldsymbol{\mu}$ with $\tilde{\boldsymbol{\mu}}$ and with the time discretization

$$\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}\approx\tilde{\boldsymbol{\mu}}(jh_t)$$

we have

$$\frac{d}{dt}\int_0^t k(t-\tau)\cdot\tilde{\boldsymbol{\mu}}(\tau)d\tau \;\approx\; \frac{1}{h_t}\sum_{l=0}^{p}\alpha_l\sum_{j=0}^{n-l}\big(\boldsymbol{W}_{\boldsymbol{(n-l)-j}}\otimes\mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}.$$

By setting

$$\tilde{\boldsymbol{\varphi}}_i:=\tilde{\boldsymbol{\varphi}}(ih_t),$$

equation (47) becomes the fully discrete problem

$$\big(\mathbb{1}(m)\otimes\mathbf{A}\big)\cdot\tilde{\boldsymbol{\mu}}_{\boldsymbol{i}}-\big(\mathbb{1}(m)\otimes\mathbf{B}\big)\left(\frac{1}{h_t}\sum_{l=0}^{p}\alpha_l\sum_{j=0}^{i-l}\big(\boldsymbol{W}_{\boldsymbol{(i-l)-j}}\otimes\mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}\right)=\tilde{\boldsymbol{\varphi}}_{\boldsymbol{i}} \tag{49}$$

for $i=0,..,n$, where n satisfies $(n+1)h_t=T$.
Equation (49) can be rewritten as

$$\left(\mathbb{1}(m)\otimes\mathbf{A}-\frac{\alpha_0}{h_t}\big(\mathbb{1}(m)\otimes\mathbf{B}\big)\big(\boldsymbol{W}_{\boldsymbol{0}}\otimes\mathbb{1}(N)\big)\right)\cdot\tilde{\boldsymbol{\mu}}_{\boldsymbol{i}}$$

$$=\big(\mathbb{1}(m)\otimes\mathbf{B}\big)\left(\frac{\alpha_0}{h_t}\sum_{j=0}^{i-1}\big(\boldsymbol{W}_{\boldsymbol{i-j}}\otimes\mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}\right.$$

$$\left.+\frac{1}{h_t}\sum_{l=1}^{p}\alpha_l\sum_{j=0}^{i-l}\big(\boldsymbol{W}_{\boldsymbol{(i-l)-j}}\otimes\mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}\right)+\tilde{\boldsymbol{\varphi}}_{\boldsymbol{i}} \tag{50}$$

for $i = 0, \cdots, n$. The sums on the right hand side only involve values of $\boldsymbol{\mu_j}$ with $j < i$. Thus they can be rearranged as in (30) and computed separately in the fast way described in (33).

Remark 8. *In the variational formulation* (10) *the convolution is taken only on the boundary of the conductor. This is inherited in the full-discrete problem* (49) *by the multiplication of the convolution quadrature with* $\big(\mathbb{1}(m)\otimes\mathbf{B}\big)$*, which sets to zero the convolutions computed on the nodes which don't belong to the boundary of the conductor. Therefore there is no need to compute them in the two sums of* (50).

5.2 Implementation and complexity analysis

The algorithm can be implemented in a direct way. First of all we must parametrize the domain Ω and create a mesh. Then we have to precompute the matrices $\boldsymbol{A}$, $\boldsymbol{B}$[15], the matrix $\boldsymbol{W_0}$[16] and the backward differentiation weights (which are chosen in order to preserve the order of convergence of the convolution quadrature and can be computed by solving the linear system (39)). Lastly we have to create the structure ***odesol*** which will contain the approximation of the convolutions on the impedance nodes. At this point we only have to choose a time step h_t and solve the linear system (50) for increasing i until we reach the end time T (we assume that $(n+1)h_t = T$) .

When $i = 0$ there are no sums on the right hand side of (50). For $i \geq 1$ we have to compute

$$\sum_{j=0}^{i-1} \big(\boldsymbol{W_{i-j}} \otimes \mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}. \tag{51}$$

This is done by splitting the sum as in (30) and by computing each summand as in (33), with an exception for the term $\boldsymbol{U_n^{(1)}}$ for which the convolution weights are computed as in remark 2 on page 21 because the hyperbola approximation is not effective (see remark 5 on page 31).

For $i \geq 1$ the sum

$$\sum_{l=1}^{p} \alpha_l \sum_{j=0}^{i-l} \big(\boldsymbol{W_{(i-l)-j}} \otimes \mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}} \tag{52}$$

[15]Those matrices were defined respectively in the equations (43) and (44).

[16]The matrix $\boldsymbol{W_0}$ can be computed, for a chosen Runge-Kutta method, as explained in remark 2 on page 21.

appears as well, where the exterior sum is taken only until $\min(p, i)$. The interior sum of (52) is computed by storing the values (51) and by adding to them the term $\big(\boldsymbol{W_0} \otimes \mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{i-l}}$.

The algorithm is summarized in the pseudo-code 4.

Algorithm 4 FEM-FCQ coupling algorithm

```
 1: create a spatial mesh
 2: compute $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{W_0}, \cdots, \boldsymbol{W_{2B-1}}$ and $(\alpha_j)_{j=0}^{p}$
 3: initialize $\boldsymbol{odesol}$
 4: solve $\big(\mathbb{1}(m) \otimes \mathbf{A} - \frac{\alpha_0}{h_t}\big(\mathbb{1}(m) \otimes \mathbf{B}\big)\big(\boldsymbol{W_0} \otimes \mathbb{1}(N)\big)\big) \cdot \tilde{\boldsymbol{\mu}}_{\boldsymbol{0}} = \tilde{\boldsymbol{\varphi}}_{\boldsymbol{0}}$
 5: update $\boldsymbol{odesol}$
 6: $\boldsymbol{convpast} = 0$
 7: $\boldsymbol{convpartial} = 0$
 8: for $i = 1 : n$ do
 9:    for $j = p : 2$ do
10:       if $j \leq i$ then
11:          $\boldsymbol{convpast(j)}$=$\boldsymbol{convpast(j-1)}$
12:       end if
13:    end for
14:    $\boldsymbol{convpast(1)}$=$\big(\boldsymbol{W_0} \otimes \mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{i-1}} + \boldsymbol{convpartial}$
15:    $\boldsymbol{convpartial} = \sum_{j=0}^{i-1} \big(\boldsymbol{W_{i-j}} \otimes \mathbb{1}(N)\big)\tilde{\boldsymbol{\mu}}_{\boldsymbol{j}}$
16:    solve
       $$\Big(\mathbb{1}(m) \otimes \mathbf{A} - \frac{\alpha_0}{h_t}\big(\mathbb{1}(m) \otimes \mathbf{B}\big)\big(\boldsymbol{W_0} \otimes \mathbb{1}(N)\big)\Big) \cdot \tilde{\boldsymbol{\mu}}_{\boldsymbol{i}} = \big(\mathbb{1}(m) \otimes \mathbf{B}\big)\Big(\frac{\alpha_0}{h_t}\boldsymbol{convpartial} + \sum_{l=1}^{p} \frac{\alpha_l}{h_t}\boldsymbol{convpast(l)}\Big) + \tilde{\boldsymbol{\varphi}}_{\boldsymbol{i}}$$
17:    update $\boldsymbol{odesol}$
18: end for
```

Proposition 3. *Let N be the number of spatial degrees of freedom, N_I the number of nodes on the impedance boundary, m the number of stages of the Runge-Kutta method and n the number of time steps. Then the algorithm 4 ends after $\mathcal{O}(C_0(N, m) + C_1(N, m) \cdot n + N_I \cdot n \log n))$ computations for two constants $C_0(N, m)$ and $C_1(N, m)$.*

Proof. First we analyze the first 7 steps of the algorithm, which are an initialization of the problem. The creation of the mesh, as well as the computation of the two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ requires a constant time $C_{A,B,Mesh}(N)$ which depends only on N. In fact the mesh and these two matrices can be provided by an external code. Therefore we omit this constant in our analysis.

The matrices $\boldsymbol{W_0}, \cdots, \boldsymbol{W_{2B-1}}$ are computed as described in remark 2 on page 21. The Laplace transform has to be evaluated on the matrix $\Delta(\zeta)$ for several values of ζ. This requires the computation of the Jordan decomposition of the matrix $\Delta(\zeta)$ for each value of ζ and thus its computation time is a not so small constant C_J.

The time for the computation of the backward difference weights as well as the time for initialising ***odesol*** are negligible.

In the 4th step we have to compute the solution of an $N \cdot m$ linear system. Its computation time is $\mathcal{O}\big(C_{LS}(N,m)\big)$ for a constant $C_{LS}(N,m)$ which depends on N and m. The matrix of this linear system can be computed and stored for later computations in time $\mathcal{O}\big(C_{KP1}(N,m) + C_{KP2}(N,m) + C_{MS}(N,m) + C_{MP}(N,m)\big)$, where $C_{KP1}(N,m)$ is the time necessary for the Kronecker product between an $m \times m$ sparse identity matrix and an $N \times N$ sparse matrix, $C_{MS}(N,m)$ is the time necessary for the sum of two sparse $N \cdot m \times N \cdot m$ matrices, $C_{MP}(N,m)$ is the time necessary for the product of two sparse $N \cdot m \times N \cdot m$ matrices and $C_{KP2}(N,m)$ is the time necessary for the Kronecker product between an $m \times m$ full matrix and a $N \times N$ sparse identity matrix.

So far, the computation effort is $\mathcal{O}\big(C_0(N,m)\big)$, where $C_0(N,m) := \max\big(C_J, C_{LS}(N,m), C_{KP1}(N,m), C_{KP2}(N,m), C_{MS}(N,m), C_{MP}(N,m)\big)$.

Now we analyse the FOR-cycle. The lines 9-13 concern the computation of the variable ***convpast***. The main computational effort is the one due to the multiplication of an $N \cdot m \times N \cdot m$ sparse matrix times an $N \cdot m$ vector. Since the convolution is to be taken only on the impedance nodes, this requires only $\mathcal{O}(N_I \cdot m^2)$ computations.

In line 15 we compute the variable ***convpartial***, which is nothing else than a truncated convolution on the impedance nodes. This requires at most $\mathcal{O}(N_I \cdot (\log(i) + 2B \cdot m^2))$ multiplications at the i-th step (the term $N_I \cdot 2B \cdot m^2$ is due to the computation of $\boldsymbol{U_n^{(1)}}$).

In the 16th line we have to solve again a linear system. Regarding the right hand side, the sum in the brackets requires $\mathcal{O}(N_I \cdot p)$ computations, the product with the matrix can be performed with $\mathcal{O}(N_I \cdot m^2)$ computations and the sum of the resulting vector with the load vector can be perfomed in $\mathcal{O}(N_I)$ additions. Thus the linear system can be solved in $\mathcal{O}(C_{LS}(N,m) + N_I \cdot p + N_I \cdot m^2 + N_I)$.

It follows that for the whole FOR-cycle the computational effort is $\mathcal{O}(n \cdot (C_{LS}(N,m) + N_I \cdot (p + 2B \cdot m^2 + \log(n))))$. Thus setting $C_1(N,m) := C_{LS}(N,m) + N_I \cdot (p + 2B \cdot m^2)$ shows the claimed computational effort. $\square$

5.3 Numerical example of a parabolic PDE with Dirichlet and impedance boundary conditions

In order to test the algorithm we have to construct a PDE which should be of the form

$$\begin{cases} -\Delta u &= 0 & \text{in } \Omega\times]0;\mathrm{T}], \\ u &= g & \text{on } \partial\Omega\times]0;\mathrm{T}], \\ \boldsymbol{\nabla} u \cdot \boldsymbol{n} &= \frac{d}{dt}\int_0^t k(t-\tau)\cdot u(\tau)d\tau & \text{on } \partial\Omega_0\times]0;\mathrm{T}], \end{cases}$$

where $\mathcal{L}(k) = -\frac{1}{\sqrt{s}}$ and for a $g \in H^{1/2}(\partial\Omega) \cap C\big(]0;\mathrm{T}]\big)$ (in order to find a reference solution, it is simpler to consider a general function g than impose $g = 0$). We choose $\Omega \setminus \Omega_0$ to be an annulus around zero with radii 0.5 and 2 (see figure 9).

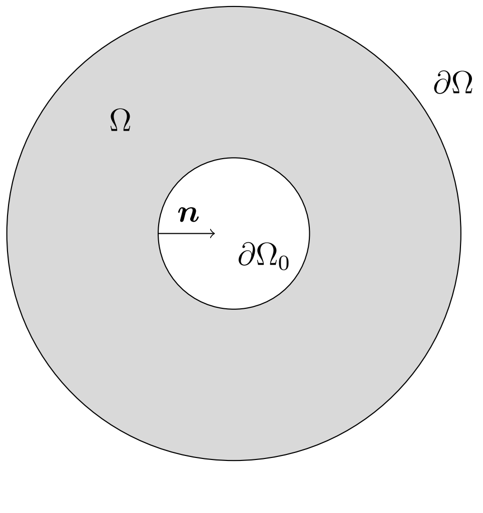


Figure 9: Domain of the PDE.

