Computer Physics Communications 185 (2014) 392–397

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

IDSOLVER: A general purpose solver for *n*th-order integro-differential equations☆

Claudio A. Gelmi*, Héctor Jorquera

Department of Chemical and Bioprocess Engineering, Pontificia Universidad Católica de Chile, Av. Vicuña Mackenna 4860, Santiago 7820436, Chile

ARTICLE INFO

Article history:
Received 22 January 2013
Received in revised form
6 September 2013
Accepted 9 September 2013
Available online 23 September 2013

Keywords:
Integro-differential equation
Iterative method
Successive relaxation method
MATLAB

ABSTRACT

Many mathematical models of complex processes may be posed as integro-differential equations (IDE). Many numerical methods have been proposed for solving those equations, but most of them are *ad hoc* thus new equations have to be solved from scratch for translating the IDE into the framework of the specific method chosen. Furthermore, there is a paucity of general-purpose numerical solvers that free the user from additional tasks.

Here we present a general-purpose MATLAB® solver that has the above features. We have chosen to use a numerical quadrature algorithm combined with an accurate and efficient ODE solver – both within a MATLAB® environment – to construct a routine (idsolver) capable of solving a wide variety of IDE of arbitrary order, including the Volterra and Fredholm IDE, variable limits on the integral, and non-linear IDE. The solver performs successive relaxation iterations until convergence is achieved. The user has to define a kernel, limits of integration and a forcing function, then launch the routine and get accurate results by tuning in a single tolerance parameter, as described below for several numerical examples. We have found, by solving several numerical examples from the literature, that the method is robust, fast and accurate.

Program summary

Program title: idsolver
Catalogue identifier: AEQU_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEQU_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: GNU General Public License
No. of lines in distributed program, including test data, etc.: 372
No. of bytes in distributed program, including test data, etc.: 3435
Distribution format: tar.gz
Programming language: MATLAB 2011b.
Computer: PC, Macintosh.
Operating system: Windows, OSX, Linux.
RAM: 1 GB (1,073,741,824 bytes).
Classification: 4.3, 4.11.
Nature of problem:
To solve a wide variety of integro-differential equations (IDE) of arbitrary order, including the Volterra and Fredholm IDE, variable limits on the integral, and non-linear IDE.
Solution method:
An efficient Lobatto quadrature, a robust and accurate IVP MATLAB's solver routine, and a recipe for combining old and new estimates that is equivalent to a successive relaxation method.
Running time:
The solver may take several seconds to execute.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author. Tel.: +56 2 2354 5861; fax: +56 2 2354 5803.
E-mail addresses: cgelmi@ing.puc.cl (C.A. Gelmi), jorquera@ing.puc.cl (H. Jorquera).

0010-4655/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cpc.2013.09.008

1. Introduction

Integro-differential equations (IDE) arise in several research fields, like reactions in continuous mixtures [1], population balances in cell growth [2], neuronal transmission [3], traveling waves [4], charged particle dynamics [5], non-local effects in solid linear elasticity [6], viscoelastic flows [7], the spreading of diseases [8], etc.

Although some of those equations are amenable for closed-form particular solutions [9], almost all of them are numerically solved by using some *ad hoc* method. Therefore, a user faced to a new IDE would have to choose one method from the literature, learn how the method is built up and then apply algebraic manipulations to formulate the method's working equations and solve them to find the sought solution.

Among the published methods for numerically solving IDE, we can cite:

(a) Perturbation methods, that generate a series of successive approximations to the sought solution, like the homotopy method presented by Dehghan and Shakeri [5]; these methods require symbolic manipulation to construct the solution and test convergence by increasing the number of terms in the perturbation expansion—the authors claim that accurate results can be obtained with a few terms, though. In the extension to boundary value IDE proposed by Yildirim [10], further algebraic manipulations are required to put the solution in closed, analytical form.

(b) Methods that use Adomian's decomposition, leading to a series expansion for the sought solution. Deeba et al. [11] – in the context of IDE describing viscoelastic flows or heat transfer in presence of memory effects – have shown that the method has error below 1% with just 2 or 3 terms of the decomposition. However, the method requires symbolic manipulation to construct the approximate solution. In an extension to boundary value IDE, Wazwaz [12] has proposed the use of Padé approximants to bypass the limitation of small radii of convergence for series generated by the Adomian decomposition.

