algorithms

Article

A Compact FEM Implementation for Parabolic Integro-Differential Equations in 2D

Gujji Murali Mohan Reddy [1], Alan B. Seitenfuss [2,†], Débora de Oliveira Medeiros [2,†], Luca Meacci [2,†], Milton Assunção [2,†] and Michael Vynnycky [3,4,*]

1 Department of Mathematics, Birla Institute of Technology & Science, Pilani, Hyderabad Campus, Telangana 500078, India; gmuralireddy1984@gmail.com
2 Department of Applied Mathematics and Statistics, Institute of Mathematical and Computer Sciences, University of São Paulo at São Carlos, P.O. Box 668, São Carlos 13560-970, São Paulo, Brazil; alanbour@usp.br (A.B.S.); deboramedeiros@usp.br (D.d.O.M.); luca.meacci@usp.br (L.M.); assuncao@icmc.usp.br (M.A.)
3 Division of Processes, Department of Materials Science and Technology, KTH Royal Institute of Technology, Brinellvägen 23, 100 44 Stockholm, Sweden; michael.vynnycky@ul.ie
4 Department of Mathematics and Statistics, University of Limerick, Limerick V94 T9PX, Ireland
* Correspondence: michaelv@kth.se
† These authors contributed equally to this work.

Received: 27 August 2020; Accepted: 18 September 2020; Published: 24 September 2020

Abstract: Although two-dimensional (2D) parabolic integro-differential equations (PIDEs) arise in many physical contexts, there is no generally available software that is able to solve them numerically. To remedy this situation, in this article, we provide a compact implementation for solving 2D PIDEs using the finite element method (FEM) on unstructured grids. Piecewise linear finite element spaces on triangles are used for the space discretization, whereas the time discretization is based on the backward-Euler and the Crank–Nicolson methods. The quadrature rules for discretizing the Volterra integral term are chosen so as to be consistent with the time-stepping schemes; a more efficient version of the implementation that uses a vectorization technique in the assembly process is also presented. The compactness of the approach is demonstrated using the software Matrix Laboratory (MATLAB). The efficiency is demonstrated via a numerical example on an L-shaped domain, for which a comparison is possible against the commercially available finite element software COMSOL Multiphysics. Moreover, further consideration indicates that COMSOL Multiphysics cannot be directly applied to 2D PIDEs containing more complex kernels in the Volterra integral term, whereas our method can. Consequently, the subroutines we present constitute a valuable open and validated resource for solving more general 2D PIDEs.

Keywords: parabolic integro-differential equations; backward-Euler; Crank–Nicolson; quadrature rules; Volterra integral term

1. Introduction

Parabolic integro-differential equations (PIDEs) arise in various physical contexts, such as heat conduction in materials with memory [1–3], the compression of poro-viscoelastic media [4], nuclear reactor dynamics [5], epidemic phenomena in biology [6] and drug absorption/release [7,8]. Existing and unique results from such kinds of problems can be found in [9–12].

Inevitably, such equations have to be solved numerically and various approaches have been developed for this, such as spectral methods, spline and collocation, the method of lines and finite-element methods [13–21]. Amongst these, one of the most attractive is arguably the finite-element method (FEM), since it can be applied to irregularly-shaped domains in higher dimensions; moreover, convergence (both a priori and a posteriori) analysis for such problems is already well-established, and for this one may refer to [9,10,12,22–28]. As regards existing software platforms which could be used for a finite-element implementation without having to resort to lengthy programming from first principles, the main choices are freefem++ [29], deal.II [30], iFEM [31], FEniCS [32], DUNE [33], FEATool [34], MATLAB's Partial Differential Equation Toolbox [35] and COMSOL Multiphysics [36]; however, it is not immediately obvious how to solve PIDEs with any of these. This is perhaps surprising in the case of MATLAB, which has become increasingly popular in recent years for the numerical solutions of various partial differential equations (PDEs) in structural mechanics and heat transfer [37–47], and has proven to be an excellent tool for academic education in general [40].

Against that background, the purpose of this article is to provide a compact formulation for solving PIDEs in two spatial dimensions (2D) using FEM on unstructured grids (we made the resulting software, which we programmed in MATLAB, openly available). This is done by extending the already available FEM formulations for PDEs [40,43,44,48,49] to PIDEs. In particular, we:

- Use piecewise linear finite element spaces on triangles for the space discretization;
- Vase the time discretizations on the backward-Euler and the Crank–Nicolson methods;
- Choose the quadrature rules for discretizing the Volterra integral term so as to be consistent with the time-stepping schemes.

In addition, we demonstrate that for PDEs, vectorization techniques can be used to speed up the sets of code for PIDEs. Importantly, these sets of code can be easily extended to more complex geometries, and to PIDEs with different kernels in the Volterra integral term. A comparison with results obtained using COMSOL Multiphysics for a test example shows that our method is competitive. Moreover, further consideration indicates that COMSOL Multiphysics cannot be directly applied to 2D PIDEs containing more complex kernels in the Volterra integral term, whereas our method can.

The rest of the paper is organized as follows. The model problem and its weak formulation are described in Section 2. The discretization is sketched in Section 3. The data representation of the triangulation is described in Section 4, together with the discrete space. The assembly of the stiffness matrix, mass matrix and load vector are presented in Section 5. Improved pseudo-codes using vectorization is presented in Section 6, and an implementation in MATLAB is discussed in Section 7. In Section 8, a comparison of the results obtained using the different sets of code is presented; results obtained using COMSOL Multiphysics are also compared. Conclusions are drawn in Section 9.

2. Model Problem and Weak Formulation

Given a Lebesgue measurable set Ω, we denote by $L^p(\Omega)$, $1 \le p \le \infty$ the Lebesgue spaces. When $p = 2$, the space $L^2(\Omega)$ is equipped with an inner product $\langle \cdot, \cdot \rangle$. For an integer $m > 0$, we use the standard notation for Sobolev spaces $W^{m,p}(\omega)$ with $1 \le p \le \infty$. When $p = 2$, we denote $W^{m,2}(\Omega)$ by $H^m(\Omega)$. The function space $H_0^1(\Omega)$ consists of elements from $H^1(\Omega)$ that vanish on the boundary of Ω, where the boundary values are to be interpreted in the sense of a trace.

Consider the following initial-boundary value problem for a linear PIDE of the form

$$\frac{\partial u}{\partial t}(\boldsymbol{x}, t) + \mathcal{A}u(\boldsymbol{x}, t) = \int_0^t \mathcal{B}(t, s)u(\boldsymbol{x}, s)\mathrm{d}s + f(\boldsymbol{x}, t), \quad (\boldsymbol{x}, t) \in \Omega \times (0, T], \tag{1}$$

$$u(\boldsymbol{x}, t) = 0, \quad (\boldsymbol{x}, t) \in \partial\Omega \times [0, T], \tag{2}$$

$$u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega. \tag{3}$$

Here, $0 < T < \infty$ and $\Omega \subset \mathbb{R}^2$ is a bounded Lipschitz domain with boundary $\partial\Omega$. Further, $\mathcal{A}u(\boldsymbol{x}, t) = -\Delta u(\boldsymbol{x}, t)$ and

$$\mathcal{B}(t, s)u(\boldsymbol{x}, s) = -\nabla \cdot (B(t, s)\nabla u(\boldsymbol{x}, s)), \tag{4}$$

where Δ denotes the Laplacian and ∇ denotes the spatial gradient. We assume that the coefficient matrix $B(t,s) = \{b_{ij}(x;t,s)\}$ is 2×2 in $L^\infty(\Omega)^{2\times 2}$. If we assume the initial function $u_0(x)$ to be in $H^2(\Omega) \cap H_0^1(\Omega)$, the source term $f(x,t)$ to be in $L^2(0;T;L^2(\Omega))$ and

$$\max_{\bar{\Omega}\times\{0\le s\le t\le T\}} |\frac{\partial}{\partial x} b_{i,j}(x;t,s)| < \infty,$$

then the problem (1) admits a unique solution

$$u \in L^2(0,T;H^2(\Omega)\cap H_0^1(\Omega)) \cap H^1(0,T;L^2(\Omega)).$$

The existence results are discussed in detail in Chapter 2 of [9]. For regularity and stability results of such problems, please refer to [10,28].

Let $a(\cdot,\cdot)\ :\ H_0^1(\Omega)\times H_0^1(\Omega) \to \mathbb{R}$ be the bilinear form corresponding to the operator $\mathcal{A}$ defined by

$$a(\phi,\psi) := \langle \nabla\phi, \nabla\psi\rangle \qquad \forall\ \phi,\psi \in H_0^1(\Omega).$$

Similarly, let $b(t,s;\cdot,\cdot)$ be the bilinear form corresponding to the operator $\mathcal{B}(t,s)$ defined on $H_0^1(\Omega)\times H_0^1(\Omega)$ by

$$b(t,s;\phi(s),\psi) := \langle B(t,s)\nabla\phi(s), \nabla\psi\rangle \qquad \forall\ \phi(s),\psi \in H_0^1(\Omega).$$

Then, the weak formulation of the problem (1) may be stated as follows: find $u : [0,T] \to H_0^1(\Omega)$ such that, for all $t \in (0,T]$,

$$\begin{aligned} \int_\Omega \frac{\partial u}{\partial t}\phi\,\mathrm{d}x' + a(u,\phi) &= \int_0^t b(t,s;u(s),\phi)\,\mathrm{d}s + \int_\Omega f\phi\,\mathrm{d}x' \ \ \forall\phi \in H_0^1(\Omega), \\ u(\cdot,0) &= u_0. \end{aligned} \tag{5}$$

3. Galerkin Discretizations

Let $\mathcal{T}$ denote a regular partition of the domain $\Omega \in \mathbb{R}^2$ into disjoint triangles K of diameter h_K such that:

- $\cup_{K_i\in\mathcal{T}}\overline{K_i} = \overline{\Omega}$, and any pair of triangles intersect along a complete edge, at a vertex, or not at all;
- No vertex of any triangle lies on the interior of a side of another triangle.

Let $\mathbb{V}$ be the finite element space defined by

$$\mathbb{V} := H_0^1(\Omega) \cap \{\phi \in C(\overline{\Omega}) :\ \phi|_K \in \mathbb{P}_1\ \forall K \in \mathcal{T}_h\},$$

where $\mathbb{P}_1$ is the space of polynomials in d variables of degree at most 1.