In the Laplace domain the PDE reads

$$\begin{cases} -\Delta\mathcal{L}(u) &= 0 & \text{in } \Omega, \\ \mathcal{L}(u) &= \mathcal{L}(g) & \text{on } \partial\Omega, \\ \boldsymbol{\nabla}\mathcal{L}(u)\cdot\boldsymbol{n} &= -\sqrt{s}\cdot\mathcal{L}(u) & \text{on } \partial\Omega_0. \end{cases}$$

Because of the domain geometry we move to polar coordinates, in which case the Laplace operator reads

$$\Delta = \frac{1}{r}\frac{\partial}{\partial r}\big(r\frac{\partial}{\partial r}\cdot\big) + \frac{1}{r^2}\frac{\partial^2}{\partial\varphi^2}\cdot.$$

Moreover we assume that

$$\mathcal{L}(g)(s) := e^{ik\varphi}\Phi(s),$$

where k is an integer and $\Phi(s)$ is a complex function (which is introduced in order to allow the inverse Laplace transform of $\mathcal{L}(u)$). With the Ansatz of separation of variables

$$\mathcal{L}(u) := u_r(r)\cdot u_\varphi(\varphi)$$

and by imposing

$$u_\varphi(\varphi) := e^{ik\varphi}\Phi(s),$$

the Laplace equation becomes

$$-\bigg(\frac{1}{r}\frac{\partial}{\partial r}\big(r\frac{\partial}{\partial r}u_r\big) - \frac{k^2}{r^2}u_r\bigg)u_\varphi = 0,$$

which is equivalent to

$$\frac{\partial^2}{\partial r^2}u_r + \frac{1}{r}\frac{\partial}{\partial r}u_r - \frac{k^2}{r^2}u_r = 0.$$

If $k \neq 0$, its solution is

$$u_r(r) = Ar^k + Br^{-k},$$

where A and B are two constants. On the other hand, if $k = 0$, its solution is

$$u_r(r) = A\log(r) + B, \tag{53}$$

where A and B are once again two constants. For the sake of simplicity, we assume that $k = 0$. The values of the two constants A and B are determined from the boundary conditions of the PDE. Those read (in the Laplace domain and in polar coordinates)

$$\begin{cases} u_r(2) &= 1, \\ -\frac{\partial}{\partial r}u|_{r=0.5} &= -\sqrt{s}u_r(0.5). \end{cases} \tag{54}$$

The minus sign on the left hand side of the second boundary condition is due to the opposite orientation of the vector $\boldsymbol{n}$ and the axes r. Inserting the Ansatz (53) in the system (54) gives

$$\left\{ \begin{array}{rcl} A\log(2) + B &=& 1, \\ 2A &=& \sqrt{s}\big(A\log(0.5) + B\big). \end{array} \right. \tag{55}$$

From the first equation of (55) we read

$$B = 1 - A\log(2).$$

Imposing it to the second equation of (55) gives

$$\begin{aligned} 0 &= 2A - \sqrt{s}\big(A\log(0.5) + B\big) \\ &= 2A - \sqrt{s}\big(A\log(0.5) + 1 - A\log(2)\big) \\ &= 2A - \sqrt{s}\big(1 - A\log(4)\big) \\ &= A\big(2 + \sqrt{s}\log(4)\big) - \sqrt{s} \end{aligned}$$

and thus

$$A = \frac{\sqrt{s}}{2 + \sqrt{s}\log(4)}, \qquad B = 1 - \frac{\sqrt{s}}{2 + \sqrt{s}\log(4)}\log(2).$$

Before moving back to time domain we still have to choose the function $\Phi(s)$. This function should have a relatively simple inverse Laplace transform and must decay strong enough in order to make $\mathcal{L}(u)$ Laplace invertible. We choose

$$\Phi(s) := \frac{1}{As^4}$$

because

$$\mathcal{L}^{-1}\left(\frac{1}{s^4}\right) = \frac{t^3}{6}$$

and

$$\mathcal{L}^{-1}\big(\Phi(s)\big) = \frac{32}{105\sqrt{\pi}}t^{7/2} + \log(4)\frac{t^3}{6}.$$

Moving back to the time domain it holds that

$$\begin{aligned} u &= \mathcal{L}^{-1}\big(u_r u_\varphi\big) \\ &= \mathcal{L}^{-1}\big((A\log(r)+B)\Phi(s)\big) \\ &= \mathcal{L}^{-1}\big((A\log(r)-A\log(2)+1)\Phi(s)\big) \\ &= \big(\log(r)-\log(2)\big)\mathcal{L}^{-1}\left(\frac{1}{s^4}\right)+\mathcal{L}^{-1}\big(\Phi(s)\big) \\ &= \big(\log(r)-\log(2)\big)\frac{t^3}{6}+\frac{32}{105\sqrt{\pi}}t^{7/2}+\log(4)\frac{t^3}{6} \\ &= \frac{32}{105\sqrt{\pi}}t^{7/2}+\frac{t^3}{6}\big(\log(r)+\log(2)\big), \end{aligned}$$

which is the solution of

$$\left\{\begin{array}{rcll} -\Delta u &=& 0 & \text{in } \Omega\times]0;\mathrm{T}], \\ u &=& \frac{32}{105\sqrt{\pi}}t^{7/2}+\frac{t^3}{6}\log(4) & \text{on } \partial\Omega\times]0;\mathrm{T}], \\ \boldsymbol{\nabla} u\cdot\boldsymbol{n} &=& \frac{d}{dt}\int_0^t k(t-\tau)\cdot u(\tau)d\tau & \text{on } \partial\Omega_0\times]0;\mathrm{T}]. \end{array}\right. \tag{56}$$

Now we can test how well the coupling of FEM and FCQ works for this PDE. We perform a first test using the hat functions for the spatial approximation of FEM and we approximate the convolution by using the FCQ based on the implicit Euler method (the derivative of the convolution is approximated with a first order backward differentiation).

We consider six different spatial grids and twelve different time steps and we compute the error at the end time $T = 4$ with respect to the analytic solution in the L^2-norm. This error is the sum of two main sources: the FEM-error and the convolution error. In the L^2 norm the FEM-error should have an algebraic decay of order 2. Regarding the convolution error, we should have a uniform algebraic convergence of order 1 (for FCQ based on implicit Euler) on each point of the inner boundary $\partial\Omega_0$. Since we are using linear functions for the spatial approximation, we expect that the convolution error has an algebraic contribution of the same rate of convergence to the coupling error in the L^2-norm.

These remarks are confirmed by the experiment (see figure 10). Considering the smallest time step h_t and approximating the rate of algebraic convergence in the L^2-norm versus the spacestep h gives the value 1.9470. On the other side, considering the finest meshgrid and approximating the rate of algebraic convergence in the L^2-norm versus the first nine time-steps h_t (when the error is dominated by the convolution quadrature error, as we can see in the image 10) gives a rate of convergence equal 0.9672.

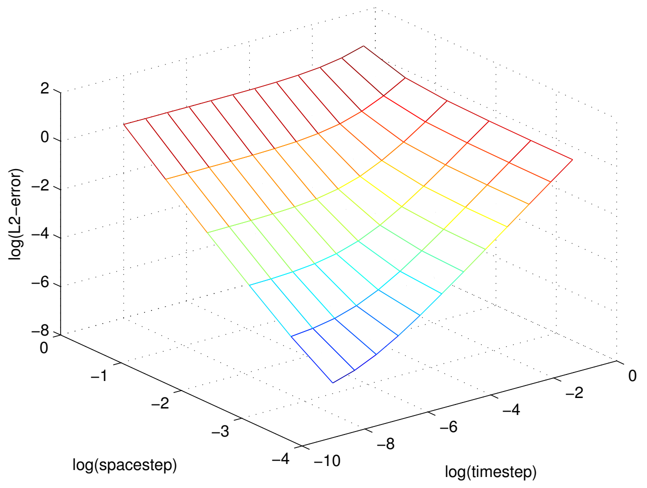


Figure 10: Error in the L^2-norm for the coupling of FEM and FCQ base on the implicit Euler method.

The values of the error can be read in table 11. Figure 11 contains the computation time of the coupling algorithm for a fixed spatial mesh (we have chosen the first mesh, whose step is $h = 0.8468$) after subtracting the time necessary for solving the linear system. We see that the growth of the computation time is steeper than a linear growth but definitely less so than a square one and almost coincides with the theoretical growth $\mathcal{O}(n \log(n))$ that we have found in proposition 3

$h \setminus h_t$	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0.8468	2.3530	1.8337	1.5807	1.4583	1.3986	1.3693
0.4234	1.3996	0.8473	0.5699	0.4348	0.3700	0.3388
0.2117	1.1968	0.6396	0.3542	0.2109	0.1404	0.1063
0.1059	1.1498	0.5927	0.3064	0.1615	0.0887	0.0526
0.0529	1.1384	0.5814	0.2952	0.1501	0.0771	0.0405
0.0265	1.1356	0.5786	0.2924	0.1473	0.0743	0.0376

$h \setminus h_t$	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}
0.8468	1.3547	1.3475	1.3439	1.3421	1.3411	1.3407
0.4234	0.3236	0.3162	0.3125	0.3106	0.3097	0.3092
0.2117	0.0901	0.0823	0.0785	0.0766	0.0757	0.0753
0.1059	0.0349	0.0264	0.0223	0.0204	0.0194	0.0190
0.0529	0.0222	0.0131	0.0087	0.0066	0.0056	0.0051
0.0265	0.0193	0.0101	0.0055	0.0033	0.0022	0.0016

Table 11: Values of the error in the L^2-norm for the coupling of FEM and FCQ base on the implicit Euler method.

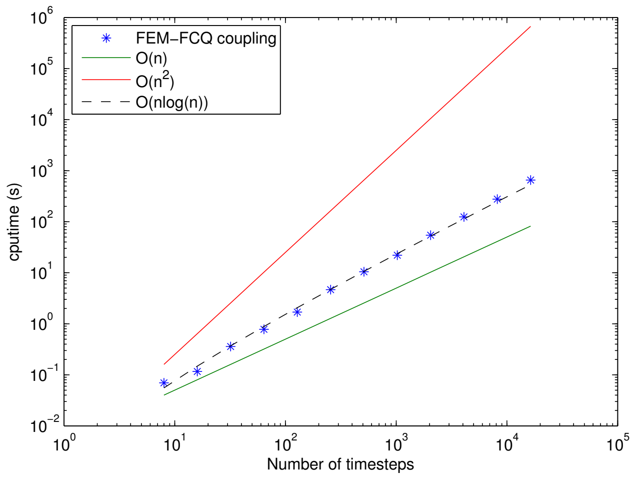


Figure 11: Cpu time in seconds (after subtracting the time necessary for solving the linear system) versus the number of time steps.

We perform a second test and this time the convolution is approximated by using the FCQ based on the 2-stage RadauIIA method (the derivative of the convolution is approximated with a third order backward differentiation). Regarding the FEM, we again choose the hat functions as elements. We consider six spatial grids and six different time steps and we compute the error at the end time $T = 4$ with respect to the analytic solution in the L^2-norm.

This time it is expected that the convolution quadrature error contributes to the coupling error with an algebraic rate equal to 3. This is only partially confirmed by the experiment because in this case the coupling error is almost always dominated by the FEM error, as we can see in figure 12. Considering the finest meshgrid and computing the rate of convergence of the convolution quadrature for the first two time steps gives the value 2.7842 which indicates that the order of convergence 3 should be correct. The values of the error in the L^2-norm can be found in the table 12.

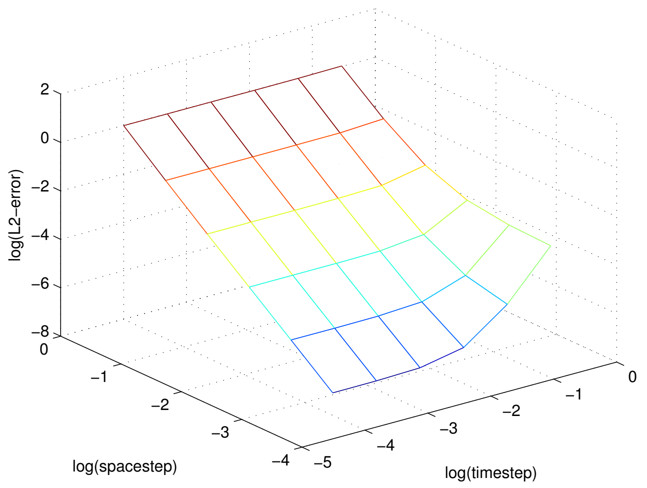


Figure 12: Error in the L^2-norm for the coupling of FEM and FCQ base on the 2-stage RadauIIA method.

$h \setminus h_t$	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0.8468	1.3736	1.3444	1.3408	1.3403	1.3403	1.3402
0.4234	0.3437	0.3131	0.3093	0.3089	0.3088	0.3088
0.2117	0.1118	0.0791	0.0753	0.0749	0.0748	0.0748
0.1059	0.0585	0.0230	0.0190	0.0186	0.0185	0.0185
0.0529	0.0465	0.0095	0.0052	0.0047	0.0046	0.0046
0.0265	0.0437	0.0063	0.0017	0.0012	0.0012	0.0012

Table 12: Values of the error in the L^2-norm for the coupling of FEM and FCQ base on the 2-stage RadauIIA method.

The domination of the coupling error by the FEM approximation is confirmed by the following experiment. The parabolic PDE system (56) is converted into an elliptic PDE system by fixing the time $T = 4$. The impedance boundary condition is converted into a Neumann one by inserting the reference solution with $r = 0.5$. Since

$$\begin{aligned}
\frac{d}{dt}\int_0^t k(t-\tau)\cdot u(\tau)d\tau &= \frac{d}{dt}\int_0^t k(t-\tau)\cdot \frac{32}{105\sqrt{\pi}}\tau^{7/2}d\tau \\
&= \frac{d}{dt}\int_0^t \frac{-1}{\sqrt{\pi}\sqrt{t-\tau}}\cdot \frac{32}{105\sqrt{\pi}}\tau^{7/2}d\tau \\
&= \frac{d}{dt}\frac{-t^4}{12} \\
&= -\frac{t^3}{3},
\end{aligned}$$

we obtain the following elliptic PDE system

$$\begin{cases}
-\Delta u &= 0 & \text{in } \Omega, \\
u &= \frac{32}{105\sqrt{\pi}}4^{7/2} + \frac{4^3}{6}\log(4) & \text{on } \partial\Omega, \\
\boldsymbol{\nabla} u \cdot \boldsymbol{n} &= -\frac{4^3}{3} & \text{on } \partial\Omega_0.
\end{cases} \tag{57}$$

Performing six space refinements and computing the error in the L^2-norm gives the values of table 13. These values have the same order of magnitude as those in the last column on the righthand of table 12, which confirms the domination by the FEM approximation.

time step	Relative error
0.8468	1.7115
0.4234	0.4006
0.2117	0.0977
0.1059	0.0242
0.0529	0.0060
0.0265	0.0015

Table 13: Relative error of FEM applied to an elliptic PDE with Dirichlet and Neumann boundary conditions.

6 Conclusion

In his work [10] C. Lubich introduced the convolution quadrature based on strong A(α)-stable linear multistep methods for computing convolutions when the Laplace transform of the convolution kernel satisfies the sectorial condition on page 11. The algorithm has excellent stability properties and inherits the order of convergence of the multistep method. Unfortunately the error constants of strong A(α)-stable linear multistep methods are huge for α close to $\pi/2$ (see chapter V.2 of [5]). This drawback has been obviated by C. Lubich and A. Ostermann with their work [12], where they have developed the convolution quadrature based on Runge-Kutta methods, as explained in chapter 3.
In their work [15] A. Schädle, M. Lopéz-Fernández and C. Lubich have developed a fast implementation of the convolution quadrature based on Runge-Kutta methods. The new algorithm implementation reduces the number of evaluations of the Laplace transform of the convolution kernel to $\mathcal{O}(\log n)$, the active memory requirement to $\mathcal{O}(\log n)$ and the number of multiplications to $\mathcal{O}(n \log n)$ (both for computing a convolution and solving an integral equation).
In the subsection 4.8 we discussed how to approach an integral equation which involves the derivative of a convolution. The experiments indicate that approximating the derivative with a backward difference method of a suitable order should preserve the order of convergence of the convolution quadrature.
Finally in chapter 5 we have developed an algorithm for solving the variational problem (10) by coupling the finite element method and the fast convolution quadrature. The experiments show that the resulting scheme inherits the spatial order of convergence of FEM and the time order of convergence of FCQ. Proposition 3 on page 56 shows that the computation time is $\mathcal{O}(C(N,m) \cdot n + N_I \cdot n \log n))$ where N indicates the number of elements, N_I is the number of elements on the boundary $\partial\Omega_0$ and n are the timesteps. This proposition is confirmed by the experiments as we can see in figure 11.

A Mathematical tools

This appendix is a brief review of some mathematical concepts which are used in the previous chapters.

A.1 The Laplace transform

Definition 8 (Laplace transform)**.** *Let $u : [0, \infty[\to \mathbb{R}$ be a piecewise continuous function which further satisfies*

- *there exist two constants c_1, $c_2 \in \mathbb{R}$ so that*

$$\lim_{t\to\infty} |u(t)| < c_1 e^{c_2 t}, \tag{58}$$

- *for every finite $T > 0$*

$$\int_0^T |u(t)| dt < \infty. \tag{59}$$

Then the Laplace transform of u is a function $\mathcal{L}(u) : \mathbb{C} \to \mathbb{C}$ defined by

$$\mathcal{L}(u)(s) := \int_0^\infty e^{-st} u(t) dt. \tag{60}$$

This integral converges absolutely and uniformely for every $s \in \mathbb{C}$ with $Re(s) > c_2$.

The next theorem recalls some properties of the Laplace transform.

Theorem 4 (Properties of the Laplace transform)**.** *Let $u, v : [0, \infty[\to \mathbb{R}$ be two piecewise continuous functions which further satisfy the conditions* (58) *and* (59)*.*

- *the Laplace transform is a linear transform: let $\lambda \in \mathbb{C}$, then*

$$\mathcal{L}(\lambda u + v) = \lambda \mathcal{L}(u) + \mathcal{L}(v),$$

- *the Laplace transform is unique up to a null function (see Lerch's theorem),*
- *consider the function $f : t \mapsto t \cdot u(t)$, then*

$$\mathcal{L}(f) = -\frac{d}{ds}\mathcal{L}(u),$$

- *consider the function* $f : t \mapsto \frac{u(t)}{t}$, *then*
$$\mathcal{L}(f) = \int_0^s \mathcal{L}(u) d\tilde{s},$$
- *consider the function* $f : t \mapsto \int_0^t u(\tilde{t}) d\tilde{t}$, *then*
$$\mathcal{L}(f) = \frac{\mathcal{L}(u)}{s},$$
- *let* $u' := \frac{d}{dt} u$ *and consider the function* $f : t \mapsto u'(t)$, *then*
$$\mathcal{L}(f) = s \cdot \mathcal{L}(u),$$
- *let* $u^{(n)} := \frac{d^n}{dt^n} u$ *and consider the function* $f : t \mapsto u^{(n)}(t)$, *then*
$$\mathcal{L}(f) = s^n \cdot \mathcal{L}(u) - s^{n-1}u(0) - s^{n-2}u'(0) - \cdots - u^{(n-1)}(0),$$
- *let* u *and* v *have the exponential order of* (58) *equal* α *and consider the convolution* $u * v : t \mapsto \int_0^t u(\tau)v(t-\tau)d\tau$, *then for every* $s \in \mathbb{C}$ *with* $Re(s) > \alpha$ *it holds*
$$\mathcal{L}(u * v) = \mathcal{L}(u)\mathcal{L}(v).$$

The inverse of the Laplace transform is given by the Bromwich integral formula (also known as Fourier-Mellin integral or Mellin's inverse formula).

Theorem 5 (Bromwich integral formula)**.** *Let* F *be a complex function which satisfies*

- *there's a* $\sigma_0 \in \mathbb{R}$ *so that the function* F *is analytic in the region* $\{s \in \mathbb{C} | Re(s) > \sigma_0\}$,
- *there's a constant* $c \in \mathbb{C}$ *so that for every real* $b > \sigma_0$
$$\lim_{t \to \infty} F(b + it) = \frac{c}{b + it} + \mathcal{O}\left(\frac{1}{|b + it|^2}\right).$$

Then the function $f : [0, \infty[\to \mathbb{R}$ *defined by*
$$f(t) := \frac{1}{2\pi i} \lim_{T \to \infty} \int_{\sigma - iT}^{\sigma + iT} e^{st} F(s) ds$$
for a real number $\sigma > \sigma_0$ *satisfies*
$$\mathcal{L}(f) = F.$$
Thusthe inverse Laplace transform can be defined as
$$\mathcal{L}^{-1}(F) := \frac{1}{2\pi i} \lim_{T \to \infty} \int_{\sigma - iT}^{\sigma + iT} e^{st} F(s) ds.$$

A.2 The generating function

Definition 9 (Generating function)**.** *Let $(u_n)_{n\in\mathbb{N}}$ be a sequence of values. The generating function of $(u_n)_{n\in\mathbb{N}}$ is the formal power series*

$$\mathcal{G}[(u_n)_{n\in\mathbb{N}}](\zeta) := \sum_{n=0}^{\infty} u_n \zeta^n.$$

In general, the generating function has to be considered just as a formal sum. In some cases, however, the generating function is equal to the Taylor series of an analytic function. For example, the generating function of the constant sequence $(u_n)_{n\in\mathbb{N}}$ defined by

$$u_n = 1 \qquad \text{for every } n \in \mathbb{N}$$

satisfies

$$\mathcal{G}[(u_n)_{n\in\mathbb{N}}](\zeta) = \frac{1}{1-\zeta} \qquad \text{for every } \zeta \in \{z \in \mathbb{C} \| z| < 1\}.$$

A.3 Matrix functional calculus

The goal of this section is to make sense of the expression

$$f(\boldsymbol{M}) \tag{61}$$

for a complex function f and a matrix $\boldsymbol{M}$. The literature contains a generalized version of (61) for general Banach algebras (see for example [14]). Nevertheless, in order to give a more practical idea, we limit ourselves to the matrix case by following chapter one of [6].

Let $f : D \to \mathbb{C}$ be an analytic function on the open set $D \subset \mathbb{C}$. For every fixed $\alpha \in D$, it holds

$$f(z) = \sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(z-\alpha)^n \qquad \text{for every } z \in D. \tag{62}$$

Let $\boldsymbol{M}$ be a square complex matrix, whose spectrum $\sigma(\boldsymbol{M})$ lies entirely in D. The idea is to evaluate $f(\boldsymbol{M})$ by using the formula (62). We recall that any square complex matrix can be expressed with its canonical Jordan form $\boldsymbol{J}$, id est: there exists an invertible matrix $\boldsymbol{Z}$ so that

$$\boldsymbol{M} = \boldsymbol{Z}\boldsymbol{J}\boldsymbol{Z}^{\boldsymbol{-1}},$$

where

$$\boldsymbol{J} = \begin{pmatrix} \boldsymbol{J_1} & & & \\ & \boldsymbol{J_2} & & \\ & & \ddots & \\ & & & \boldsymbol{J_p} \end{pmatrix}$$

and

$$\boldsymbol{J_i} = \begin{pmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{pmatrix},$$

where the λ_i's are the eigenvalues of $\boldsymbol{M}$ and $\boldsymbol{J_i} \in \mathbb{C}^{m_i \times m_i}$. If in (62) we substitute z with $\boldsymbol{M}$ and we multiply the value α (in the subtraction) with the identity matrix $\boldsymbol{I}$, we have

$$\begin{aligned} f(\boldsymbol{M}) &= \sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!} (\boldsymbol{M} - \alpha \boldsymbol{I})^n \\ &= \sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!} (\boldsymbol{Z}\boldsymbol{J}\boldsymbol{Z}^{-1} - \alpha \boldsymbol{Z}\boldsymbol{I}\boldsymbol{Z}^{-1})^n \\ &= \sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!} (\boldsymbol{Z}(\boldsymbol{J} - \alpha \boldsymbol{I})\boldsymbol{Z}^{-1})^n \\ &= \sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!} \boldsymbol{Z}(\boldsymbol{J} - \alpha \boldsymbol{I})^n \boldsymbol{Z}^{-1} \\ &= \boldsymbol{Z}\Big(\sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!} (\boldsymbol{J} - \alpha \boldsymbol{I})^n \Big) \boldsymbol{Z}^{-1}. \end{aligned}$$

For the inner bracket on the right hand side we have (by adapting the size of $\boldsymbol{I}$)

$$\begin{aligned} (\boldsymbol{J} - \alpha \boldsymbol{I})^n &= \begin{pmatrix} \boldsymbol{J_1} - \alpha \boldsymbol{I} & & & \\ & \boldsymbol{J_2} - \alpha \boldsymbol{I} & & \\ & & \ddots & \\ & & & \boldsymbol{J_p} - \alpha \boldsymbol{I} \end{pmatrix}^n \\ &= \begin{pmatrix} (\boldsymbol{J_1} - \alpha \boldsymbol{I})^n & & & \\ & (\boldsymbol{J_2} - \alpha \boldsymbol{I})^n & & \\ & & \ddots & \\ & & & (\boldsymbol{J_p} - \alpha \boldsymbol{I})^n \end{pmatrix}. \end{aligned}$$

Now we consider

$$\sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(\boldsymbol{J_i} - \alpha \boldsymbol{I})^n$$

for a general Jordan block $\boldsymbol{J_i}$. Since f is analyitc in D and $\sigma(\boldsymbol{M}) \subset D$, we can choose $\alpha = \lambda_i$. Then $\boldsymbol{J_i} - \lambda_i \boldsymbol{I}$ is a superdiagonal matrix with all the non-zero entries equal 1. Powering of this matrix brings the superdiagonal towards the top right-hand corner, as the following example shows.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ & 0 & 1 & 0 \\ & & 0 & 1 \\ & & & 0 \end{pmatrix}^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ & 0 & 0 & 1 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ & 0 & 1 & 0 \\ & & 0 & 1 \\ & & & 0 \end{pmatrix}^3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix}$$

Thus $\boldsymbol{J_i} - \lambda_i \boldsymbol{I}$ is nilpotent with index m_i, which implies

$$\sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(\boldsymbol{J_i} - \alpha \boldsymbol{I})^n = \sum_{i=0}^{m_i-1} \frac{f^{(n)}(\alpha)}{n!}(\boldsymbol{J_i} - \alpha \boldsymbol{I})^n.$$

By considering the movement of the superdiagonal we conclude that

$$\sum_{i=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(\boldsymbol{J_i} - \alpha \boldsymbol{I})^n = \begin{pmatrix} f(\lambda_i) & f'(\lambda_i) & \cdots & \frac{f^{(m_i-1)}(\lambda_i)}{(m_i-1)!} \\ & f(\lambda_i) & \ddots & \vdots \\ & & \ddots & f'(\lambda_i) \\ & & & f(\lambda_i) \end{pmatrix}.$$

This motivates the following definitions (which is definition 1.2 on page 3 of [6]).