With $\mathcal{N} = \{z_1,\cdots,z_M\}$ as the set of nodes of $\mathcal{T}$, we consider the nodal basis $\mathcal{B} = \{\phi_1,\cdots,\phi_M\}$, where the hat function $\phi_l \in \mathbb{P}_1$ is characterized by $\phi_l(z_k) = \delta_{kl}$, where δ_{kl} is the Kronecker delta.

Let $0 = t_0 < t_1 < \ldots < t_N = T$ be a partition of $[0,T]$ with $\tau_n := t_n - t_{n-1}$. Set $f^n(\cdot) = f(\cdot,t_n)$ for $t = t_n,\ n \in [0,1,\ldots,N]$. Now, for $n = 1,2,\cdots,N$, we define

$$\partial v^n := \frac{v^n - v^{n-1}}{\tau_n}, \quad t_{n-1/2} := \frac{t_n + t_{n-1}}{2} \quad \text{and} \quad v^{n-1/2} := \frac{v^n + v^{n-1}}{2}.$$

In this article, we consider uniform time steps, i.e., $\tau = \tau_1 = \ldots = \tau_N$.

3.1. Backward-Euler Scheme

Let I_h be the Lagrange interpolant corresponding to $\mathbb{V}$. Then, the fully discrete backward-Euler scheme may be stated as follows: given U^0, where $U^0 = I_h u_0$, find $U^n \in \mathbb{V},\ n \in [1:N]$ such that

$$\int_\Omega \partial U^n \phi \mathrm{d}\boldsymbol{x}' + a(U^n, \phi) = \sigma^n(b(t_n; U, \phi)) + \int_\Omega f^n \phi \mathrm{d}\boldsymbol{x}', \quad \forall \phi \in \mathbb{V}, \tag{6}$$

where

$$\sigma^n(b(t_n; v, \phi)) = \langle \sum_{j=0}^{n-1} \tau_{j+1} B(t_n, t_j) \nabla v(t_j), \nabla \phi \rangle, \tag{7}$$

if the left rectangular rule is used to discretize the integral term, or

$$\sigma^n(b(t_n; v, \phi)) = \langle \sum_{j=1}^{n} \tau_j B(t_n, t_j) \nabla v(t_j), \nabla \phi \rangle, \tag{8}$$

if the right rectangular rule is used. In any case, both are consistent with the backward-Euler scheme and the order of accuracy is same.

The discrete problem (6) becomes: find $\mathbf{U}^n = \sum_{j=1}^{M} u(z_j, t_n)\phi_j(\boldsymbol{x}) \in \mathbb{R}^m$ such that

$$(\mathbf{M} + \tau \mathbf{A})\mathbf{U}^n = \mathbf{M}\mathbf{U}^{n-1} + \tau^2 \mathbf{A}(B(t_n, t_0)\mathbf{U}^0 + \ldots + B(t_n, t_{n-1})\mathbf{U}^{n-1}) + \tau \mathbf{b}^n, \tag{9}$$

where the mass matrix $\mathbf{M}$, the stiffness matrix $\mathbf{A}$ and the load vector $\mathbf{b}^n$ are given by

$$\mathbf{M} = \{M_{i,j}\} = \int_\Omega \phi_j \phi_i \mathrm{d}\boldsymbol{x}', \tag{10}$$

$$\mathbf{A} = \{A_{i,j}\} = \int_\Omega \nabla \phi_j \cdot \nabla \phi_i \mathrm{d}\boldsymbol{x}', \tag{11}$$

$$\mathbf{b}^n = \{b_i^n\} = \int_\Omega f^n \phi_i \mathrm{d}\boldsymbol{x}', \tag{12}$$

respectively.

3.2. Crank–Nicolson Scheme

We now state the Crank–Nicolson scheme as follows: given U^0, where $U^0 = I_h u_0$, find $U^n \in \mathbb{V},\ n \in [1:N]$ such that

$$\int_\Omega \partial U^n \phi \mathrm{d}x' + a(U^{n-1/2}, \phi) = \sigma^n(b(t_{n-1/2}; U, \phi)) + \int_\Omega f^{n-1/2} \phi \mathrm{d}\boldsymbol{x}' \quad \forall \phi \in \mathbb{V}, \tag{13}$$

where

$$\sigma^n(b(t_{n-1/2}; v, \phi)) := \left\langle \sum_{j=0}^{n-2} \frac{\tau_{j+1}}{2} \left[B(t_{n-1/2}, t_j) \nabla v^j + B(t_{n-1/2}, t_{j+1}) \nabla v^{j+1} \right], \nabla \phi \right\rangle$$

$$+ \left\langle \frac{\tau_n}{4} \left[B(t_{n-1/2}, t_{n-1}) \nabla v^{n-1} + B(t_{n-1/2}, t_{n-1/2}) \nabla v^{n-1/2} \right], \nabla \phi \right\rangle.$$

Here, the trapezoidal rule is used to discretize the integral term in order to be consistent with the Crank–Nicolson scheme.

Using (13), we obtain the following matrix equation for the Crank–Nicolson scheme:

$$
\begin{aligned}
(\mathbf{M}+\frac{\tau}{2}\mathbf{A}-\frac{\tau^2}{8}B(t_{\frac{1}{2}},t_1)\mathbf{A})\mathbf{U}^1 &= (\mathbf{M}-\frac{\tau}{2}\mathbf{A})\mathbf{U}^0+\frac{3\tau^2}{8}B(t_{\frac{1}{2}},t_0)\mathbf{A}\mathbf{U}^0+\tau\hat{b}^1,\\
(\mathbf{M}+\frac{\tau}{2}\mathbf{A}-\frac{\tau^2}{8}B(t_{n-1/2},t_n)\mathbf{A})\mathbf{U}^n &= (\mathbf{M}-\frac{\tau}{2}\mathbf{A})\mathbf{U}^{n-1}+\tau\hat{b}^n\\
+\frac{\tau^2}{2}\mathbf{A}\left[B(t_{n-1/2},t_0)\mathbf{U}^0+2\sum_{i=1}^{n-2}B(t_{n-1/2},t_i)\mathbf{U}^i\right] &+\frac{7\tau^2}{8}B(t_{n-1/2},t_{n-1})A\mathbf{U}^{n-1},
\end{aligned}
\tag{14}
$$

for $n > 1$ and where $\mathbf{M}$ is the mass matrix, $\mathbf{A}$ is the stiffness matrix and $\hat{b}_n$ is the load vector:

$$
\hat{b}^n = \{b_i\} = \int_\Omega \frac{f^n+f^{n-1}}{2}\phi_i \mathrm{d}\boldsymbol{x}'. \tag{15}
$$

3.3. Remark on More General Boundary Conditions

Problems involving more general boundary conditions can also be addressed in a similar manner. Suppose that the domain boundary is split into two components Γ_D and Γ_N such that $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$, assuming that Γ_D is closed and has a nonzero measure. For sufficiently regular Neumann boundary data g_N, consider a problem posed with mixed-type boundary conditions:

$$
\begin{aligned}
\frac{\partial u}{\partial t}(\boldsymbol{x},t)+\mathcal{A}u(\boldsymbol{x},t) &= \int_0^t \mathcal{B}(t,s)u(\boldsymbol{x},s)ds+f(\boldsymbol{x},t),\ \ (\boldsymbol{x},t)\in\Omega\times(0,T],\\
u(\boldsymbol{x},t) &= g(\boldsymbol{x}),\ \ (\boldsymbol{x},t)\in\Gamma_D\times[0,T],\\
\frac{\partial u}{\partial n}(\boldsymbol{x},t) &= g_N(\boldsymbol{x}),\ \ (\boldsymbol{x},t)\in\Gamma_N\times[0,T],\\
u(\boldsymbol{x},0) &= u_0(\boldsymbol{x}),\ \ \boldsymbol{x}\in\Omega.
\end{aligned}
\tag{16}
$$

We assume there is a function $u_D(\boldsymbol{x},t)\in H^1(\Omega)$ for $t\in[0,T]$ such that $u_D(\boldsymbol{x},t)=g(\boldsymbol{x})$ for $\boldsymbol{x}\in\partial\Omega, t\in[0,T]$. Then, the weak form of this problem reads: find $u = w + u_D : [0,T] \to H^1_{g,\Gamma_D}(\Omega), w \in H^1_{0,\Gamma_D}(\Omega)$ such that

$$
\begin{aligned}
\int_\Omega \frac{\partial w}{\partial t}\phi\,\mathrm{d}\boldsymbol{x}'+a(w,\phi) = &-\int_\Omega\frac{\partial u_D}{\partial t}\phi\,\mathrm{d}\boldsymbol{x}'+\int_0^t b(t,s;w(s),\phi)\,\mathrm{d}s - a(u_D,\phi)\\
&+\int_0^t b(t,s;u_D(s),\phi)\,\mathrm{d}s+\int_\Omega f\phi\,\mathrm{d}\boldsymbol{x}'+\int_{\Gamma_N} g_N\phi\,\mathrm{d}S\\
&-\int_0^t\left(\int_{\Gamma_N}B(t,s)g_N(s)\phi\,\mathrm{d}S\right)\mathrm{d}s\ \ \forall\phi\in H^1_0(\Omega), t\in(0,T],\\
u(\cdot,0) = u_0,&
\end{aligned}
$$

where

$$
H^1_{g,\Gamma_D}(\Omega) := \{w\in H^1(\Omega) : w = g \text{ on } \Gamma_D\}
$$

and

$$
H^1_{0,\Gamma_D}(\Omega) := \{w\in H^1(\Omega) : w = 0 \text{ on } \Gamma_D\}.
$$

4. Data Representation

To fix ideas ahead of benchmark computations, consider the L-shaped domain $\Omega = [-1,1]^2\backslash[-1,0]\times[0,1]$ with a closed polygonal boundary $\Gamma = \Gamma_D$, as shown in Figure 1, where nodes are represented as blue squares and the exterior continuous line represents the boundary with Dirichlet boundary condition Γ_D; this domain was chosen as it is often used for benchmarking FEM computations [41,44,50,51].

In order to represent the relationship between node and spatial coordinates, and between elements and nodes, it is necessary to define arrays that store this information. Thus, we define the array `c4n`, which represents the node-coordinate relation, and the array `n4e`, which represents the element-node relation to generate the domain; these are given in Table 1. In addition, the Dirichlet boundary conditions are represented by a data structure `dir` which contains nodal information; this is also given in Table 1. The first columns of the arrays `c4n`, `n4e` and `dir` represent the node number, element number and edge number, respectively.

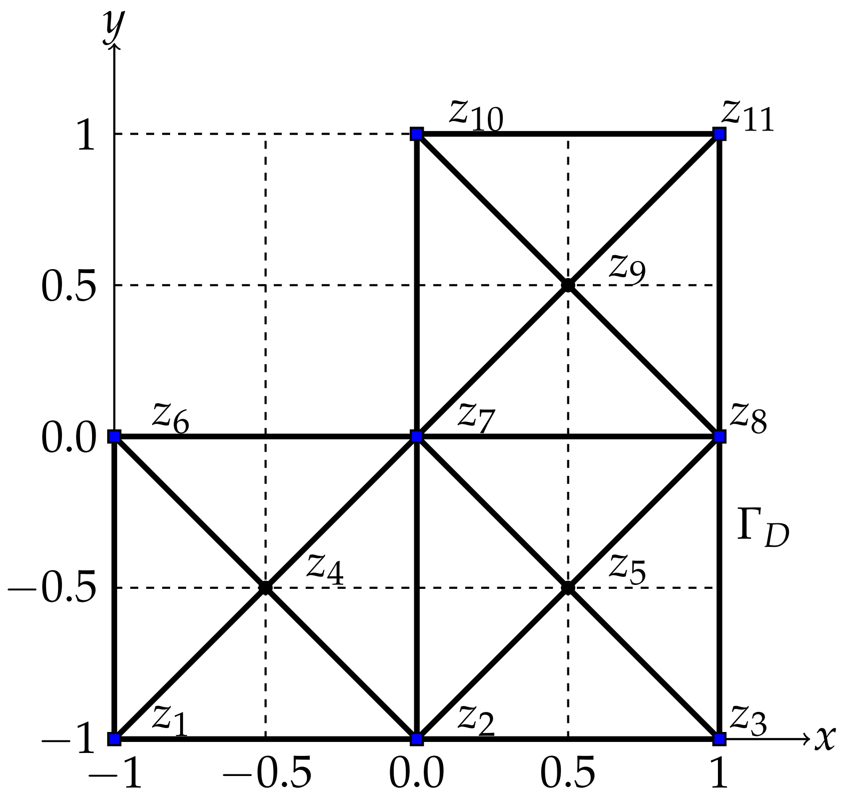

Figure 1. Domain representation.

Table 1. Arrays `c4n`, `n4e` and `dir`.

	c4n	
1	-1.0	-1.0
2	0.0	-1.0
3	1.0	-1.0
4	-0.5	-0.5
5	0.5	-0.5
6	-1.0	0.0
7	0.0	0.0
8	1.0	0.0
9	0.5	0.5
10	0.0	1.0
11	1.0	1.0

Table 1. *Cont.*

n4e			
1	1	2	4
2	2	3	5
3	3	8	5
4	8	7	5
5	7	2	5
6	7	4	2
7	4	7	6
8	6	1	4
9	6	7	9
10	9	7	11
11	9	11	10
12	9	10	6

dir		
1	1	2
2	2	3
3	3	8
4	8	11
5	11	10
6	10	7
7	7	6
8	6	1

5. Assembly

In this section, we assemble the stiffness matrix, the mass matrix, the load vector and the initial and boundary conditions. The assembly is standard and builds on the ideas for PDEs presented in [40,44]; however, we include it here for completeness, ensuring that the standard routines and the non-standard ones related to PIDEs are available in one location. In what follows, we take $\boldsymbol{x} = (x, y)$ and let (x_1, y_1), (x_2, y_2) and (x_3, y_3) be the vertices of an element K and ϕ_1, ϕ_2 and ϕ_3 the corresponding local basis functions in $\mathbb{V}$. The sets of code which make use of the subroutines presented in this section are given in Appendix A.

For the sets of pseudocode that follow, we first need to establish a common notation; this is given in Table 2.

Table 2. Some common functions and operations.

$A(:)$	All the elements of A, treated as a single column
$A(:, j)$	j-th column of A
$A(i, :)$	i-th row of A
$A == k$	Indices (i, j) where $A(i, j) = k$
A^T	Transpose of A
$A(\alpha, \beta)$	Entry of A at position (α, β)
$a \leftarrow b$	Assignment
$A * B$	Matrix multiplication
$A. * B$	Element-wise multiplication
$A./B$	Element-wise division
$A \backslash B$	Return, without repeating, elements of A which are not in B
ACCUMARRAY(x, y, s)	Return an array with size s (vector containing the dimensions of the output). The values in the output are sums of the values in y having identical subscripts in x
DET(A)	Determinant of A
I_d	$d \times d$ identity matrix
NCOL(A)	Number of columns in A
INV(A)	Inverse of A
NROW(A)	Number of rows in A

Table 2. *Cont.*

$\mathrm{ONES}(m,n)$	$m \times n$ matrix with all entries equal to ones
$\mathrm{REPMAT}(A,m,n)$	Returns $m \times n$ block matrix with each block equal to A
$\mathrm{RESHAPE}(A,m,n)$	Reshape A into $m \times n$ matrix provided $m * n =$ NCOL(A)*NROW(A)
$\mathrm{SPARSE}(m,n)$	$m \times n$ sparse matrix of zeros
$\mathrm{SUM}(A,k)$	Sum of elements of A along the index k
$\mathrm{UNIQUE}(A)$	Return entries of A without repeating
$\mathrm{ZEROS}(m,n)$	$m \times n$ matrix with all entries equal to zero
$[,]$	Horizontal concatenation
$[;]$	Vertical concatenation

5.1. Assembling the Stiffness Matrix

Since

$$\phi_j(x_k, y_k) = \delta_{jk}, \qquad j,k = 1,2,3, \tag{17}$$

a simple calculation reveals

$$\phi_j(x,y) = \det \begin{bmatrix} 1 & x & y \\ 1 & x_{j+1} & y_{j+1} \\ 1 & x_{j+2} & y_{j+2} \end{bmatrix} \Big/ \det \begin{bmatrix} 1 & x_j & y_j \\ 1 & x_{j+1} & y_{j+1} \\ 1 & x_{j+2} & y_{j+2} \end{bmatrix}. \tag{18}$$

Hence, we obtain

$$\nabla \phi_j(x,y) = \frac{1}{\mathcal{K}} \begin{bmatrix} y_{j+1} - y_{j+2} \\ x_{j+2} - x_{j+1} \end{bmatrix}. \tag{19}$$

Here, the indices are to be understood modulo 3, and $\mathcal{K}$ is the area of triangle K. Thus, using (11), the entries of the stiffness matrix can be computed as

$$A_{jk} = \int_K \nabla \phi_j (\nabla \phi_k)^T \mathrm{d}\mathbf{x}' = \frac{1}{\mathcal{K}} (y_{j+1} - y_{j+2}, x_{j+2} - x_{j+1}) \begin{bmatrix} y_{k+1} - y_{k+2} \\ x_{k+2} - x_{k+1} \end{bmatrix}, \tag{20}$$

with indices modulo 3. This can be written as

$$A = GG^T, \quad G := \begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{21}$$

The pseudocode given in Algorithm 1 computes the local stiffness matrix contributions, based on Equations (17)–(21).

Algorithm 1 Computation of local stiffness matrix contributions.

```
1: procedure STIMA($c4n$)
2:     $d \leftarrow \mathrm{NCOL}(c4n)$
3:     $M_1 \leftarrow [\mathrm{ONES}(1, d+1); c4n^T]$
4:     $M_2 \leftarrow [\mathrm{ZEROS}(1, d); I_d]$
5:     $G \leftarrow \mathrm{INV}(M_1) * M_2$
6:     $P_1 \leftarrow [\mathrm{ONES}(1, d+1); c4n^T]$
7:     $S \leftarrow \frac{\mathrm{DET}(P_1)*G*G^T}{\prod_{j=1}^{d} j}$
8:     return $S$
9: end procedure
```

Using the local contributions, one can compute the global contributions using the pseudocode given in Algorithm 2.

Algorithm 2 Computation of the global contributions.

```
1: procedure StiffAssemb($n4e, c4n$)
2:     $r \leftarrow \mathrm{NROW}(c4n)$
3:     $A \leftarrow \mathrm{SPARSE}(r, r)$
4:     for $j \leftarrow 1$ to $r$, do
5:         $A(n4e(j,:), n4e(j,:)) \leftarrow A(n4e(j,:), n4e(j,:)) + \mathrm{STIMA}(c4n(n4e(j,:),:))$
6:     end for
7:     return $A$
8: end procedure
```

5.2. Assembling the Mass Matrix

The entries of the mass matrix M can be computed as

$$M_{jk} = \int_K \phi_j \phi_k \mathrm{d}\boldsymbol{x}'. \tag{22}$$

For triangular, piecewise affine elements, we obtain

$$\int_K \phi_j \phi_k \mathrm{d}\boldsymbol{x}' = \frac{1}{24} \det \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}; \tag{23}$$

the pseudocode for M, based on Equations (22) and (23), is given in Algorithm 3.

Algorithm 3 Assembling the mass matrix.

```
1: procedure MassAssemb($n4e, c4n$)
2:     $r \leftarrow \mathrm{NROW}(c4n)$
3:     $M \leftarrow \mathrm{SPARSE}(r, r)$
4:     for $j \leftarrow 1$ to $r$ do
5:         $X_1 \leftarrow [[1\ 1\ 1]; (c4n(n4e(j,:),:))^T]$
6:         $X_2 \leftarrow [[2\ 1\ 1]; [1\ 2\ 1]; [1\ 1\ 2]]$
7:         $M(n4e(j,:), n4e(j,:)) \leftarrow M(n4e(j,:), n4e(j,:)) + (\frac{1}{24} * \mathrm{DET}(X_1) * X_2)$
8:     end for
9:     return $M$
10: end procedure
```

5.3. Assembling the Load Vector

The volume forces, which are calculated using Algorithm 4, are used for assembling the load vector. Using the value of f at the center of gravity (x_S, y_S) of K, the integral $\int_K f \phi_j \mathrm{d}\boldsymbol{x}'$ is approximated by

$$\int_K f \phi_j \mathrm{d}\boldsymbol{x}' \approx \frac{1}{6} \det \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} f(x_S, y_S). \tag{24}$$

In addition, the load vector contains a contribution from the integral terms containing the kernel B; this is given in Algorithm 5.