Definition 10 (Matrix function via Jordan canonical form)**.** *Let $f : D \to \mathbb{C}$ be an analytic function on the open set $D \subset \mathbb{C}$ and $\boldsymbol{M}$ be a square complex matrix with $\sigma(\boldsymbol{M}) \subset D$. Let $\boldsymbol{J}$ be its canonical Jordan form and $\boldsymbol{Z}$ be an invertible matrix so that*

$$\boldsymbol{M} = \boldsymbol{Z}\boldsymbol{J}\boldsymbol{Z}^{\boldsymbol{-1}}.$$

We define

$$f(\boldsymbol{M}) := \boldsymbol{Z}\, diag(f(\boldsymbol{J_i}))\boldsymbol{Z}^{-1},$$

where

$$f(\boldsymbol{J_i}) := \begin{pmatrix} f(\lambda_i) & f'(\lambda_i) & \cdots & \frac{f^{(m_i-1)}(\lambda_i)}{(m_i-1)!} \\ & f(\lambda_i) & \ddots & \vdots \\ & & \ddots & f'(\lambda_i) \\ & & & f(\lambda_i) \end{pmatrix}.$$

Remark 9. *Let's assume that $\boldsymbol{M}$ is also diagonalizable, id est: there exists a regular matrix $\boldsymbol{T}$ and a diagonal matrix $\boldsymbol{D}$ so that*

$$\boldsymbol{M} = \boldsymbol{T}\boldsymbol{D}\boldsymbol{T^{-1}},$$

where

$$\boldsymbol{D} = \begin{pmatrix} d_{11} & & & \\ & d_{22} & & \\ & & \ddots & \\ & & & d_{nn} \end{pmatrix}.$$

Then computing $f(\boldsymbol{M})$ with the definition 10 is simple because no derivatives of f have to be computed. In fact we have

$$\begin{aligned} f(\boldsymbol{M}) &= \boldsymbol{T} f(\boldsymbol{D}) \boldsymbol{T}^{-1} \\ &= \boldsymbol{T} \begin{pmatrix} f(d_{11}) & & & \\ & f(d_{22}) & & \\ & & \ddots & \\ & & & f(d_{nn}) \end{pmatrix} \boldsymbol{T}^{-1}. \end{aligned}$$

With Cauchy's integral formula we can construct an equivalent definition, which in fact can be generalized to operators (see for example definition 10.26 on page 243 of [14]). Bevore giving its formal definition, we perform some motivating computations. We start by recalling the Cauchy's integral formula for analytic functions.

Theorem 6 (Cauchy's integral formula)**.** *Let $f : D \to \mathbb{C}$ be an analytic function on the open set $D \subset \mathbb{C}$, let $z_0 \in D$ and let γ be a closed contour in D with counterclockwise direction and enclosing z_0. Then*

$$f(z_0) = \frac{1}{2\pi i} \oint_\gamma \frac{f(z)}{z - z_0} dz.$$

Let's assume that $\boldsymbol{M}$ is diagonalizable, then for a complex number $z \notin \sigma(\boldsymbol{M})$ we have

$$\begin{aligned} (z\boldsymbol{I} - \boldsymbol{M})^{-1} &= (\boldsymbol{T} z \boldsymbol{I} \boldsymbol{T}^{-1} - \boldsymbol{T}\boldsymbol{D}\boldsymbol{T}^{-1})^{-1} \\ &= \left(\boldsymbol{T}(z\boldsymbol{I} - \boldsymbol{D})\boldsymbol{T}^{-1}\right)^{-1} \\ &= \boldsymbol{T}(z\boldsymbol{I} - \boldsymbol{D})^{-1}\boldsymbol{T}^{-1} \end{aligned}$$

Since $\boldsymbol{D}$ is a diagonal matrix, we have

$$(z\boldsymbol{I}-\boldsymbol{D})^{-1} = \begin{pmatrix} \frac{1}{z-d_{11}} & & & \\ & \frac{1}{z-d_{22}} & & \\ & & \ddots & \\ & & & \frac{1}{z-d_{nn}} \end{pmatrix}.$$

Choosing γ so that it encloses $\sigma(\boldsymbol{M})$ and considering a general entry d_{ii} on the main diagonal of the matrix $\boldsymbol{D}$ gives

$$f(d_{ii}) = \frac{1}{2\pi i}\oint_\gamma \frac{f(z)}{z-d_{ii}}dz.$$

On the other side, with the definition 10, we have

$$f(\boldsymbol{D}) = \begin{pmatrix} f(d_{11}) & & & \\ & f(d_{22}) & & \\ & & \ddots & \\ & & & f(d_{nn}) \end{pmatrix}.$$

Thus, by considering the integral componentwise we have

$$f(\boldsymbol{D}) = \frac{1}{2\pi i}\oint_\gamma f(z)(z\boldsymbol{I}-\boldsymbol{D})^{-1}dz.$$

Moreover, the linearity of the integral allows writing

$$\boldsymbol{T}\left(\frac{1}{2\pi i}\oint_\gamma f(z)(z\boldsymbol{I}-\boldsymbol{D})^{-1}dz\right)\boldsymbol{T}^{-1} = \frac{1}{2\pi i}\oint_\gamma f(z)\boldsymbol{T}(z\boldsymbol{I}-\boldsymbol{D})^{-1}\boldsymbol{T}^{-1}dz.$$

Therefore, by collecting all the results and starting with definition 10 (with $\boldsymbol{T}=\boldsymbol{Z}$), we have

$$\begin{aligned} f(\boldsymbol{M}) &= \boldsymbol{T}f(\boldsymbol{D})\boldsymbol{T}^{-1} \\ &= \boldsymbol{T}\left(\frac{1}{2\pi i}\oint_\gamma f(z)(z\boldsymbol{I}-\boldsymbol{D})^{-1}dz\right)\boldsymbol{T}^{-1} \\ &= \frac{1}{2\pi i}\oint_\gamma f(z)\boldsymbol{T}(z\boldsymbol{I}-\boldsymbol{D})^{-1}\boldsymbol{T}^{-1}dz \\ &= \frac{1}{2\pi i}\oint_\gamma f(z)(z\boldsymbol{I}-\boldsymbol{M})^{-1}dz. \end{aligned}$$

This motivates the following definition (which is definition 1.11 on page 8 of [6]).

Definition 11 (Matrix function via Cauchy's integral formula)**.** *Let $f : D \to \mathbb{C}$ be an analytic function on the open set $D \subset \mathbb{C}$ and $\boldsymbol{M}$ be a square complex matrix with $\sigma(\boldsymbol{M}) \subset D$. We define*

$$f(\boldsymbol{M}) := \frac{1}{2\pi i} \oint_{\gamma} f(z)(z\boldsymbol{I} - \boldsymbol{M})^{-1} dz,$$

where γ is a closed contour contained in D, with counterclockwise direction and enclosing $\sigma(\boldsymbol{M})$.

The theorem 1.12 on page 8 of [6] states that definition 11 is equivalent to the definition 10. Its proofs refers to the theorem 6.2.28 of [7].

B Codes

B.1 Convolution

```
%compute int_0^t f(t-tau)g(tau) dtau, where the Laplace transform of the
%kernel is F(s)=1/sqrt(s). The function g(t) can be choosen between
%four possibilities. The weights Matrices are computed with the
%Cauchy integral on a circle and can be computed for the first three
%RadauIIA methods. The code plots the absolute error at the
%endtime Tend versus the timestep h in double logarithmic scale and
%approximate the algebraic order of convergence.

function simple_classic_convolution

%final time
Tend=4;

%Laplace transform of f=1/sqrt(pi*t)
F=@(t) 1./sqrt(t);

%function g(t)
%g=@(t) exp(t);
%g = @(t) sqrt(pi)*t;
g = @(t) sqrt(pi)*t.^2;
%g=@(t) sqrt(pi*t);

%analytic solution
%sol=@(t) exp(t).*erf(sqrt(t));
%sol=@(t) 4/3*t.^(3/2);
sol=@(t) 16/15*t.^(5/2);
%sol=@(t) pi*t/2;

%choose the Runge-Kutta method
%[c,D]=impliciteuler;
[c,D]=Radau2stages;
%[c,D]=Radau3stages;

%plot the error
for i=1:4
    h(i)=0.5^(i-1);
    err(i)=error(Tend,h(i),g,sol,F,c,D);
end
figure;
loglog(h,err,'*-');
xlabel('logarithm of timestep h');
ylabel('logarithm of absolute error');
legend('absolute error at final time');

%compute the algebraic order of convergence
p=polyfit(log(h),log(err),1);
fprintf('algebraic order of convergence %f\n',p(1));
end



function err=error(Tend,h,g,sol,F,c,D)

%number of iterations
Nite=ceil(Tend/h);
N=Nite-1;
```

```
%define a function G so that G(t)=g(t+c_1,...,t+c_m) where m is the number
%of stages of the Runge-Kutta method and (c_1,...,c_m) is the time vector
%of the Runge-Kutta Method
G=@(j) g(j*h+c*h);

%compute the approximated solution
u=W_0_cerchio(F,D,h)*G(N);
for i=1:N
    u=u+W_cerchio(F,D,h,i)*G(N-i);
end

%compute the absolute error at final time
err=abs(sol(Tend)-real(u(end)));

end


function [c,D]=impliciteuler
c=1;
D=@(z) 1-z;
end


function [c,D]=Radau2stages
c=[1/3; 1];
D=@(z) 0.5*[3 1-4*z;-9 4*z+5];
end


function [c,D]=Radau3stages
c=[(4-sqrt(6))/10; (4+sqrt(6))/10; 1];
RKA=[(88-7*sqrt(6))/360 (296-169*sqrt(6))/1800 (-2+3*sqrt(6))/225;
    (296+169*sqrt(6))/1800 (88+7*sqrt(6))/360 (-2-3*sqrt(6))/225;
    (16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
b=[(16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
invRKA=inv(RKA);
D=@(z) invRKA -z*invRKA*[1;1;1]*b*invRKA;
end


%compute the first weight W_0 over the circle. F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep
function W_0_cerchio_test=W_0_cerchio(F,D,h)
W_0_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=500;
for l=0:L-1
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_0_cerchio_test=W_0_cerchio_test+1/L*V*diag(F(diag(Diag)))*inv(V);
end
end


%compute the n-th weight W_n over the circle .F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep, n is the index of the n-th weight.
function W_cerchio_test=W_cerchio(F,D,h,n)
```

```
W_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=500;
for l=1:L
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_cerchio_test=W_cerchio_test+V*diag(F(diag(Diag)))*inv(V)*...
        exp(-2i*pi*n*l/L);
end
W_cerchio_test=rho^(-n)/L*W_cerchio_test;
end
```

B.2 Approximation of the contour integral

```
%Test the hyperbola approximation of the convolution weights when the
%Laplace transform of the convolution kernel is F(s)=1/sqrt(s). The code
%plots the error of the hyperbola approximation of the convolution weights
%versus the weight index with respect to reference weights compute on the
%hyperbola with a large number of quadrature points.


function testweights_hyperbola

%timestep
h=0.25;

%partitioning constant
B=10;

%quadrature point number (the quadrature points are 2*K+1)
K=10;

%Laplace transform di f=1/sqrt(pi*t)
F=@(t) 1./sqrt(t);

%choose the Runge-Kutta method
[e]=impliciteuler;
%[e]=Radau2stages;
%[e]=Radau3stages;

rho = rho_optimal(B,K);

%first contour
l=1;
err=zeros(3,1999);
for i=B^0:2*B^1-1
    %compute reference solution
    a=real(W_hyperbola(i,h,50,F,e,l,rho,B));
    %compute approximation of the hyperbola
    b=real(W_hyperbola(i,h,10,F,e,l,rho,B));
    err(1,i)=abs(a(end)-b(end));
end

%second contour
l=2;
for i=B^1:2*B^2-1
    %compute reference solution
    a=real(W_hyperbola(i,h,50,F,e,l,rho,B));
    %compute approximation of the hyperbola
    b=real(W_hyperbola(i,h,10,F,e,l,rho,B));
    err(2,i)=abs(a(end)-b(end));
end

%third contour
l=3;
for i=B^2:2*B^3-1
    %compute reference solution
    a=real(W_hyperbola(i,h,50,F,e,l,rho,B));
    %compute approximation of the hyperbola
    b=real(W_hyperbola(i,h,10,F,e,l,rho,B));
    err(3,i)=abs(a(end)-b(end));
end

%plot the errors
figure;
loglog(1:19,err(1,1:19),10:199,err(2,10:199),100:1999,err(3,100:1999));
axis([1 2500 10^(-12) 1]);
xlabel('index of weight');
ylabel('absolute error');
legend('1st contour','2nd contour','3rd contour');
end


function [e]=impliciteuler
b=1;
RKA=1;
e=@(n,z) (1+z*b*((eye(1)-z*RKA)\ones(1,1)))^n*b*inv(eye(1)-z*RKA);
end


function [e]=Radau2stages
b=[3/4 1/4];
RKA=[5/12 -1/12;3/4 1/4];
e=@(n,z) (1+z*b*((eye(2)-z*RKA)\ones(2,1)))^n*b*inv(eye(2)-z*RKA);
end


function [e]=Radau3stages
RKA=[(88-7*sqrt(6))/360 (296-169*sqrt(6))/1800 (-2+3*sqrt(6))/225;
    (296+169*sqrt(6))/1800 (88+7*sqrt(6))/360 (-2-3*sqrt(6))/225;
    (16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
b=[(16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
invRKA=inv(RKA);
e=@(n,z) (1+z*b*((eye(3)-z*RKA)\ones(3,1)))^n*b*inv(eye(3)-z*RKA);
end

%compute the optimal rho with the strategy described in section 4.2. B is
%the partitioning constant and K is the quadrature point number
function rho =rho_optimal(B,K)
alpha=1;
d=alpha;
sigma=0;
a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
eps_K=@(rho) exp(-2*pi*d/a(rho)*K);
valorevecchio = 100;
for i=1:9999
    rho=i/10000;
    valoremuovo=eps*eps_K(rho)^(rho-1)+eps_K(rho)^rho;
    if valoremuovo < valorevecchio
```

```
        j=i;
        valorevecchio=valoremuovo;
    end
end
rho=j/10000;
end

%compute the convolution weight by approximating it on the hyperbola. n is
%the index of the convolution weight, h is the timestep, K is the number
%of quadrature points on the hyperbola, F is the Laplace transform of the
%convolution kernel, e is related to the Runge-Kutta method, l indicates
%which contour is taken into account, rho is a parameter of the
%hyperbola and B is the partitioning constant
function W_hyperbola_test=W_hyperbola(n,h,K,F,e,l,rho,B)
alpha=1;
d=alpha;
sigma=0;
a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
tau=1/K*a(rho);
mu=2*pi*d*K*(1-rho)/(2*B^l-2)/h/a(rho);

%hyperbola function
hyperbola=@(x) mu*(1-sin(alpha+x*1i));

%hyperbola derivative function
dhyperbola=@(x) -1i*mu*cos(alpha+x*1i);

%quadrature weights
w=@(k) 1i*tau/2/pi*dhyperbola(k*tau);

%quadrature points
lambda=@(k) hyperbola(k*tau);

%compute the weight
W_hyperbola_test=0;
for k=-K:K
    a=w(k);
    b=e(n,h*lambda(k));
    c=F(lambda(k));
    W_hyperbola_test=W_hyperbola_test+h*a*b*c;
end

end
```