Algorithm 4 Source function.

```
1: procedure F($\mathbf{x}, t$)
2:     $VolumeForce \leftarrow (1 - 2 * t). * \pi^2. * \exp(-\pi^2 * t). * \sin(\pi. * \mathbf{x}(:,1)). * \sin(\pi. * \mathbf{x}(:,2))$
3:     return $VolumeForce$
4: end procedure
```

Algorithm 5 Coefficient function in the kernel.

```
1: procedure INTGRAL($t, s$)
2:     $BTS \leftarrow \exp(-\pi^2 * (t - s))$
3:     return $BTS$
4: end procedure
```

5.4. Boundary and Initial Conditions

The sets of pseudocode for the boundary and initial conditions given by Equations (2) and (3), respectively, are given in Algorithms 6 and 7, respectively.

Algorithm 6 Boundary conditions.

```
1: procedure UD($\mathbf{x}, \mathbf{t}$)
2:     $DBV(\mathbf{x}(:,1) == -1) \leftarrow 0$
3:     $DBV(\mathbf{x}(:,1) == 0) \leftarrow 0$
4:     $DBV(\mathbf{x}(:,1) == 1) \leftarrow 1$
5:     $DBV(\mathbf{x}(:,2) == -1) \leftarrow 0$
6:     $DBV(\mathbf{x}(:,2) == 0) \leftarrow 0$
7:     $DBV(\mathbf{x}(:,2) == 1) \leftarrow 0$
8:     return $DBV$
9: end procedure
```

Algorithm 7 Initial conditions.

```
1: procedure U0($c4n$)
2:     $\mathbf{x} \leftarrow c4n(:,1)$
3:     $\mathbf{y} \leftarrow c4n(:,2)$
4:     $u_0 \leftarrow \sin(\pi * \mathbf{x}) * \sin(\pi * \mathbf{y})$
5:     return $u_0$
6: end procedure
```

6. Improved Assembly

To improve the implementation, we present in this section an optimization of the algorithm that switches away from a looping approach towards the utilization of vector tools, available in software such as MATLAB. A brief description of these improvements follows. Similar approaches can be found in [40,43], and the sets of code supplied are modified versions of those given in [52]; they are included for completeness. The sets of pseudocode which make use of the subroutines presented in this section are given in Appendix B.

6.1. Assembling the Stiffness Matrix

The vectorization of the code for the stiffness matrix is done by using the commands SPARSE and RESHAPE, which allow us to store in the vectors `I` and `J` the indices related to the contributions of each node of each element for building the stiffness matrix A. This provides the possibility to evaluate the stiffness matrix A in one line, i.e., without a loop, as shown in Algorithm 8.

Algorithm 8 Assembling the stiffness matrix with vectorization.

```
 1: procedure StiffAssemb($n4e, nC, area4, d21, d31, nE$)
 2:     $I \leftarrow \mathrm{RESHAPE}(n4e(:,[1\ 2\ 3\ 1\ 2\ 3\ 1\ 2\ 3])^T, 9 * nE, 1)$
 3:     $J \leftarrow \mathrm{RESHAPE}(n4e(:,[1\ 1\ 1\ 2\ 2\ 2\ 3\ 3\ 3])^T, 9 * nE, 1)$
 4:     $a \leftarrow (\mathrm{SUM}(d21. * d31, 2)./area4)^T$
 5:     $b \leftarrow (\mathrm{SUM}(d31. * d31, 2)./area4)^T$
 6:     $c \leftarrow (\mathrm{SUM}(d21. * d21, 2)./area4)^T$
 7:     $A \leftarrow [-2 * a + b + c; a - b; a - c; a - b; b; -a; a - c; -a; c]$
 8:     $A \leftarrow \mathrm{SPARSE}(I, J, A(:), nC, nC)$
 9:     return $A$
10: end procedure
```

6.2. Assembling the Mass Matrix

Similarly to the previous paragraph, we can build the mass matrix M by using the SPARSE command. We remark that, in this case, the evaluation of the matrix is reduced to the calculation of only two vectors, as shown in Algorithm 9.

Algorithm 9 Assembling the mass matrix with vectorization.

```
1: procedure MassAssemb($nC, n4e, area$)
2:     $I \leftarrow n4e([1\ 2\ 3\ 1\ 2\ 3\ 1\ 2\ 3],:)$
3:     $J \leftarrow n4e([1\ 1\ 1\ 2\ 2\ 2\ 3\ 3\ 3],:)$
4:     $A_6 \leftarrow \frac{1}{6} * area^T$
5:     $A_{12} \leftarrow \frac{1}{12} * area^T$
6:     $K_g \leftarrow [A_6; A_{12}; A_{12}; A_{12}; A_6; A_{12}; A_{12}; A_{12}; A_6]$
7:     $M \leftarrow \mathrm{SPARSE}(I(:), J(:), K_g(:), nC, nC)$
8:     return $M$
9: end procedure
```

6.3. Assembling the Load Vector

In order to assemble the right-hand side, we refer to what we have done in Section 5.3 considering the volume forces and the center of gravity. Here, however, the improvement consists of neglecting the loop by inserting the MATLAB command ACCUMARRAY, which creates an array of given size, where the values are collected through the REPMAT command, which in this case returns a 3×1 array containing copies of the corresponding evaluation of the element.

6.4. Boundary and Initial Conditions

For the vectorized version also, the sets of pseudocode for the boundary and initial conditions are given in Algorithms 6 and 7, respectively.

7. Overview of an Implementation in Matlab

A minimal MATLAB implementation of the presented algorithm is available from Supplementary Materials. There are six zipfiles which contain the necessary MATLAB files therein:

- `BE_LRR_unvectorized.zip` (backward-Euler, left rectangular rule, unvectorized);
- `BE_LRR_vectorized.zip` (backward-Euler, left rectangular rule, vectorized);
- `BE_RRR_unvectorized.zip` (backward-Euler, right rectangular rule, unvectorized);
- `BE_RRR_vectorized.zip` (backward-Euler, right rectangular rule, vectorized);
- `CN_unvectorized.zip` (Crank–Nicolson, unvectorized);
- `CN_vectorized.zip` (Crank–Nicolson, vectorized).

To run the demo program, uncompress the desired zip file and run the `Main.m` file; this:

- Specifies the geometry;
- Computes the solution;
- Plots the solution.

In particular, the computation of the solution is carried out by `SCHEME_NAME_PIDE.m`, which sets the initial conditions via `U0.m`, and assembles the stiffness matrix and the mass matrix via `StiffAssemb.m` and `MassAssemb.m`, respectively; thereafter, `solve_SCHEME_NAME.m` updates the boundary conditions via `Ud.m`, assembles the load vector via `f.m` and obtains the solution at the current time step. Furthermore, note that whilst most of the subroutines are the same in both the vectorized and unvectorized versions of the code, those for `StiffAssemb.m` and `MassAssemb.m` are necessarily slightly different, in line with the discussion in Sections 5 and 6.

8. Numerical Results

In this section, we study a numerical example and compare the results obtained using different versions of the MATLAB-based sets of code; in addition, solutions were obtained using COMSOL Multiphysics. For the example, we consider the L-shaped domain $\Omega = [-1,1]^2 \backslash [-1,0] \times [0,1]$ mentioned earlier; take

$$f(x,y,t) = (1-2t)\pi^2 u(x,y,t) \tag{25}$$

and

$$B(t,s) = \exp(-\pi^2(t-s)), \tag{26}$$

which is a kernel that occurs in different settings in many of the references given earlier, and others [2–4,7,14,53,54]; this leads to the following exact solution for u:

$$u(x,y,t) = \exp(-\pi^2 t)\sin(\pi x)\sin(\pi y). \tag{27}$$

Figure 2a shows the exact solution to the problem, whereas Figure 2b corresponds to the backward-Euler approximation, where the left rectangular rule is applied to treat the Volterra integral term. Figure 2c,d show the solutions obtained using the backward-Euler approximation with the right rectangular rule and the Crank–Nicolson approximation with the trapezoidal rule, respectively, to treat the Volterra integral term. In addition, Figure 3 shows a comparison of the analytical solution with the solution computed by COMSOL Multiphysics with 66142 P1 elements for $t = T, y = -0.5$, where $T = 0.1$; as expected, the agreement is very good.

To illustrate the advantage of the vectorized code for each numerical scheme over the unvectorized one, which uses *for* loops for the assembly process, we have presented a comparison of the runtimes with respect to the number of elements, N. This is given in Table 3, which also shows the runtimes for COMSOL Multiphysics. Note that all sets of code were run on an Intel Core i7 Notebook with a 1.7 GHz processor and 8 GB of RAM. We can note from Table 3 that vectorized versions of the MATLAB sets of code presented in this article on the finest mesh get close to COMSOL's performance. Nevertheless, there are several factors behind the results. On the one hand, in the developed MATLAB sets of code, we discretize the Volterra integral term; as a result, one has to store the memory term values at all of the time discretization points. On the other hand, because of the particular form of the kernel in (26), and as explained in Appendix C, COMSOL Multiphysics does not need to evaluate the integral at all; even so, the difference in computational time is not decisive. However, and as also explained in Appendix C, we would not have been able to use COMSOL Multiphysics at all in this way if the kernel had not had a separable form, whereas the sets of code presented in this article do not have this limitation.

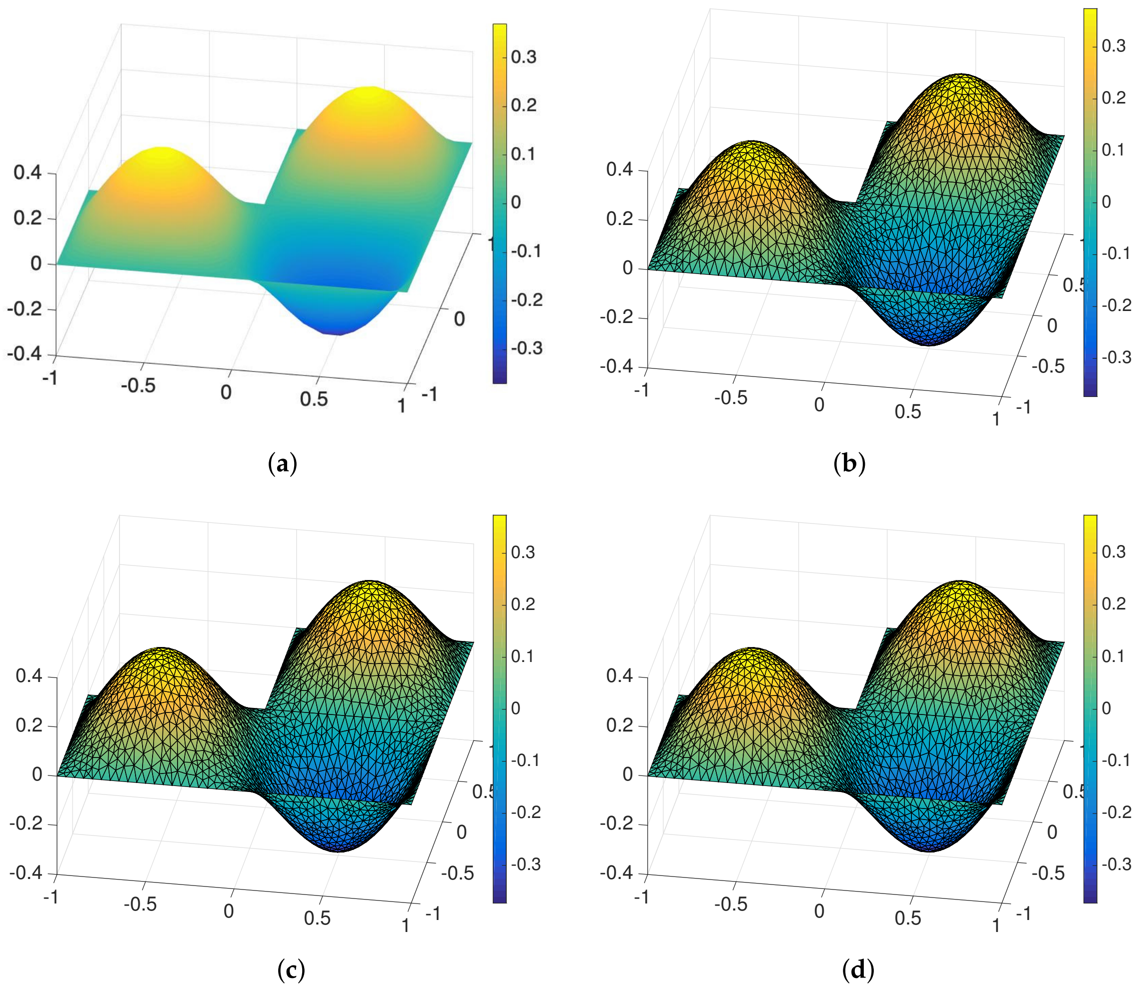

Figure 2. (**a**) The exact solution; (**b**) the backward-Euler FEM solution with the left rectangular quadrature rule applied to approximate the Volterra integral term; (**c**) the backward-Euler FEM solution with the right rectangular quadrature rule applied to approximate the Volterra integral term; (**d**) the Crank–Nicolson FEM solution with the trapezoidal quadrature rule applied to approximate the Volterra integral term. All plots are for $T = 0.1$, and the FEM solutions were computed using P1 elements with $\tau = 0.00125$ and $h = 0.05$.

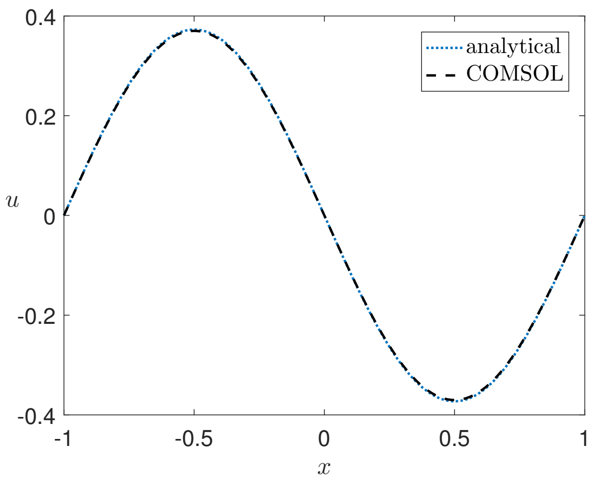

Figure 3. Comparison of the analytical solution with the solution computed by COMSOL Multiphysics with 66,142 elements for $T = 0.1, y = -0.5$.

In addition, we define an efficiency factor, C_{eff}, given by $C_{eff} = t_{unvec}/t_{vec}$, where t_{unvec} is the runtime of the unvectorized code and t_{vec} is the runtime of the vectorized code; the results for this are shown in Table 4. From the latter, we observe that refining the mesh increases the efficiency factor C_{eff}. This shows that the relative performance of the vectorized code improves significantly as the number of elements is increased.

Table 3. Quantitative information on the runtimes of the unvectorized and vectorized MATLAB sets of code with different numerical schemes, and the same for code from COMSOL Multiphysics, when solving the PIDE. Abbreviations: BELR, backward-Euler scheme with left rectangular rule; BERR, backward-Euler scheme with right rectangular rule; CN, Crank–Nicolson scheme with trapezoidal rule; N, number of elements; t_{unvec}^{SCHEME}, runtime of the unvectorized code in seconds involving loops in the assembly process, using a particular method $SCHEME$ (= BELR, BERR or CN); t_{vec}^{SCHEME}, runtime of the vectorized code in seconds without loops in the assembly process, using a particular method $SCHEME$; t_C, runtime for COMSOL Multiphysics.

h	N	$t_{unvec}^{\mathbf{BELR}}$ [s]	$t_{vec}^{\mathbf{BELR}}$ [s]	$t_{unvec}^{\mathbf{BERR}}$ [s]	$t_{vec}^{\mathbf{BERR}}$ [s]	$t_{unvec}^{\mathbf{CN}}$ [s]	$t_{vec}^{\mathbf{CN}}$ [s]	t_C [s]
0.2	258	0.81	0.77	0.82	0.77	0.83	0.77	0.31
0.1	978	0.98	0.81	1.00	0.81	1.04	0.82	0.55
0.05	4086	2.06	1.08	1.96	1.08	2.13	1.08	1.73
0.025	16,406	10.26	4.36	10.51	4.40	10.93	4.41	6.93
0.0125	66,142	114.40	38.04	112.86	38.24	114.42	38.36	34.13

Table 4. The efficiency factor, C_{eff}, for different MATLAB-based sets of code and different space meshes with a fixed time step $(\tau = 0.00125)$.

h	N	$C_{eff}^{\mathbf{BELR}}$	$C_{eff}^{\mathbf{BERR}}$	$C_{eff}^{\mathbf{CN}}$
0.2	258	1.06	1.06	1.07
0.1	978	1.22	1.23	1.27
0.05	4086	1.91	1.82	1.97
0.025	16,406	2.35	2.39	2.48
0.0125	66,142	3.01	2.95	2.98

9. Conclusions

In this paper, we have presented a compact implementation using the finite-element method for solving linear PIDEs in arbitrary 2D geometries, both in terms of sets of pseudocode and in terms of a MATLAB code, which we have made openly available; we note that such a code is not available in other competing FEM software, such as freefem++ or deal.II. In addition, it was found that a vectorized version of the MATLAB code solves the model problem considered around three times more quickly than the unvectorized code on finer meshes. Furthermore, because of the particular form of the kernel in the Volterra integral term, the model problem could be represented in a form that was amenable to solution using the commercial finite-element software COMSOL Multiphysics. For this particular problem, in which the kernel has a separable form, COMSOL Multiphysics slightly outperforms even our vectorized code when using a mesh of around 66,000 elements.

Future development of this work would involve extending the sets of code to handle non-linear integral terms [55] and to three dimensions.

Supplementary Materials: The sets of MATLAB code used to generate the results presented in this paper are available online at https://github.com/LucaMeacci/efficient-PIDE-2D.

Author Contributions: Conceptualization, G.M.M.R.; methodology, G.M.M.R.; software, G.M.M.R., A.B.S., D.d.O.M. and L.M.; validation, M.A. and M.V.; formal analysis, G.M.M.R.; investigation, all; writing—original draft preparation, G.M.M.R. and M.V.; writing—review and editing, G.M.M.R. and M.V.; supervision, G.M.M.R. and M.V.; project administration, G.M.M.R. and M.V.; funding acquisition, G.M.M.R. and M.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) grant numbers 2016/19648-9, 2017/11428-2 and 2018/07643-8 and the Programa de Excelência Académica (Proex)—Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code PROEX-9740044/D. The APC was funded by KTH Royal Institute of Technology.

Acknowledgments: The sixth author acknowledges the award of a visiting researcher grant from the University of São Paulo.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. Unvectorized Pseudocode

In this section, the unvectorized sets of pseudocode are presented.

- **Pseudo-code for Algorithms A1–A6 the backward-Euler scheme with the left rectangular rule, corresponding to Equation (9) and employing (7):**

Algorithm A1 Backward-Euler scheme with left rectangular rule.

Main function

```
procedure BELRFEMPIDE($c4n, n4e, dir, N, dt$)
    $r \leftarrow \text{NROW}(c4n)$
    $U \leftarrow \text{ZEROS}(r, N+1)$
    $FN \leftarrow \text{UNIQUE}([1\ 2\ \ldots r]) \backslash \text{UNIQUE}(dir)$
    $A \leftarrow \text{STIFFASSEMB}(n4e, c4n)$
    $B \leftarrow \text{MASSASSEMB}(n4e, c4n)$
    $U(:,1) \leftarrow \text{U0}(c4n)$
    for $n \leftarrow 2$ to $(N+1)$, do
        $U(:,n) \leftarrow \text{SOLVEBELRR}(c4n, n4e, dir, dt, FN, A, B.n, U)$
    end for
    return $U$
end procedure
```

Auxiliary function

```
procedure SOLVEBELRR($c4n, n4e, dir, dt, FN, A, B, n, U$)
    $r \leftarrow \text{NROW}(c4n)$
    $b \leftarrow \text{SPARSE}(r, 1)$
    for $j \leftarrow 1$ to $\text{NROW}(n4e)$, do
        $M_1 \leftarrow [[1\ 1\ 1]; c4n(n4e(j,:),:)^T]$
        $S \leftarrow \text{SUM}(c4n(n4e(j,:),:))$
        $b(n4e(j,:)) \leftarrow b(n4e(j,:)) + \text{DET}(M_1) * dt * \frac{1}{6} * \text{F}(\frac{S}{3}, (n-1)*dt)$
    end for
    $Int \leftarrow dt^2 * A * U(:,1) * \text{INTGRAL}((n-1)*dt, 0)$
    if $n > 2$, then
        for $k \leftarrow 2$ to $(n-1)$, do
            $Int \leftarrow Int + \big(dt^2 * A * U(:,k) * \text{INTGRAL}((n-1)*dt, (k-1)*dt)\big)$
        end for
    end if
    $b \leftarrow b + (B * U(:,n-1)) + Int$
    $u \leftarrow \text{SPARSE}(r, 1)$
    $u_d \leftarrow \text{UNIQUE}(dir)$
    $u(u_d) \leftarrow \text{UD}(c4n(u_d,:), (n-1)*dt)$
    $b \leftarrow b - (dt * A + B) * u$
    $u(FN) \leftarrow \text{INV}(dt * A(FN,FN) + B(FN,FN)) * b(FN)$
    $solBELR \leftarrow u$
    return $solBELR$
end procedure
```

- **Code for the backward-Euler scheme with the right rectangular rule, corresponding to Equation (9) and employing (8):**

Algorithm A2 Backward-Euler scheme with right rectangular rule.