B.3 Integral equation

```
% Solve the integral equation y(t) = H(t) - int_0^t f(t-tau)y(tau)dtau,
% y(0) = 0,  where f(t)=1/sqrt(pi*t), F(s)=1/sqrt(s),
% H(t)=4/3*t^(3/2)+t*sqrt(pi). The analytic solution is y(t)=t*sqrt(pi).
% The Runge-Kutta method is choosen between the first three RadauIIA
% methods. The code plots the absolute error at final time versus the
% timestep h and approximates the algebraic order of convergence.


function integral_equation_fast

%final time
Tend=4;
```

```
%partitioning number
B=10;

%quadrature point number (the quadrature points are 2*K+1)
K=15;

%number of time refinements
NoRef=5;

%function H(t)
H=@(t) 4/3*t.^(3/2)+t*sqrt(pi);

%Laplace transform of the kernel f=1/sqrt(pi*t)
F=@(t) 1./sqrt(t);

%analytic solution of the integral equation
sol=@(t) t*sqrt(pi);


%Runge-Kutta method
%[r,e,c,RKA,D]=impliciteuler;
[r,e,c,RKA,D]=Radau2stages;
%[r,e,c,RKA,D]=Radau3stages;

%preallocation
h=zeros(1,NoRef);
err=zeros(1,NoRef);
time=zeros(1,NoRef);

%compute the error
for i=1:NoRef
    h(i)=0.5^(i);
    tic;
    err(i)=error(Tend,h(i),sol,F,r,e,c,D,RKA,H,B,K);
    time(i)=toc;
end

%plot the absolute error at final time versus the timestep h
figure;
loglog(h,err,'*-');
xlabel('logarithm of timestep h');
ylabel('logarithm of absolute error');
legend('absolute error at final time');

%compute the algebraic order of convergence
p=polyfit(log(h),log(err),1);
fprintf('algebraic order of convergence after %u refinements: %f\n',...
    NoRef,p(1));

end

%compute the absolute error at final time Tend. h is the timestep, sol is
%the analytic solution, F is the Laplace transform of the kernel,
%r,e,c,D,RKA are related to the Runge-Kutta method, H is the function H(t),
%B is the partitioning number and K is the quadrature point number.
function err=error(Tend,h,sol,F,r,e,c,D,RKA,H,B,K)

%number of iterations
Nite=ceil(Tend/h);


%the first 2*B weights are computed with the Cauchy integral on the circle
```

```
W_0=W_0_cerchio(F,D,h);
W=cell(2*B,1);
for i=1:2*B
    W{i}=W_cerchio(F,D,h,i);
end

%inizialize contour
[w,lambda,Lmax]=contour(B,Nite,K,h);

%precompute the partitions
[b,L]=tinte(Nite,B);

%approximated solution vector: y(i) = y((i-1+c)*h)
y=cell(1,Nite);
y{1}=real((eye(size(W_0))+W_0)\H(c*h));

%struct odesol
%odesol{a,b} contains 2*K+1 solutions of the contour l=a at time t=b*h
%necessary only until time (Nite-1)*h
odesol=cell(Lmax,Nite-1);
odesol{Lmax,Nite-1}=[];

%compute the first values of odesol and store some values which are used
%many times
rr=zeros(Lmax,2*K+1);
e0=cell(Lmax,2*K+1);
a2=cell(Lmax,2*K+1);
for l=1:Lmax
    for j=-K:K
        rr(l,j+K+1)=r(h*lambda(l,j+K+1));
        e0{l,j+K+1}=e(0,h*lambda(l,j+K+1));
        odesol{l,1}{1}(j+K+1)=h*e0{l,j+K+1}*y{1};
        a2{l,j+K+1}=(eye(size(RKA))-h*lambda(l,j+K+1)*RKA)\ones(size(c))...
            *F(lambda(l,j+K+1));
    end
end

%vector used for updating odesol
k1=zeros(Lmax,1);
k2=ones(Lmax,1);


for i=1:Nite-1
    ypast=0;

    %weights approximated on the hyperbola only from the second contour
    if L(i)>1

    for l=2:L(i)
        for j=-K:K
            %those terms are computed separately otherwise Matlab makes an
            %error (smallnumber*smallnumber=hugenumber)
            a1=w(l,j+K+1)*rr(l,j+K+1)^(i-b(l,i));
            a3=odesol{l,b(l,i)}{1}(1,j+K+1);
            ypast=ypast+a1*a2{l,j+K+1}*a3;
        end
    end


    %classic (non-fast) convolution for the first contour (weights
    %approximated with Cauchy integral on the circle)
    for m=b(2,i):i-1
```

```
        ypast=ypast+W{i-m}*y{m+1};
    end
    else
    for m=0:i-1
        ypast=ypast+W{i-m}*y{m+1};
    end
    end

    %compute the new value of the solution
    y{i+1}= real((eye(size(W_0))+W_0)\(H((i+c)*h)-ypast));

    %update odesol
    if i+1<Nite
        for l=1:Lmax
            if i+1==(2+k1(l))*B^l-1
                %cancel old values and compute the new ones
                if l>1
                    %store a value in the past
                    odesol{l,b(l,i+1)}{1}=odesol{l,b(l,i+1)}{2};
                end
                for j=-K:K
                    odesol{l,i+1}{1}(j+K+1)=rr(l,j+K+1).*odesol{l,i}{2}(j+K
                          +1)+h*e0{l,j+K+1}*y{i+1};
                end
                k1(l)=k1(l)+1;
            else
                %compute new values
                for j=-K:K
                    odesol{l,i+1}{1}(j+K+1)=rr(l,j+K+1).*odesol{l,i}{1}(j+K
                          +1)+h*e0{l,j+K+1}*y{i+1};
                end
                if size(odesol{l,i},2)==2
                    for j=-K:K
                        odesol{l,i+1}{2}(j+K+1)=rr(l,j+K+1).*odesol{l,i}{2}(
                              j+K+1)+h*e0{l,j+K+1}*y{i+1};
                    end
                end
            end
            if i+1==k2(l)*B^l+1
                %initialize values for the future
                for j=-K:K
                    odesol{l,i+1}{2}(j+K+1)=h*e0{l,j+K+1}*y{i+1};
                end
                k2(l)=k2(l)+1;
            end
        end
    end

end

%compute the absolute error
err=abs(sol(Tend)-y{end}(end));

end


function [r,e,c,RKA,D]=impliciteuler
c=1;
RKA=1;
r=@(z) 1/(1-z);
e=@(n,z) (1-z)^(-n-1);
D=@(z) 1-z;
```

```
end


function [r,e,c,RKA,D]=Radau2stages
b=[3/4 1/4];
c=[1/3; 1];
RKA=[5/12 -1/12;3/4 1/4];
r=@(z) 1+z*b*((eye(2)-z*RKA)\ones(2,1));
e=@(n,z) (1+z*b*((eye(2)-z*RKA)\ones(2,1)))^n*b*inv(eye(2)-z*RKA);
D=@(z) 0.5*[3 1-4*z;-9 4*z+5];
end


function [r,e,c,RKA,D]=Radau3stages
b=[(16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
c=[(4-sqrt(6))/10; (4+sqrt(6))/10; 1];
RKA=[(88-7*sqrt(6))/360 (296-169*sqrt(6))/1800 (-2+3*sqrt(6))/225;
    (296+169*sqrt(6))/1800 (88+7*sqrt(6))/360 (-2-3*sqrt(6))/225;
    (16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
r=@(z) 1+z*b*((eye(3)-z*RKA)\ones(3,1));
e=@(n,z) (1+z*b*((eye(3)-z*RKA)\ones(3,1)))^n*b*inv(eye(3)-z*RKA);
invRKA=inv(RKA);
D=@(z) invRKA -z*(RKA\[1;1;1])*b*invRKA;
end

%compute the first weight W_0 over the circle. F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep
function W_0_cerchio_test=W_0_cerchio(F,D,h)
W_0_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=100;
for l=0:L-1
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_0_cerchio_test=W_0_cerchio_test+1/L*V*diag(F(diag(Diag)))*inv(V);
end
end

%compute the n-th weight W_n over the circle .F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep, n is the index of the n-th weight.
function W_cerchio_test=W_cerchio(F,D,h,n)
W_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=100;
for l=1:L
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_cerchio_test=W_cerchio_test+V*diag(F(diag(Diag)))*inv(V)*exp(-2i*pi*n*
          l/L);
end
W_cerchio_test=rho^(-n)/L*W_cerchio_test;
end

%compute all the hyperbola quadrature weights, the hyperbola quadrature
%points and the number of contours involved. B is the partitioning number,
%Nite is the number of iterations, K is the quadrature point number, h is
```

```
%the timestep
function [w,lambda,Lmax]=contour(B,Nite,K,h)

%compute the biggest number of contour involved
if ceil(log((Nite)/2)/log(B))==log((Nite)/2)/log(B);
    Lmax =log((Nite)/2)/log(B)+1;
else
    Lmax=ceil(log((Nite)/2)/log(B));
end

%set the hyperbola parameters
alpha=1;
d=alpha;
sigma=0;
rho=rho_optimal(B,K,alpha,d);

a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
tau=1/K*a(rho);


w=zeros(Lmax,2*K+1);
lambda=zeros(Lmax,2*K+1);
for l=1:Lmax
    mu=2*pi*d*K*(1-rho)/(2*B^l-2)/h/a(rho);

    %hyperbola function
    hyperbola=@(x) mu*(1-sin(alpha+x*1i))+sigma;
    %hyperbola derivative function
    dhyperbola=@(x) -1i*mu*cos(alpha+x*1i);

    %compute the hyperbola quadrature weights and the hyperbola quadrature
    %point
    for k=-K:K
        w(l,k+K+1)=1i*tau/2/pi*dhyperbola(k*tau);
        lambda(l,k+K+1)=hyperbola(k*tau);
    end
end
end

%compute the optimal value of rho with the strategy described in section
%4.2. B is the partitioning number, K is the quadrature number, alpha and d
%are parameters of the hyperbola
function rho =rho_optimal(B,K,alpha,d)
a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
eps_K=@(rho) exp(-2*pi*d/a(rho)*K);
valorevecchio = 100;
for i=1:99
    rho=i/100;
    valoremuovo=eps*eps_K(rho)^(rho-1)+eps_K(rho)^rho;
    if valoremuovo < valorevecchio
        j=i;
        valorevecchio=valoremuovo;
    end
end
rho=j/100;
end

% compute all the partition times. Nite is the number of iterations. For
% N=2:Nite store in b(:,N-1) the times (N-1)=b(1,N-1)-b(2,N-1), b1-b2,...
% b(L(N-1)-1,N-1))-0=b(L(N),N-1)
function [b,L]=tinte(Nite,B)
if ceil(log((Nite)/2)/log(B))==log((Nite)/2)/log(B);
```

```
    Lmax =log((Nite)/2)/log(B)+1;
else
    Lmax=ceil(log((Nite)/2)/log(B));
end
q=zeros(Lmax+1,Nite-1);
b=zeros(Lmax+1,Nite-1);
L=ones(1,Nite-1);
for N=2:Nite
    N=N-1;
    b(1,N)=N;
    for n =1:N
        if 2*B^L(N) == n+1
            L(N)=L(N)+1;
        end
        k = 1;
        while mod(n+1,B^k)==0 && k < L(N)
            q(k,N) = q(k,N)+1;
            k=k+1;
        end
        for k = 1:L(N)-1
            b(k+1,N) = q(k,N)*B^k;
        end
    end
end
end
```