Main function

```
procedure BERRFEMPIDE($c4n, n4e, dir, N, dt$)
    $r \leftarrow$ NROW($c4n$)
    $U \leftarrow$ ZEROS($r, N+1$)
    $FN \leftarrow$ UNIQUE([1 2 ... $r$])\UNIQUE($dir$)
    $A \leftarrow$ STIFFASSEMB($n4e, c4n$)
    $B \leftarrow$ MASSASSEMB($n4e, c4n$)
    $U(:,1) \leftarrow$ U0($c4n$)
    for $n \leftarrow 2$ to $(N+1)$, do
        $U(:,n) \leftarrow$ SOLVEBERRR($c4n, n4e, dir, dt, FN, A, B.n, U$)
    end for
    return $U$
end procedure
```

Auxiliary function

```
procedure SOLVEBERRR($c4n, n4e, dir, dt, FN, A, B, n, U$)
    $r \leftarrow$ NROW($c4n$)
    $b \leftarrow$ SPARSE($r, 1$)
    for $j \leftarrow 1$ to NROW($n4e$), do
        $M_1 \leftarrow [[1\ 1\ 1]; c4n(n4e(j,:),:)^T]$
        $S \leftarrow$ SUM($c4n(n4e(j,:),:)$)
        $b(n4e(j,:)) \leftarrow b(n4e(j,:)) +$ DET$(M_1) * dt * \frac{1}{6} * \mathrm{F}(\frac{S}{3}, (n-1)*dt)$
    end for
    $Int \leftarrow 0$
    if $n > 2$, then
        $Int \leftarrow dt^2 * A * U(:,1) *$ INTGRAL$((n-1)*dt, dt)$
        for $k \leftarrow 2$ to $(n-2)$, do
            $Int \leftarrow Int + \left(dt^2 * A * U(:,k) * \mathrm{INTGRAL}((n-1)*dt, k*dt)\right)$
        end for
    end if
    $b \leftarrow b + (B * U(:,n-1)) + Int$
    $u \leftarrow$ SPARSE($r, 1$)
    $u_d \leftarrow$ UNIQUE($dir$)
    $u(u_d) \leftarrow$ UD$(c4n(u_d,:), (n-1)*dt)$
    $b \leftarrow b - \left(dt * A + B + dt^2 * A * \mathrm{INTGRAL}((n-1)*dt, (n-1)*dt)\right) * u$
    $M_1 \leftarrow dt * A(FN,FN) + B(FN,FN) + dt^2 * A(FN,FN) * \mathrm{INTGRAL}((n-1)*dt, (n-1)*dt)$
    $u(FN) \leftarrow$ INV$(M_1) * b(FN)$
    $solBERR \leftarrow u$
    return $solBERR$
end procedure
```

- **Code for Crank–Nicolson scheme with trapezoidal rule, corresponding to Equation (14):**

Algorithm A3 Crank–Nicolson scheme with the trapezoidal rule.

Main function

```
procedure CNFEMPIDE($c4n, n4e, dir, N, dt$)
    $r \leftarrow \mathrm{NROW}(c4n)$
    $U \leftarrow \mathrm{ZEROS}(r, N+1)$
    $FN \leftarrow \mathrm{UNIQUE}([1\ 2\ \ldots r]) \backslash \mathrm{UNIQUE}(dir)$
    $A \leftarrow \textsc{StiffAssemb}(n4e, c4n)$
    $B \leftarrow \textsc{MassAssemb}(n4e, c4n)$
    $U(:,1) \leftarrow \mathrm{U0}(c4n)$
    for $j \leftarrow 2$ to $(N+1)$, do
        $U(:,j) \leftarrow \textsc{solve}\mathrm{CN}(c4n, n4e, dir, dt, FN, n, U, A, B)$
    end for
    return $U$
end procedure
```

Auxiliary function

```
procedure solveCN($c4n, n4e, dir, dt, FN, n, U, A, B$)
    $r \leftarrow \mathrm{NROW}(c4n)$
    $b \leftarrow \mathrm{SPARSE}(r, 1)$
    for $j \leftarrow 1$ to $\mathrm{NROW}(n4e)$, do
        $X_1 \leftarrow \mathrm{DET}([[1\ 1\ 1]; c4n(n4e(j,:),:)^T])$
        $X_2 \leftarrow \mathrm{SUM}(c4n(n4e(j,:),:))$
        $X_3 \leftarrow \mathrm{SUM}(c4n(n4e(j,:),:))$
        $b(n4e(j,:)) \leftarrow b(n4e(j,:)) + X_1 * dt * \frac{1}{12} * (\mathrm{F}(X_2/3, (n-1)*dt) + \mathrm{F}(X_3/3, (n-2)*dt))$
    end for
    $Int \leftarrow \frac{1}{8}.*dt^2 * A * U(:,1) * (2 * \textsc{intgral}((n-1.5)*dt, 0) + \textsc{intgral}((n-1.5)*dt, (n-1.5)*dt))$
    if $n > 2$, then
        $Int \leftarrow \frac{1}{2} * dt^2 * A * U(:,1) * \textsc{intgral}((n-1.5)*dt, 0)$
                    $+\frac{1}{8} * dt^2 * A * U(:,n-1) * (\textsc{intgral}((n-1.5)*dt, (n-1.5)*dt)$
                                        $+6 * \textsc{intgral}((n-1.5)*dt, (n-2)*dt))$
        for $k \leftarrow 2$ to $(n-2)$, do
            $Int \leftarrow Int + dt^2 * A * U(:,k) * \textsc{intgral}((n-1.5)*dt, (k-1)*dt)$
        end for
    end if
    $b \leftarrow b + (B - \frac{dt}{2} * A) * U(:,n-1) + Int$
    $u \leftarrow \mathrm{SPARSE}(r, 1)$
    $u_d \leftarrow \mathrm{UNIQUE}(dir)$
    $u(u_d) \leftarrow \textsc{Ud}(c4n(u_d,:), (n-1)*dt)$
    $b \leftarrow b - \Big(\Big(\frac{dt}{2} - \frac{dt^2}{8} * \textsc{intgral}((n-1.5)*dt, (n-1)*dt)\Big) * A + B\Big) * u$
    $M_1 \leftarrow \Big(\Big(\frac{dt}{2} - \frac{dt^2}{8} * \textsc{intgral}((n-1.5)*dt, (n-1)*dt)\Big) * A(FN, FN) + B(FN, FN)\Big)$
    $u(FN) \leftarrow \mathrm{INV}(M1) * b(FN)$
    $solCN \leftarrow u$
    return $solCN$
end procedure
```

Appendix B. Vectorized Pseudocode

In this section, the vectorized sets of pseudocode are presented.

- **Code for the backward-Euler scheme with the left rectangular rule, corresponding to Equation (9) and employing (7):**

Algorithm A4 Vectorized version of backward-Euler scheme with the left rectangular rule.

Main function

```
procedure BELRFEMPIDE($c4n, n4e, dir, N, dt$)
    $nE \leftarrow$ NROW($n4e$)
    $nC \leftarrow$ NROW($c4n$)
    $U \leftarrow$ ZEROS($nC, N+1$)
    $FN \leftarrow$ UNIQUE($[1\ 2\ \ldots nC]$)\UNIQUE($dir$)
    $c1 \leftarrow c4n(n4e(:,1),:)$
    $d21 \leftarrow c4n(n4e(:,2),:) - c1$
    $d31 \leftarrow c4n(n4e(:,3),:) - c1$
    $area4 \leftarrow 2 * (d21(:,1).*d31(:,2) - d21(:,2).*d31(:,1))$
    $A \leftarrow$ STIFFASSEMB($n4e, nC, area4, d21, d31, nE$)
    $B \leftarrow$ MASSASSEMB($nC, n4e^T, \frac{1}{4} * area4$)
    $U(:,1) \leftarrow$ U0($c4n$)
    for $j \leftarrow 2$ to $(N+1)$, do
        $U(:,j) \leftarrow$ SOLVEBELRR($c4n, n4e, dir, dt, FN, A, B, n, U, c1, d21, d31, area4$)
    end for
    return $U$
end procedure
```

Auxiliary function

```
procedure SOLVEBELRR($c4n, n4e, dir, dt, FN, A, B, n, U, c1, d21, d31, area4$)
    $r \leftarrow$ NROW($c4n$)
    $fsT \leftarrow$ F($c1 + \frac{d21+d31}{3}, (n-1)*dt$)
    $L \leftarrow$ REPMAT($dt.*area4.*fsT.*\frac{1}{12}, 3, 1$)
    $b \leftarrow$ ACCUMARRAY($n4e(:), L, [r\ 1]$)
    $Int \leftarrow dt^2 * A * U(:,1) *$ INTGRAL($(n-1)*dt, 0$)
    if $n > 2$, then
        for $k \leftarrow 2$ to $(n-1)$, do
            $Int \leftarrow Int + dt^2 * A * U(:,k) *$ INTGRAL($(n-1)*dt, (k-1)*dt$)
        end for
    end if
    $b \leftarrow b + B * U(:,n-1) + Int$
    $u \leftarrow$ SPARSE($r, 1$)
    $u_d \leftarrow$ UNIQUE($dir$)
    $u(u_d) \leftarrow$ UD($c4n(u_d,:), (n-1)*dt$)
    $b \leftarrow b - (dt * A + B) * u$
    $M_1 \leftarrow dt * A(FN, FN) + B(FN, FN)$
    $u(FN) \leftarrow$ INV($M_1$) $* b(FN)$
    $solBELR \leftarrow u$
    return $solBELR$
end procedure
```

- **Code for the backward-Euler scheme with the right rectangular rule, corresponding to Equation (9) and employing (8):**

Algorithm A5 Vectorized version of backward-Euler scheme with the left rectangular rule.

Main function

```
procedure BERRFEMPIDE($c4n, n4e, dir, N, dt$)
    $nE \leftarrow$ NROW($n4e$)
    $nC \leftarrow$ NROW($c4n$)
    $U \leftarrow$ ZEROS($nC, N+1$)
    $FN \leftarrow$ UNIQUE($[1\ 2\ \dots nC]$)\UNIQUE($dir$)
    $c1 \leftarrow c4n(n4e(:,1),:)$
    $d21 \leftarrow c4n(n4e(:,2),:) - c1$
    $d31 \leftarrow c4n(n4e(:,3),:) - c1$
    $area4 \leftarrow 2 * (d21(:,1).*d31(:,2) - d21(:,2).*d31(:,1))$
    $A \leftarrow$ STIFFASSEMB($n4e, nC, area4, d21, d31, nE$)
    $B \leftarrow$ MASSASSEMB($nC, n4e^T, \frac{1}{4} * area4$)
    $U(:,1) \leftarrow$ U0($c4n$)
    for $j \leftarrow 2$ to $(N+1)$, do
        $U(:,j) \leftarrow$ SOLVEBERRR($c4n, n4e, dir, dt, FN, A, B, n, U, c1, d21, d31, area4$)
    end for
    return $U$
end procedure
```

Auxiliary function

```
procedure SOLVEBERRR($c4n, n4e, dir, dt, FN, A, B, n, U, c1, d21, d31, area4$)
    $r \leftarrow$ NROW($c4n$)
    $fsT \leftarrow$ F($c1 + \frac{d21+d31}{3}, (n-1)*dt$)
    $L \leftarrow$ REPMAT($dt.*area4.*fsT.*\frac{1}{12}, 3, 1$)
    $b \leftarrow$ ACCUMARRAY($n4e(:), L, [r\ 1]$)
    $Int \leftarrow 0$
    if $n > 2$, then
        $Int \leftarrow dt^2 * A * U(:,1) *$ INTGRAL($(n-1)*dt, dt$)
        for $k \leftarrow 2$ to $(n-2)$, do
            $Int \leftarrow Int + dt^2 * A * U(:,k) *$ INTGRAL($(n-1)*dt, k*dt$)
        end for
    end if
    $b \leftarrow b + B * U(:,n-1) + Int$
    $u \leftarrow$ SPARSE($r, 1$)
    $u_d \leftarrow$ UNIQUE($dir$)
    $u(u_d) \leftarrow$ UD($c4n(u_d,:), (n-1)*dt$)
    $b \leftarrow b - (dt*A + B + dt^2 * A *$ INTGRAL($(n-1)*dt, (n-1)*dt$)$) * u$
    $M_1 \leftarrow dt * A(FN,FN) + B(FN,FN) + dt^2 * A(FN,FN) *$ INTGRAL($(n-1)*dt, (n-1)*dt$)
    $u(FN) \leftarrow$ INV($M_1$) $* b(FN)$
    $solBERR \leftarrow u$
    return $solBERR$
end procedure
```

- **Code for Crank–Nicolson scheme with trapezoidal rule, corresponding to Equation (14):**

Algorithm A6 Crank–Nicolson scheme with the trapezoidal rule.

Main function

```
procedure CNFEMPIDE($c4n, n4e, dir, N, dt$)
    $nE \leftarrow$ NROW($n4e$)
    $nC \leftarrow$ NROW($c4n$)
    $U \leftarrow$ ZEROS($nC, N+1$)
    $FN \leftarrow$ UNIQUE($[1\ 2\ \ldots nC]$)\UNIQUE($dir$)
    $c1 \leftarrow c4n(n4e(:,1),:)$
    $d21 \leftarrow c4n(n4e(:,2),:) - c1$
    $d31 \leftarrow c4n(n4e(:,3),:) - c1$
    $area4 \leftarrow 2 * (d21(:,1).*d31(:,2) - d21(:,2).*d31(:,1))$
    $A \leftarrow$ STIFFASSEMB($n4e, nC, area4, d21, d31, nE$)
    $B \leftarrow$ MASSASSEMB($nC, n4e^T, \frac{1}{4} * area4$)
    $U(:,1) \leftarrow$ U0($c4n$)
    for $j \leftarrow 2$ to $(N+1)$, do
        $U(:,j) \leftarrow$ SOLVECN($c4n, n4e, dir, dt, FN, A, B, n, U, c1, d21, d31, area4$)
    end for
    return $U$
end procedure
```

Auxiliary function

```
procedure SOLVECN($c4n, n4e, dir, dt, FN, A, B, n, U, c1, d21, d31, area4$)
    $r \leftarrow$ NROW($c4n$)
    $fsT_1 \leftarrow$ F($c1 + \frac{d21+d31}{3}, (n-1)*dt$)
    $fsT_2 \leftarrow$ F($c1 + \frac{d21+d31}{3}, (n-2)*dt$)
    $fsT_{mid} \leftarrow 0.5 * (fsT_1 + fsT_2)$
    $L \leftarrow$ REPMAT($dt.*area4.*fsT_{mid}.*\frac{1}{12}, 3, 1$)
    $b \leftarrow$ ACCUMARRAY($n4e(:), L, [r\ 1]$)
    $Int \leftarrow \frac{1}{8} * dt^2 * A * U(:,1) * \big(2 *$ INTGRAL$((n-1.5)*dt, 0)$
                                     $+$INTGRAL$((n-1.5)*dt, (n-1.5)*dt)\big)$
    if $n > 2$, then
        $Int \leftarrow \frac{1}{2} * dt^2 * A * U(:,1) *$ INTGRAL$((n-1.5)*dt, 0)$
                $+\frac{1}{8} * dt^2 * A * U(:,n-1) * \big($INTGRAL$((n-1.5)*dt, (n-1.5)*dt)$
                        $6 *$ INTGRAL$((n-1.5)*dt, (n-2)*dt)\big)$
        for $j \leftarrow 2$ to $(n-2)$, do
            $Int \leftarrow Int + dt^2 * A * U(:,j) *$ INTGRAL$((n-1.5)*dt, (j-1)*dt)$
        end for
    end if
    $b \leftarrow b + (B - \frac{dt}{2} * A) * U(:,n-1) + Int$
    $u \leftarrow$ SPARSE($r, 1$)
    $u_d \leftarrow$ UNIQUE($dir$)
    $u(u_d) \leftarrow$ UD($c4n(u_d,:), (n-1)*dt$)
    $b \leftarrow b - \big(\big(\frac{dt}{2} - \frac{dt^2}{8} *$ INTGRAL$((n-1.5)*dt, (n-1)*dt)\big) * A + B\big) * u$
    $M_1 \leftarrow \big(\frac{dt}{2} - \frac{dt^2}{8} *$ INTGRAL$((n-1.5)*dt, (n-1)*dt)\big) * A(FN, FN) + B(FN, FN)$
    $u(FN) \leftarrow$ INV$(M_1) * b(FN)$
    $solCN \leftarrow u$
    return $solCN$
end procedure
```

Appendix C. Solution Using COMSOL Multiphysics

The fact that (26) is separable in t and s implies that the problem can be reformulated as a hyperbolic initial boundary value problem defined by the so-called telegraph equation [53]. In particular, on using (4) and setting $B\,(t,s) = B_0\,(t)\,B_1\,(s)$, Equation (1) becomes

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - B_0\,(t)\,\phi + f, \tag{A1}$$

where

$$\phi = \int_0^t B_1\,(s) \left\{ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right\} \mathrm{d}s, \tag{A2}$$

which differentiates with respect to t and using (A1) gives

$$\frac{\partial \phi}{\partial t} = B_1\,(t) \left(\frac{\partial u}{\partial t} + B_0\,(t)\,\phi - f \right). \tag{A3}$$

Thus, the system to solve is (A1) and (A3), subject to

$$u(\boldsymbol{x},t) = g(\boldsymbol{x}), \quad (\boldsymbol{x},t) \in \partial\Omega \times [0,T], \tag{A4}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega, \tag{A5}$$

$$\phi(\boldsymbol{x},0) = 0, \quad \boldsymbol{x} \in \Omega. \tag{A6}$$

This is then a relatively straightforward task in COMSOL Multiphysics, which can be carried out from the graphical user interface; in particular, **`PDE General Form`** and **`Weak Form (subdomain)`** modules are used for Equations (A1) and (A3), respectively. Moreover, we are able to ensure a fair comparison with the results of the other sets of code by importing into COMSOL Multiphysics the same meshes as were used for the earlier computations and using the same time step (0.025); in addition, for the time-stepping, we used the generalized-α method [56,57].

On the other hand, if $B\,(t,s)$ is not separable, e.g., $B(t,s) = \exp(-\pi^2(t-s)^2)$ instead of (26), it is not possible to have a formulation that does not retain an integral sign and we have not, at this point, found a simple way to solve this problem, although it would no doubt require coupling COMSOL Multiphysics to a MATLAB script.

References

1. Gurtin, M.E.; Pipkin, A.C. A general theory of heat conduction with finite wave speeds. *Arch. Ration. Mech. Anal.* **1968**, *31*, 113–126. [CrossRef]
2. Joseph, D.D.; Preziosi, L. Heat waves. *Rev. Mod. Phys.* **1989**, *61*, 41–73. [CrossRef]
3. Araújo, A.; Ferreira, J.A.; Oliveira, P. The effect of memory terms in diffusion phenomena. *J. Comput. Math.* **2006**, *24*, 91–102.
4. Habetler, G.J.; Schiffman, R.L. A finite difference method for analyzing the compression of poro-viscoelastic media. *Computing* **1970**, *6*, 342–348. [CrossRef]
5. Pao, C.V. Solution of a nonlinear integrodifferential system arising in nuclear reactor dynamics. *J. Math. Anal. Appl.* **1974**, *48*, 470–492. [CrossRef]
6. Capasso, V. Asymptotic stability for an integro-differential reaction-diffusion system. *J. Math. Anal. Appl.* **1984**, *103*, 575–588. [CrossRef]
7. Barbeiro, S.; Ferreira, J.A. Integro-differential models for percutaneous drug absorption. *Int. J. Comput. Maths* **2007**, *84*, 451–467. [CrossRef]
8. Barbeiro, S.; Ferreira, J.A. Coupled vehicle-skin models for drug release. *Comput. Methods Appl. Mech. Eng.* **2009**, *198*, 2078–2086. [CrossRef]
9. Chuanmiao, C.; Tsimin, S. *Finite Element Methods for Integrodifferential Equations*; World Scientific Publishing Co. Inc.: River Edge, NJ, USA, 1998.