B.4 Integral equation with derivative

```
% Solve the integral equation y(t) = H(t) - d/dt int_0^t f(t-tau)y(tau)dtau,
% , y(0) = 0,  where f(t)=1/sqrt(pi*t), F(s)=1/sqrt(s). Many functions
% H(t) are listed and depend on the chosen analytic solution y(t). The
% latter should have a fast enough decay at zero in order to make the
% convolution smooth enough so that the theoretic convolution quadrature
% order of convergence is obtained. y(t)=sqrt(pi)*t.^(7/2) turned out to be
% a good choice. The Runge-Kutta method is choosen between the first three
% RadauIIA methods. The derivative can be approximated with the first
% 4-stage BDF. The code plots the relative error at final time versus the
% timestep h and approximates the algebraic order of convergence.


function integral_equation_with_derivative_fast

%final time
Tend=4;

%partitioning number
B=10;

%quadrature point number (the quadrature points are 2*K+1)
K=15;

%Runge-Kutta method
[r,e,c,RKA,D]=impliciteuler;
%[r,e,c,RKA,D]=Radau2stages;
%[r,e,c,RKA,D]=Radau3stages;

%Laplace transform of the kernel f=1/sqrt(pi*t)
F=@(t) 1./sqrt(t);
```

```
%analytic solution the integral equation
%sol=@(t) sqrt(t*pi);
%sol=@(t) t*sqrt(pi);
%sol = @(t) sqrt(pi)*t.^2;
%sol = @(t) sqrt(pi)*t.^5;
%sol = @(t) sqrt(pi)*t.^10;
%sol = @(t) sqrt(pi)*t.^20;
sol = @(t) sqrt(pi)*t.^(7/2);

%function H(t)
%H=@(t) pi/2+ sqrt(t*pi);
%H=@(t) 2*sqrt(t)+t*sqrt(pi);
%H=@(t) 8/3*t.^(3/2)+sqrt(pi)*t.^2;
%H=@(t) 256/63*t.^(9/2)+sqrt(pi)*t.^5;
%H=@(t) 262144/46189*t.^(19/2) +sqrt(pi)*t.^10;
%H=@(t) 274877906944/34461632205*t.^(39/2) +sqrt(pi)*t.^20;
H=@(t) 35*pi*t.^3/32+sqrt(pi)*t.^(7/2);


%analytic convolution (just to know)
%conv=@(t) pi*t/2;
%conv=@(t) 4/3*t.^(3/2);
%conv=@(t) 16/15*t.^(5/2);
%conv=@(t) 512/693*t.^(11/2);
%conv=@(t) 524288/969969*t.^(21/2);
%conv=@(t) 549755813888/1412926920405*t.^(41/2);
%conv=@(t) 35*pi*t.^4/128;


%compute the absolute error at final time
time=zeros(1,5);
for i=1:5
    tic;
    [err{i},h(i)]=errore(i,Tend,B,r,e,c,RKA,D,K,H,F,sol);
    time(i)=toc;

end

%plot the relative error at final time versus the timestep h
figure;
loglog(h,[err{1}(end) err{2}(end) err{3}(end) err{4}(end) err{5}(end)],...
    '+-b');
xlabel('log(h)');
ylabel('log(relative error) at t=4');
hold on;
legend('relative error at final time');

%compute the algebraic order of convergence
p=polyfit(log(h),log([err{1}(end) err{2}(end) err{3}(end) err{4}(end) ...
    err{5}(end)]),1);
fprintf('algebraic order of convergence: %f\n',p(1));

end


%compute the relative error at final time Tend. the timestep is 0.5^p, B is
%the partitioning number, r,e,c,RKA,D are related to the Runge-Kutta
%method, H is the function H(t), K is the quadrature number, F is the
%Laplace transform of the kernel and sol is the reference solution
function [err,h]=errore(p,Tend,B,r,e,c,RKA,D,K,H,F,sol)

%timestep
```

```
h=0.5^(p);

%number of iterations
Nite=ceil(Tend/h);

%coefficients of the BDF method of
%order 1
a=[1 1;0 -h]\[0;1];
%order 2
%a=[1 1 1; 0 -h -2*h; 0 h^2/2 2*h^2]\[0;1;0];
%order 3
%M=[1 1 1 1;...
%   0 -h -2*h -3*h;...
%   0 h^2/2 2*h^2 9/2*h^2;...
%   0 -h^3/6 -8/6*h^3 -27/6*h^3];
%a=M\[0;1;0;0];
%order 4
%M=[1 1 1 1 1;...
%   0 -h -2*h -3*h -4*h;...
%   0 h^2/2 2*h^2 9/2*h^2 16/2*h^2;...
%   0 -h^3/6 -8/6*h^3 -27/6*h^3 -4^3/6*h^3;...
%   0 -h^4/24 -(2*h)^4/24 -(3*h)^4/24 -(4*h)^4/24];
%a=M\[0;1;0;0;0];

%the first 2*B weights are computed with the Cauchy integral on the circle
W_0=W_0_cerchio(F,D,h);
W=cell(2*B,1);
for i=1:2*B
    W{i}=W_cerchio(F,D,h,i);
end

%inizialize contour
[w,lambda,Lmax]=contour(B,Nite,K,h);

%precompute the partitions
[b,L]=tinte(Nite,B);

%approximated solution vector: y(i) = y((i-1+c)*h)
y=cell(1,Nite);
y{1}=real((eye(size(W_0))+a(1)*W_0)\H(c*h));


%struct odesol
%odesol{a,b} contains 2*K+1 solutions of the contour l=a at time t=b*h
%necessary only until time (Nite-1)*h
odesol=cell(Lmax,Nite-1);
odesol{Lmax,Nite-1}=[];

%compute the first values of odesol and store some values which are used
%many times
rr=zeros(Lmax,2*K+1);
e0=cell(Lmax,2*K+1);
a2=cell(Lmax,2*K+1);
for l=1:Lmax
    for j=-K:K
        rr(l,j+K+1)=r(h*lambda(l,j+K+1));
        e0{l,j+K+1}=e(0,h*lambda(l,j+K+1));
        odesol{l,1}{1}(j+K+1)=h*e0{l,j+K+1}*y{1};
        a2{l,j+K+1}=(eye(size(RKA))-h*lambda(l,j+K+1)*RKA)\ones(size(c))*F(
              lambda(l,j+K+1));
    end
end
```

```
%vector used for updating odesol
k1=zeros(Lmax,1);
k2=ones(Lmax,1);

%vectors of partial convolution and past values of the convolution
ypast=0;
convpastpast=0;
convpastpastpast=0;
convpastpastpastpast=0;

for i=1:Nite-1

    %store old values of the convolution
    if i>3
        convpastpastpastpast=convpastpastpast;
    end
    if i>2
        convpastpastpast=convpastpast;
    end
    if i>1
        convpastpast=convpast;
    end
    convpast=W_0*y{i}+ypast;

    %compute the partial convolution
    ypast=0;

    %weights approximated on the hyperbola only from the second contour
    if L(i)>1

    for l=2:L(i)
        for j=-K:K
            %those terms are computed separately otherwise Matlab makes an
            %error (smallnumber*smallnumber=hugenumber)
            a1=w(l,j+K+1)*rr(l,j+K+1)^(i-b(l,i));
            a3=odesol{l,b(l,i)}{1}(1,j+K+1);
            ypast=ypast+a1*a2{l,j+K+1}*a3;
        end
    end

    %classic (non-fast) convolution for the first contour (weights
    %approximated with Cauchy integral on the circle)
    for m=b(2,i):i-1
        ypast=ypast+real(W{i-m}*y{m+1});
    end
    else
    for m=0:i-1
        ypast=ypast+real(W{i-m}*y{m+1});
    end
    end

    %BDF order 1
    y{i+1}=real((eye(size(W_0))+W_0*a(1))\(H((i+c)*h)-a(1)*ypast -...
        a(2)*convpast));

    %BDF order 2
    %y{i+1}= real((eye(size(W_0))+W_0*a(1))\(H((i+c)*h)-a(1)*ypast-a(2)...
    %    *convpast-a(3)*convpastpast));

    %BDF order 3
    %y{i+1}= real((eye(size(W_0))+W_0*a(1))\(H((i+c)*h)-a(1)*ypast -...
    %    a(2)*convpast-a(3)*convpastpast-a(4)*convpastpastpast));

    %BDF order 4
    %y{i+1}= real((eye(size(W_0))+W_0*a(1))\(H((i+c)*h)-a(1)*ypast...
    %    -a(2)*convpast-a(3)*convpastpast-a(4)*convpastpastpast...
    %    -a(5)*convpastpastpastpast));

    %update odesol
    if i+1<Nite

        for l=1:Lmax
            if i+1==(2+k1(l))*B^l-1
                %cancel old values and compute the new ones
                if l>1
                    %store a value in the past
                    odesol{l,b(l,i+1)}{1}=odesol{l,b(l,i+1)}{2};
                end
                for j=-K:K
                    odesol{l,i+1}{1}(j+K+1)=rr(l,j+K+1).*odesol{l,i}{2}(j+K
                          +1)+h*e0{l,j+K+1}*y{i+1};
                end
                k1(l)=k1(l)+1;
            else
                %compute new values
                for j=-K:K
                    odesol{l,i+1}{1}(j+K+1)=rr(l,j+K+1).*odesol{l,i}{1}(j+K
                          +1)+h*e0{l,j+K+1}*y{i+1};
                end
                    if size(odesol{l,i},2)==2
                for j=-K:K
                        odesol{l,i+1}{2}(j+K+1)=rr(l,j+K+1).*odesol{l,i}{2}(
                              j+K+1)+h*e0{l,j+K+1}*y{i+1};
                    end
                    end
            end
            if i+1==k2(l)*B^l+1
                 %initialize values for the future
                 for j=-K:K
                    odesol{l,i+1}{2}(j+K+1)=h*e0{l,j+K+1}*y{i+1};
                 end
                 k2(l)=k2(l)+1;
            end
        end
    end
end

%compute the absolute error
for i=1:length(y)
err(i)=abs((y{i}(end)-sol(h*i))/sol(h*i));
end

end


function [r,e,c,RKA,D]=impliciteuler
c=1;
RKA=1;
r=@(z) 1/(1-z);
e=@(n,z) (1-z)^(-n-1);
D=@(z) 1-z;
end
```

```
function [r,e,c,RKA,D]=Radau2stages
b=[3/4 1/4];
c=[1/3; 1];
RKA=[5/12 -1/12;3/4 1/4];
r=@(z) 1+z*b*((eye(2)-z*RKA)\ones(2,1));
e=@(n,z) (1+z*b*((eye(2)-z*RKA)\ones(2,1)))^n*b*inv(eye(2)-z*RKA);
D=@(z) 0.5*[3 1-4*z;-9 4*z+5];
end


function [r,e,c,RKA,D]=Radau3stages
b=[(16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
c=[(4-sqrt(6))/10; (4+sqrt(6))/10; 1];
RKA=[(88-7*sqrt(6))/360 (296-169*sqrt(6))/1800 (-2+3*sqrt(6))/225;
    (296+169*sqrt(6))/1800 (88+7*sqrt(6))/360 (-2-3*sqrt(6))/225;
    (16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
r=@(z) 1+z*b*((eye(3)-z*RKA)\ones(3,1));
e=@(n,z) (1+z*b*((eye(3)-z*RKA)\ones(3,1)))^n*b*inv(eye(3)-z*RKA);
invRKA=inv(RKA);
D=@(z) invRKA -z*(RKA\[1;1;1])*b*invRKA;
end


%compute the first weight W_0 over the circle. F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep
function W_0_cerchio_test=W_0_cerchio(F,D,h)
W_0_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=100;
for l=0:L-1
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_0_cerchio_test=W_0_cerchio_test+1/L*V*diag(F(diag(Diag)))*inv(V);
end
end

%compute the n-th weight W_n over the circle .F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep, n is the index of the n-th weight.
function W_cerchio_test=W_cerchio(F,D,h,n)
W_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=100;
for l=1:L
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_cerchio_test=W_cerchio_test+V*diag(F(diag(Diag)))*inv(V)*exp(-2i*pi*n*
          l/L);
end
W_cerchio_test=rho^(-n)/L*W_cerchio_test;
end


%compute all the hyperbola quadrature weights, the hyperbola quadrature
%points and the number of contours involved. B is the partitioning number,
%Nite is the number of iterations, K is the quadrature point number, h is
```

```
%the timestep
function [w,lambda,Lmax]=contour(B,Nite,K,h)

%compute the biggest number of contour involved
if ceil(log((Nite)/2)/log(B))==log((Nite)/2)/log(B);
    Lmax =log((Nite)/2)/log(B)+1;
else
    Lmax=ceil(log((Nite)/2)/log(B));
end

%set the hyperbola parameters
alpha=1;
d=alpha;
sigma=0;
rho=rho_optimal(B,K,alpha,d);

a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
tau=1/K*a(rho);


w=zeros(Lmax,2*K+1);
lambda=zeros(Lmax,2*K+1);
for l=1:Lmax
    mu=2*pi*d*K*(1-rho)/(2*B^l-2)/h/a(rho);

    %hyperbola function
    hyperbola=@(x) mu*(1-sin(alpha+x*1i))+sigma;
    %hyperbola derivative function
    dhyperbola=@(x) -1i*mu*cos(alpha+x*1i);

    %compute the hyperbola quadrature weights and the hyperbola quadrature
    %point
    for k=-K:K
        w(l,k+K+1)=1i*tau/2/pi*dhyperbola(k*tau);
        lambda(l,k+K+1)=hyperbola(k*tau);
    end
end
end


%compute the optimal value of rho with the strategy described in section
%4.2. B is the partitioning number, K is the quadrature number, alpha and d
%are parameters of the hyperbola
function rho =rho_optimal(B,K,alpha,d)
a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
eps_K=@(rho) exp(-2*pi*d/a(rho)*K);
valorevecchio = 100;
for i=1:99
    rho=i/100;
    valoremuovo=eps*eps_K(rho)^(rho-1)+eps_K(rho)^rho;
    if valoremuovo < valorevecchio
        j=i;
        valorevecchio=valoremuovo;
    end
end
rho=j/100;
end


% compute all the partition times. Nite is the number of iterations. For
% N=2:Nite store in b(:,N-1) the times (N-1)=b(1,N-1)-b(2,N-1), b1-b2,...
% b(L(N-1)-1,N-1))-0=b(L(N),N-1)
```

```
function [b,L]=tinte(Nite,B)
if ceil(log((Nite)/2)/log(B))==log((Nite)/2)/log(B);
    Lmax =log((Nite)/2)/log(B)+1;
else
    Lmax=ceil(log((Nite)/2)/log(B));
end
q=zeros(Lmax+1,Nite-1);
b=zeros(Lmax+1,Nite-1);
L=ones(1,Nite-1);
for N=2:Nite
    N=N-1;
    b(1,N)=N;
    for n =1:N
        if 2*B^L(N) == n+1
            L(N)=L(N)+1;
        end
        k = 1;
        while mod(n+1,B^k)==0 && k < L(N)
            q(k,N) = q(k,N)+1;
            k=k+1;
        end
        for k = 1:L(N)-1
            b(k+1,N) = q(k,N)*B^k;
        end
    end
end
end
```

B.5 Parabolic PDE with impedance boundary conditions

```
function [y,h]=main

%timestep
h_t=0.25;

%choose the meshgrid (6 is the finest)
nMesh=2;

%radius of the internal circle
radius=0.5;

%final time
Tend=4;

%partitioning number
B=10;

%quadrature point number (the quadrature points are 2*K+1)
K=15;

%coefficients of the BDF method of
%order 1
%da=[1 1;0 -h_t]\[0;1];
%order 2
%da=[1 1 1; 0 -h_t -2*h_t; 0 h_t^2/2 2*h_t^2]\[0;1;0];
%order 3
da=[1 1 1 1;0 -h_t -2*h_t -3*h_t;0 h_t^2/2 2*h_t^2 9/2*h_t^2;...
```

```
    0 -h_t^3/6 -8/6*h_t^3 -27/6*h_t^3]\[0;1;0;0];
%order 4
%dM=[1 1 1 1 1;...
%   0 -h_t -2*h_t -3*h_t -4*h_t;...
%   0 h_t^2/2 2*h_t^2 9/2*h_t^2 16/2*h_t^2;...
%   0 -h_t^3/6 -8/6*h_t^3 -27/6*h_t^3 -4^3/6*h_t^3;...
%   0 -h_t^4/24 -(2*h_t)^4/24 -(3*h_t)^4/24 -(4*h_t)^4/24];
%da=dM\[0;1;0;0;0];

%number of iterations
Nite=ceil(Tend/h_t);

%Laplace transform of the kernel f=1/sqrt(pi*t)
F=@(t) -1./sqrt(t);

%Runge-Kutta method
%[r,e,c,RKA,D,m]=impliciteuler;
[r,e,c,RKA,D,m]=Radau2stages;
%[r,e,c,RKA,D,m]=Radau3stages

%the first 2*B weights are computed with the Cauchy integral on the circle
W_0=W_0_cerchio(F,D,h_t);
W=cell(2*B,1);
for i=1:2*B
    W{i}=W_cerchio(F,D,h_t,i);
end

%inizialize contour
[w,lambda,Lmax]=contour(B,Nite,K,h_t);

%precompute the partitions
[b,L]=tinte(Nite,B,Lmax);

%set the quadrature rule (for LehrFEM)
QuadRule = P706();

%load the mesh and compute the stiffnes matrix,the mass matrix on the
%boundary, the degrees of freedom FreeDofs, the Dirichlet nodes
%DirichletNodes, the Robin nodes RobinNodes, the number of coordinates
%nCoordinates , and the mesh Mesh
[A,M,FreeDofs,DNodes,RNodes,nCoordinates,Mesh]=caricamesh(nMesh,radius);


%extend node vectors
extDNodes=zeros(length(DNodes)*m,1);
extFreeDofs=zeros(length(FreeDofs)*m,1);
extRNodes=zeros(length(RNodes)*m,1);
for i=1:m
    extDNodes((1:length(DNodes))+(i-1)*length(DNodes))=DNodes+(i-1)...
        *nCoordinates;
    extRNodes((1:length(RNodes))+(i-1)*length(RNodes))=RNodes+(i-1)...
        *nCoordinates;
    extFreeDofs((1:length(FreeDofs))+(i-1)*length(FreeDofs))=FreeDofs+...
        (i-1)*nCoordinates;
end

%solution vector:u((1:nCoordinates)+(j?1)?nCoordinates,i)=u((i?1+c(j))?h t)
u=zeros(nCoordinates*m,Nite);

%solution at time t=c(:)?h

%first compute Dirichlet nodes
```

```
for i=1:m
    u(DNodes+(i-1)*nCoordinates,1)=32/105/sqrt(pi)*(c(i)*h_t)^(7/2)...
        +log(4)*(c(i)*h_t)^3/6;
end
Fmod=-kron(speye(m),A)*u(:,1);

%then compute solution of FreeDofs
mtemp=kron(speye(m),M)*kron(W_0,speye(nCoordinates));
Z=sparse(kron(speye(m),A(FreeDofs,FreeDofs))-...
    mtemp(extFreeDofs,extFreeDofs)*da(1));
u(extFreeDofs,1)=Z\Fmod(extFreeDofs);

%initialize odesol and set the values at time t=h %odesol{a,b} contains the
%2?K+1 solutions for the contour l=a %at time t=b?h; is necessary only
%until time (Nite?1)?h
[odesol,k1,k2,rr,e0,a2]=ODEsol(Lmax,RNodes,lambda,u,h_t,Nite,e,m,...
    nCoordinates,r,K,RKA,c,F);

%vectors of partial convolution and past values of the convolution
ypast=zeros(m*nCoordinates,1);
convpast=zeros(m*nCoordinates,1);
convpastpast=zeros(m*nCoordinates,1);
convpastpastpast=zeros(m*nCoordinates,1);
convpastpastpastpast=zeros(m*nCoordinates,1);

for i = 1:Nite-1
    %store old values of the convolution
    if i>3
        convpastpastpastpast=convpastpastpast;
    end
    if i>2
        convpastpastpast=convpastpast;
    end
    if i>1
        convpastpast=convpast;
    end
    convpast(extRNodes)=kron(W_0,speye(length(RNodes)))*u(extRNodes,i)...
        +ypast(extRNodes);

    %compute the partial convolution
    [ypast]=calcolaypast(L,i,RNodes,extRNodes,nCoordinates,K,odesol,b,w,...
        m,W,u,a2,rr);

    %impose Dirichlet nodes
    for ii=1:m
        u(DNodes+(ii-1)*nCoordinates,i+1)=32/105/sqrt(pi)*((c(ii)+i)*...
            h_t)^(7/2)+log(4)*((c(ii)+i)*h_t)^3/6;
    end

    Fmod=-kron(speye(m),A)*u(:,i+1);

    %solve the linear system

    % BDF order 1
    %u(extFreeDofs,i+1)=Z\(Fmod(extFreeDofs)+kron(speye(m),...
    %      M(FreeDofs,FreeDofs))*(da(1)*ypast(extFreeDofs)+da(2)*...
    %      convpast(extFreeDofs)));

    % BDF order 2
    %u(extFreeDofs,i+1)=Z\(Fmod(extFreeDofs)+kron(speye(m),...
    %     M(FreeDofs,FreeDofs))*(da(1)*ypast(extFreeDofs)+da(2)*...
    %     convpast(extFreeDofs)+da(3)*convpastpast(extFreeDofs)));
    % BDF order 3
    u(extFreeDofs,i+1)=Z\(Fmod(extFreeDofs)+kron(speye(m),...
        M(FreeDofs,FreeDofs))*(da(1)*ypast(extFreeDofs)+da(2)*...
        convpast(extFreeDofs)+da(3)*convpastpast(extFreeDofs)+da(4)*...
        convpastpastpast(extFreeDofs)));

    % BDF order 4
    %u(extFreeDofs,i+1)=Z\(Fmod(extFreeDofs)+kron(speye(m),...
    %    M(FreeDofs,FreeDofs))*(da(1)*ypast(extFreeDofs)+da(2)*...
    %     convpast(extFreeDofs)+da(3)*convpastpast(extFreeDofs)+da(4)*...
    %     convpastpastpast(extFreeDofs)+da(5)*...
    %     convpastpastpastpast(extFreeDofs)));

    %update odesol
    if i+1<Nite
        [odesol,k1,k2]=aggiornaodesol(odesol,i,Lmax,k1,k2,B,b,RNodes,...
            h_t,u,m,nCoordinates,e0,rr,K);
    end
end

%compute the error in the L2 norm
ErrHandle =@(x) solution(x,Tend);
y = L2Err_LFE(Mesh,real(u((m-1)*nCoordinates+1:end,Nite)),QuadRule,...
    ErrHandle);
h = get_MeshWidth(Mesh);

%plot the approximated solution
plot_LFE(real(u((m-1)*nCoordinates+1:end,Nite)),Mesh);
colorbar;
title('approximation');
%plot the reference solution
plot_LFE(solution(Mesh.Coordinates,Tend),Mesh);
colorbar;
title('exact solution');

end


%reference solution
function y=solution(x,t)
R=sqrt(x(:,1).^2+x(:,2).^2);
y=32/105/sqrt(pi)*t^(7/2)+t^3/6*(log(R)+log(2));
end


%compute all the hyperbola quadrature weights, the hyperbola quadrature
%points and the number of contours involved. B is the partitioning number,
%Nite is the number of iterations, K is the quadrature point number, h is
%the timestep
function [w,lambda,Lmax]=contour(B,Nite,K,h)

%compute the biggest number of contour involved
if ceil(log((Nite)/2)/log(B))==log((Nite)/2)/log(B);
    Lmax =log((Nite)/2)/log(B)+1;
else
    Lmax=ceil(log((Nite)/2)/log(B));
end

%set the hyperbola parameters
alpha=1;
d=alpha;
```

```
sigma=0;
rho=rho_optimal(B,K,alpha,d);

a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
tau=1/K*a(rho);


w=zeros(Lmax,2*K+1);
lambda=zeros(Lmax,2*K+1);
for l=1:Lmax
    mu=2*pi*d*K*(1-rho)/(2*B^l-2)/h/a(rho);

    %hyperbola function
    hyperbola=@(x) mu*(1-sin(alpha+x*1i))+sigma;
    %hyperbola derivative function
    dhyperbola=@(x) -1i*mu*cos(alpha+x*1i);

    %compute the hyperbola quadrature weights and the hyperbola quadrature
    %point
    for k=-K:K
        w(l,k+K+1)=1i*tau/2/pi*dhyperbola(k*tau);
        lambda(l,k+K+1)=hyperbola(k*tau);
    end
end
end


%compute the optimal value of rho with the strategy described in section
%4.2. B is the partitioning number, K is the quadrature number, alpha and d
%are parameters of the hyperbola
function rho =rho_optimal(B,K,alpha,d)
a=@(rho) acosh(2*B/(1-rho)/sin(alpha));
eps_K=@(rho) exp(-2*pi*d/a(rho)*K);
valorevecchio = 100;
for i=1:99
    rho=i/100;
    valoremuovo=eps*eps_K(rho)^(rho-1)+eps_K(rho)^rho;
    if valoremuovo < valorevecchio
        j=i;
        valorevecchio=valoremuovo;
    end
end
rho=j/100;
end


% compute all the partition times. Nite is the number of iterations. For
% N=2:Nite store in b(:,N-1) the times (N-1)=b(1,N-1)-b(2,N-1), b1-b2,...
% b(L(N-1)-1,N-1))-0=b(L(N),N-1)
function [b,L]=tinte(Nite,B,Lmax)
q=zeros(Lmax+1,Nite-1);
b=zeros(Lmax+1,Nite-1);
L=ones(1,Nite-1);
for N=2:Nite
    N=N-1;
    b(1,N)=N;
    for n =1:N
        if 2*B^L(N) == n+1
            L(N)=L(N)+1;
        end
        k = 1;
        while mod(n+1,B^k)==0 && k < L(N)
```

```
            q(k,N) = q(k,N)+1;
            k=k+1;
        end
        for k = 1:L(N)-1
            b(k+1,N) = q(k,N)*B^k;
        end
    end
end
end


function [r,e,c,RKA,D,m]=impliciteuler
m=1;
c=1;
RKA=1;
r=@(z) 1/(1-z);
e=@(n,z) (1-z)^(-n-1);
D=@(z) 1-z;
end


function [r,e,c,RKA,D,m]=Radau2stages
m=2;
b=[3/4 1/4];
c=[1/3; 1];
RKA=[5/12 -1/12;3/4 1/4];
r=@(z) 1+z*b*((eye(2)-z*RKA)\ones(2,1));
e=@(n,z) (1+z*b*((eye(2)-z*RKA)\ones(2,1)))^n*b*inv(eye(2)-z*RKA);
D=@(z) 0.5*[3 1-4*z;-9 4*z+5];
end


function [r,e,c,RKA,D,m]=Radau3stages
m=3;
b=[(16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
c=[(4-sqrt(6))/10; (4+sqrt(6))/10; 1];
RKA=[(88-7*sqrt(6))/360 (296-169*sqrt(6))/1800 (-2+3*sqrt(6))/225;
    (296+169*sqrt(6))/1800 (88+7*sqrt(6))/360 (-2-3*sqrt(6))/225;
    (16-sqrt(6))/36 (16+sqrt(6))/36 1/9];
r=@(z) 1+z*b*((eye(3)-z*RKA)\ones(3,1));
e=@(n,z) (1+z*b*((eye(3)-z*RKA)\ones(3,1)))^n*b*inv(eye(3)-z*RKA);
invRKA=inv(RKA);
D=@(z) invRKA -z*(RKA\[1;1;1])*b*invRKA;
end

%for a given mesh number compute the stiffness matrix, the mass matrix on
%the boundary, the degrees of freedom vector, the Dirichlet nodes vector,
%the Robin nodes vector, the number of coordinates nCoordinates. nMesh is
%the mesh number (must be between 1 and 6) and radius is the internal
%radius of the annulus
function [A,M,FreeDofs,DNodes,RNodes,nCoordinates,Mesh]=caricamesh...
    (nMesh,radius)

Coordinates=sprintf('Coordinates%u.dat',nMesh);
Elements=sprintf('Elements%u.dat',nMesh);
Mesh = load_Mesh(Coordinates,Elements);
Mesh = add_Edges(Mesh);

%set the flags of points on the exterior boundary to -1 and on the interior
%to -2, and collects the degree of freedoms
nCoordinates = size(Mesh.Coordinates,1);
Loc = get_BdEdges(Mesh);
```

```
BE = Mesh.Edges(Loc,:);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);

for i=1:size(Loc,1)
    if (Mesh.Coordinates(BE(i,1),1)^2 +Mesh.Coordinates(BE(i,1),2)^2 > ...
            (radius^2+0.5))
        Mesh.BdFlags(Loc(i))=-1;
    else
        Mesh.BdFlags(Loc(i))=-2;
    end
end

%Extract Dirichlet nodes
for j = -1
    DEdges = Loc(Mesh.BdFlags(Loc) == j);
    DNodes = unique([Mesh.Edges(DEdges,1); Mesh.Edges(DEdges,2)]);
end

%Ectract Robin nodes
for j = -2
    REdges = Loc(Mesh.BdFlags(Loc) == j);
    RNodes = unique([Mesh.Edges(REdges,1); Mesh.Edges(REdges,2)]);
end

%compute the degrees of freedom vector
FreeDofs = setdiff(1:nCoordinates,DNodes);

%assemble the stiffness matrix
A = assemMat_LFE(Mesh,@STIMA_Lapl_LFE);

%compute the mass matrix but just integrating on the inner boundary
M=zeros(size(Mesh.Coordinates,1),size(Mesh.Coordinates,1));
nEdges = size(Mesh.Edges,1);
for i = 1:nEdges
    if (Mesh.BdFlags(i) == -2)
        dist=norm(Mesh.Coordinates(Mesh.Edges(i,1),:)-Mesh.Coordinates...
            (Mesh.Edges(i,2),:));
        M(Mesh.Edges(i,1), Mesh.Edges(i,2))= dist/6;

        % the matrix is symmetric
        M(Mesh.Edges(i,2), Mesh.Edges(i,1)) = M(Mesh.Edges(i,1),...
            Mesh.Edges(i,2));
        M(Mesh.Edges(i,1),Mesh.Edges(i,1))=M(Mesh.Edges(i,1),...
            Mesh.Edges(i,1))+ dist/3;
        M(Mesh.Edges(i,2),Mesh.Edges(i,2))=M(Mesh.Edges(i,2),...
            Mesh.Edges(i,2))+ dist/3;
    end
end
M=sparse(M);
end


%create the struct odesol and compute the first values for t=c*h_t.
%odesol{a,b} contains 2*K+1 solutions of the contour l=a at time t=b*h for
%every Robin node
%k1 and k2 are the vectors used for updating odesol, rr,e0,a2 are values
%which are used many times in the code. Lmax is the number of contours
%involved in the whole algorithm, RNodes is a vector which indicates the
%Robin nodes,lambda are the quadrature point on the hyperbola, u is the
%approximate solution, h_t is the timestep, Nite is the number of
%iterations, nCoordinates is the number of coordinates of FEM, e,m,r,RKA,c
%are related to the Runge-Kutta method, F is the Laplace transform of the
```

```
%kernel
function [odesol,k1,k2,rr,e0,a2]=ODEsol(Lmax,RNodes,lambda,u,h_t,Nite,e,...
    m,nCoordinates,r,K,RKA,c,F)

odesol=cell(Lmax,Nite-1);
odesol{Lmax,Nite-1}=[];

rr=zeros(Lmax,2*K+1);
e0=cell(Lmax,2*K+1);
a2=cell(Lmax,2*K+1);

%compute the first values of odesol and store some values which are used
%many times
for l=1:Lmax
    for ii=1:size(RNodes,1)
        for j=-K:K
            rr(l,j+K+1)=r(h_t*lambda(l,j+K+1));
            e0{l,j+K+1}=e(0,h_t*lambda(l,j+K+1));
            odesol{l,1}{1}(ii,j+K+1)=h_t*e0{l,j+K+1}*u(RNodes(ii)+...
                (0:m-1)'*nCoordinates,1);
            a2{l,j+K+1}=(eye(size(RKA))-h_t*lambda(l,j+K+1)*RKA)\ones(...
                size(c))*F(lambda(l,j+K+1));

        end
    end
end
%vector used for updating odesol
k1=zeros(Lmax,1);
k2=ones(Lmax,1);
end

%update the structure odesol. i is the iteration number, Lmax is the number
%of all the contours involved in the whole algorithm, k1 and k2 are used
%for updating odesol, B is the partitioning number, b,m ar related to
%the Runge-Kutta method, RNodes is the vector which indicates the Robin
%nodes, h_t is the time step, u is the approximated solution, nCoordinates
%is the number of coordinates of FEM, e0,rr are given by the function
%ODEsol,K is the quadrature point number
function [odesol,k1,k2]=aggiornaodesol(odesol,i,Lmax,k1,k2,B,b,RNodes,...
    h_t,u,m,nCoordinates,e0,rr,K)
        for l=1:Lmax
            if i+1==(2+k1(l))*B^l-1
                %cancel old values and compute the new ones
                if l>1
                    odesol{l,b(l,i+1)}{1}=odesol{l,b(l,i+1)}{2};
                end
                for ii=1:size(RNodes,1)
                    for j=-K:K
                        odesol{l,i+1}{1}(ii,j+K+1)=rr(l,j+K+1).*...
                            odesol{l,i}{2}(ii,j+K+1)+h_t*e0{l,j+K+1}*...
                            u(RNodes(ii)+(0:m-1)'*nCoordinates,i+1);
                    end
                end
                k1(l)=k1(l)+1;
            else
                %compute new values
                for ii=1:size(RNodes,1)
                    for j=-K:K
                        odesol{l,i+1}{1}(ii,j+K+1)=rr(l,j+K+1).*...
                            odesol{l,i}{1}(ii,j+K+1)+h_t*e0{l,j+K+1}*...
                            u(RNodes(ii)+(0:m-1)'*nCoordinates,i+1);
                    end
                    if size(odesol{l,i},2)==2
                        for j=-K:K
                            odesol{l,i+1}{2}(ii,j+K+1)=rr(l,j+K+1).*...
                                odesol{l,i}{2}(ii,j+K+1)+h_t*e0{l,j+K+1}*...
                                u(RNodes(ii)+(0:m-1)'*nCoordinates,i+1);
                        end
                    end
                end
            end
            if i+1==k2(l)*B^l+1
                %initialize values for the future
                for ii=1:size(RNodes,1)
                    for j=-K:K
                        odesol{l,i+1}{2}(ii,j+K+1)=h_t*e0{l,j+K+1}*...
                            u(RNodes(ii)+(0:m-1)'*nCoordinates,i+1);
                    end
                end
                k2(l)=k2(l)+1;
            end
        end
end


%compute the partial (truncated) convolution.
%L is the vector containing the informations about which contours are used,
%i is the iteration number, RNodes is the vector which indicates the Robin
%nodes, extRNodes is RNodes extended with respect to the number of stages
%of the Runge-Kutta method, nCoordinates is the number of coordinates of
%FEM, K is the quadrature point number, odesol contains the solution of the
%ODE's, b,m are related to the Runge-Kutta method, w contains the
%quadrature wheights of the hyperbola, W are the convolution wheights
%computed on the circle, u is the approximated solution, a2,rr have been
%computed in the function ODEsol
function [ypast]=calcolaypast(L,i,RNodes,extRNodes,nCoordinates,K,odesol...
    ,b,w,m,W,u,a2,rr)
ypast=zeros(nCoordinates*m,1);

%weights approximated on the hyperbola only from the second contour
if L(i)>1
    for l=2:L(i)
        for ii=1:size(RNodes,1)
            for j=-K:K
                %those terms are computed separately otherwise Matlab makes
                %an error (smallnumber*smallnumber=hugenumber)
                a1=w(l,j+K+1)*rr(l,j+K+1)^(i-b(l,i));
                a3=odesol{l,b(l,i)}{1}(ii,j+K+1);
                ypast(RNodes(ii)+(0:m-1)'*nCoordinates)=ypast(RNodes(ii)...
                    +(0:m-1)'*nCoordinates)+a1*a2{l,j+K+1}*a3;
            end
        end
    end

    %classic (non-fast) convolution for the first contour (weights
    %approximated with Cauchy integral on the circle)o
    for mm=b(2,i):i-1
        ypast(extRNodes)=ypast(extRNodes)+kron(W{i-mm},speye(length(...
            RNodes)))*u(extRNodes,mm+1);
    end
else
    for mm=0:i-1
        ypast(extRNodes)=ypast(extRNodes)+kron(W{i-mm},speye(length(...
            RNodes)))*u(extRNodes,mm+1);
    end
end

end



%compute the first weight W_0 over the circle. F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep
function W_0_cerchio_test=W_0_cerchio(F,D,h)
W_0_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=100;
for l=0:L-1
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_0_cerchio_test=W_0_cerchio_test+1/L*V*diag(F(diag(Diag)))*inv(V);
end
end


%compute the n-th weight W_n over the circle .F is the Laplace transform
%of the kernel, D is the delta matrix of the Runge-Kutta method, h is the
%timestep, n is the index of the n-th weight.
function W_cerchio_test=W_cerchio(F,D,h,n)
W_cerchio_test=0;
%radius of the circle
rho=0.6;
%number of quadrature points on the circle
L=100;
for l=1:L
    A=D(rho*exp(2i*pi*l/L))/h;
    [V,Diag] = eig(A);
    W_cerchio_test=W_cerchio_test+V*diag(F(diag(Diag)))*inv(V)*exp(-2i*pi*n*
          l/L);
end
W_cerchio_test=rho^(-n)/L*W_cerchio_test;
end
```