10. Yanik, E.G.; Fairweather, G. Finite element methods for parabolic and hyperbolic partial integro-differential equations. *Nonlinear Anal.* **1988**, *12*, 785–809. [CrossRef]
11. Engler, H. On some parabolic integro-differential equations—Existence and asymptotics of solutions. *Lect. Notes Math.* **1983**, *1017*, 161–167.
12. Reddy, G.M.M.; Sinha, R.K.; Cuminato, J.A. A posteriori error analysis of the Crank-Nicolson finite element method for parabolic integro-differential equations. *J. Sci. Comput.* **2019**, *79*, 414–441. [CrossRef]
13. Sameeh, M.; Elsaid, A. Chebyshev collocation method for parabolic partial integrodifferential equations. *Adv. Math. Phys.* **2016**, *2016*, 7854806. [CrossRef]
14. Filiz, A. Numerical solution of parabolic Volterra integro-differential equations via backward-Euler scheme. *J. Comput. Appl. Math.* **2013**, *3*, 277–282.
15. Soliman, A.F.; El-asyed, A.M.; El-Azab, M.S. On the numerical solution of partial integro-differential equations. *Math. Sci. Lett.* **2012**, *1*, 71–80. [CrossRef]
16. Avazzadeh, Z.; Rizi, Z.B.; Ghaini, F.M.M.; Loghmani, G.B. A numerical solution of nonlinear parabolic-type Volterra partial integro-differential equations using radial basis functions. *Eng. Anal. Bound. Elem.* **2012**, *36*, 881–893. [CrossRef]
17. Ma, J. Finite element methods for partial Volterra integro-differential equations on two-dimensional unbounded spatial domains. *Appl. Math. Comput.* **2007**, *186*, 598–609. [CrossRef]
18. Fakhar-Izadi, F.; Dehghan, M. The spectral methods for parabolic Volterra integro-differential equations. *J. Comput. Appl. Math.* **2011**, *235*, 4032–4046. [CrossRef]
19. Kauthen, J.P. The method of lines for parabolic partial integro-differential equations. *J. Integr. Equa. Appl.* **1992**, *4*, 69–81. [CrossRef]
20. Larsson, S.; Thomee, V.; Wahlbin, L.B. Numerical solution of parabolic integro-differential equations by the discontinuous Galerkin method. *Math. Comput.* **1998**, *67*, 45–71. [CrossRef]
21. Greenwell-Yanik, C.E.; Fairweather, G. Analyses of spline collocation methods for parabolic and hyperbolic problems in 2 space variables. *SIAM J. Numer. Anal.* **1986**, *23*, 282–296. [CrossRef]
22. Lin, Y.P.; Thomée, V.; Wahlbin, L.B. Ritz-Volterra projections to finite-element spaces and applications to integrodifferential and related equations. *SIAM J. Numer. Anal.* **1991**, *28*, 1047–1070. [CrossRef]
23. Pani, A.K.; Sinha, R.K. Error estimates for semidiscrete Galerkin approximation to a time dependent parabolic integro-differential equation with nonsmooth data. *Calcolo* **2000**, *37*, 181–205. [CrossRef]
24. Reddy, G.M.M.; Sinha, R.K. Ritz-Volterra reconstructions and *A Posteriori* Error Anal. Finite Elem. Method Parabol. Integro-Differ. Equations.*IMA J. Numer. Anal.* **2015**, *35*, 341–371. [CrossRef]
25. Reddy, G.M.M.; Sinha, R.K. On the Crank-Nicolson anisotropic a posteriori error analysis for parabolic integro-differential equations. *Math. Comp.* **2016**, *85*, 2365–2390. [CrossRef]
26. Reddy, G.M.M.; Sinha, R.K. The backward Euler anisotropic a posteriori error analysis for parabolic integro-differential equations. *Numer. Methods Partial. Differ. Equ.* **2016**, *32*, 1309–1330. [CrossRef]
27. Shaw, S.; Whiteman, J.R. Numerical solution of linear quasistatic hereditary viscoelasticity problems. *SIAM J. Numer. Anal.* **2000**, *38*, 80–97. [CrossRef]
28. Thomée, V.; Zhang, N.Y. Error estimates for semidiscrete finite element methods for parabolic integro-differential equations. *Math. Comput.* **1989**, *53*, 121–139. [CrossRef]
29. Hecht, F. New development in freefem++. *J. Numer. Math.* **2012**, *20*, 251–265. [CrossRef]
30. deal.II—An Open Source Finite Element Library. Available online: https://www.dealii.org/ (accessed on 21 September 2020).
31. iFEM—Mathematical Software. Available online: http://swmath.org/software/7766 (accessed on 21 September 2020).
32. FEniCS Project. Available online: https://fenicsproject.org/ (accessed on 21 September 2020).
33. DUNE Numerics. Available online: https://www.dune-project.org/ (accessed on 21 September 2020).
34. Physics Simulation Made Easy. Available online: https://www.featool.com/ (accessed on 21 September 2020).
35. MathWorks. Available online: http://www.mathworks.com (accessed on 21 September 2020).
36. COMSOL. Available online: http://www.comsol.com (accessed on 21 September 2020).
37. Acosta, G.; Bersetche, F.M.; Borthagaray, J.P. A short FE implementation for a 2d homogeneous Dirichlet problem of a fractional Laplacian. *Comput. Math. Appl.* **2017**, *74*, 784–816. [CrossRef]

38. Alberty, J.; Carstensen, C.; Funken, S.A.; Klose, R. Matlab implementation of the finite element method in elasticity. *Computing* **2002**, *69*, 239–263. [CrossRef]
39. Abdulle, A.; Nonnenmacher, A. A short and versatile finite element multiscale code for homogenization problems. *Comput. Methods Appl. Mech. Eng.* **2009**, *198*, 2839–2859. [CrossRef]
40. Alberty, J.; Carstensen, C.; Funken, S.A. Remarks around 50 lines of Matlab: Short finite element implementation. *Numer. Algorithms* **1999**, *20*, 117–137. [CrossRef]
41. Bahriawati, C.; Carstensen, C. Three MATLAB implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control. *Comput. Methods Appl. Math.* **2005**, *5*, 333–361. [CrossRef]
42. Carstensen, C.; Klose, R. Elastoviscoplastic finite element analysis in 100 lines of Matlab. *J. Numer. Math.* **2002**, *10*, 157–192. [CrossRef]
43. Cuvelier, F.; Japhet, C.; Scarella, G. An efficient way to assemble finite element matrices in vector languages. *BIT Numer. Math.* **2016**, *56*, 833–864. [CrossRef]
44. Funken, S.; Praetorius, D.; Wissgott, P. Efficient implementation of adaptive P1-FEM in Matlab. *Comput. Methods Appl. Math.* **2011**, *11*, 460–490. [CrossRef]
45. Korzec, M.; Ahnert, T. Time-stepping methods for the simulation of the self-assembly of nano-crystals in Matlab on a GPU. *J. Comput. Phys.* **2013**, *251*, 396–413. [CrossRef]
46. Sutton, O.J. The virtual element method in 50 lines of MATLAB. *Numer. Algorithms* **2017**, *75*, 1141–1159. [CrossRef]
47. Wang, H.; Tian, H. A fast Galerkin method with efficient matrix assembly and storage for a peridynamic model. *J. Comput. Phys.* **2012**, *231*, 7730–7738. [CrossRef]
48. Rahman, T.; Valdman, J. Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements. *Appl. Math. Comput.* **2013**, *219*, 7151 – 7158. [CrossRef]
49. Frank, F.; Reuter, B.; Aizinger, V.; Knabner, P. FESTUNG: A MATLAB/GNU Octave toolbox for the discontinuous Galerkin method, Part I: Diffusion operator. *Comput. Math. Appl.* **2015**, *70*, 11–46. [CrossRef]
50. Kim, S.; Lee, H.C. Finite element method to control the domain singularities of poisson equation using the stress intensity factor: Mixed boundary condition. *Int. J. Numer. Anal. Model.* **2017**, *14*, 500–510.
51. Schiff, B. Finite-element eigenvalues for the Laplacian over an L-shaped domain. *J. Comput. Phys.* **1988**, *76*, 233–242. [CrossRef]
52. Cuvelier, F.; Japhet, C.; Scarella, G. *An Efficient Way to Perform the Assembly of Finite Element Matrices in Matlab and Octave*; Technical Report; INRIA: Rocquencourt, France, 2013.
53. Araújo, A.; Branco, J.R.; Ferreira, J.A. On the stability of a class of splitting methods for integro-differential equations. *Appl. Numer. Math.* **2009**, *59*, 436–453. [CrossRef]
54. Khuri, S.A.; Sayfy, A. A numerical approach for solving an extended Fisher-Kolomogrov-Petrovskii-Piskunov equation. *J. Comp. Appl. Math.* **2010**, *233*, 2081–2089. [CrossRef]
55. Chu, K.T. A direct matrix method for computing analytical Jacobians of discretized nonlinear integro-differential equations. *J. Comput. Phys.* **2009**, *228*, 5526–5538. [CrossRef]
56. Chung, J.; Hulbert, G.M. A time integration algorithm for structural dynamics with improved numerical dissipation—The generalized-α method. *J. Appl. Mech.—Trans. ASME* **1993**, *60*, 371–375. [CrossRef]
57. Jansen, K.E.; Whiting, C.H.; Hulbert, G.M. A generalized-alpha method for integrating the filtered Navier-Stokes equations with a stabilized finite element method. *Comput. Methods Appl. Mech. Eng.* **2000**, *190*, 305–319. [CrossRef]