References

[1] Lehel Banjai. Multistep and multistage convolution quadrature for the wave equation: algorithms and experiments. *SIAM J. Sci. Comput.*, 32(5):2964–2994, 2010.

[2] Lehel Banjai, Christian Lubich, and Jens Melenk. Runge–kutta convolution quadrature for operators arising in wave propagation. *Numerische Mathematik*, pages 1–20, 2011. 10.1007/s00211-011-0378-z.

[3] Annalisa Buffa, Yvon Maday, and Francesca Rapetti. A sliding mesh-mortar method for a two dimensional eddy currents model of electric engines. *M2AN Math. Model. Numer. Anal.*, 35(2):191–228, 2001.

[4] E. Hairer, Ch. Lubich, and M. Schlichte. Fast numerical solution of non-linear Volterra convolution equations. *SIAM J. Sci. Statist. Comput.*, 6(3):532–541, 1985.

[5] E. Hairer and G. Wanner. *Solving ordinary differential equations. II*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1996. Stiff and differential-algebraic problems.

[6] Nicholas J. Higham. *Functions of matrices*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2008. Theory and computation.

[7] Roger A. Horn and Charles R. Johnson. *Topics in matrix analysis*. Cambridge University Press, Cambridge, 1991.

[8] María López-Fernández, Christian Lubich, Cesar Palencia, and Achim Schädle. Fast Runge-Kutta approximation of inhomogeneous parabolic equations. *Numer. Math.*, 102(2):277–291, 2005.

[9] María López-Fernández, César Palencia, and Achim Schädle. A spectral order method for inverting sectorial Laplace transforms. *SIAM J. Numer. Anal.*, 44(3):1332–1350 (electronic), 2006.

[10] C. Lubich. Convolution quadrature and discretized operational calculus. I. *Numer. Math.*, 52(2):129–145, 1988.

[11] C. Lubich. Convolution quadrature and discretized operational calculus. II. *Numer. Math.*, 52(4):413–425, 1988.

[12] Ch. Lubich and A. Ostermann. Runge-Kutta methods for parabolic equations and convolution quadrature. *Math. Comp.*, 60(201):105–131, 1993.

[13] Christian Lubich and Achim Schädle. Fast convolution for nonreflecting boundary conditions. *SIAM J. Sci. Comput.*, 24(1):161–182 (electronic), 2002.

[14] Walter Rudin. *Functional analysis.* Higher mathematics series. McGraw-Hill Inc., 1973.

[15] Achim Schädle, María López-Fernández, and Christian Lubich. Fast and oblivious convolution quadrature. *SIAM J. Sci. Comput.*, 28(2):421–438 (electronic), 2006.