(c) Spectral methods that use a basis of functions to construct an approximation to the solution. Akyüz and Sezer [13] have proposed a Chebyshev collocation method applied to linear Fredholm and Volterra IDE. The resulting equations for the coefficients of the Chebyshev expansion are obtained from a linear system of equations. Nonetheless, algebraic manipulations have to be performed to construct that linear system. Kajani et al. [14] have applied sine and cosine wavelets for linear IDE with good convergence properties; however the user has to choose several parameters in the solution, like the number of subintervals within the original domain. More recently, Driscoll [15] has proposed a software (`chebfun`) that uses an automatic Chebyshev spectral approximation that can handle the different manipulations to construct a solution for linear IDE; for non-linear IDE, an extension of the code is provided as well. A closely related methodology is the application of the Tau method [16–18], which is essentially a spectral method.

(d) The differential transform method technique turns the original IDE problem into a system of algebraic equations. Arikoglu and Ozkol [19] have used this method to solve boundary value IDE, while Darania and Ebadian [20] have used it on linear, higher order Fredholm IDE. The user has to specify the number of grid points within the domain and then solve a recursive set of equations to compute the solution values at those grid points; then the solution for any point in the domain is straightforward by using the transform equations.

(e) Methods that combine ODE solvers with iterative methods for linear systems. An earlier work developed by one of us [21] intended to solve a linear IDE by first applying an efficient IVP solver and then using a residual minimization technique like the Stabilized Bi-Conjugate Gradient iteration (BICGSTAB) proposed by van der Vorst [22]. That iterative solver performed well for several types of linear IDE and it bypassed the use of a matrix representation for the underlying linear operator, thus preventing ill-conditioning issues. This latter point has been remarked by Driscoll [15] in the context of an automatic method of Chebyshev spectral collocation.

Therefore, although several methods are available to solve an IDE, users need to learn their idiosyncrasies before constructing the working equations to solve for their specific IDE—this also involves performing algebraic manipulations along the process. Issues such as convergence, number of terms in the approximate numerical solution, etc. are left to the user to find out, by way of numerical experiments.

Our approach for solving IDE has followed the maxim of trying to formulate the problem in its simplest formulation, and then applying robust numerical techniques – already implemented as automatic routines – that free the user from making decisions upon methodological issues.

In summary, the backbone of our method is: (a) an efficient Lobatto quadrature to handle the integral term in the IDE, (b) a robust and accurate IVP solver such as MATLAB®'s ode113 routine, and (c) a recipe for combining old and new estimates that is equivalent to a successive relaxation method [23] that can be applied to non-linear IDE as well.

Although our methodology was programmed in MATLAB®, it can be easily translated to other scientific languages.

2. Problem description

In this work, we have developed an algorithm for solving the following nth-order integro-differential equation:

$$\begin{aligned} &y^{(n)}(x) = c(x,y) + d(x)\int_{\alpha(x)}^{\beta(x)} k(x,s)F(y(s))ds \quad x \in [a,b] \\ &y(a) = y_0, \qquad y'(a) = y_1, \\ &y''(a) = y_2, \ldots, \qquad y^{(n-1)}(a) = y_{n-1} \end{aligned} \tag{1}$$

Where the exponent n, denotes the order of the derivative of y. The variable integration limits are given by functions that satisfy:

$$a \le \alpha(x) \le \beta(x) \le b \tag{2}$$

Eq. (1) includes as especial cases a linear Volterra integro-differential equation ($\beta(x) = b$) and the Fredholm analog ($\alpha(x) = a$ and $\beta(x) = b$). The function $F(y)$ may be non-linear, although in most applications it is linear.

The method and its computational implementation using the MATLAB® language will be discussed in the following section.

3. Computational method

The first step of the method consists of generating an initial guess for $y(x)$. This is carried out by discarding the integral term in Eq. (1) and solving the resulting ordinary differential equation (ODE) by applying the `ode113` solver (a variable order Adams–Bashforth–Moulton PECE solver) of MATLAB® (www.mathworks.com).

The second step consists of solving Eq. (1) using the initial guess of $y(x)$ on the right hand side and checking if the new solution is

similar or not to that initial guess. The similarity is measured using the following error definition:

$$error = \sum_{k=1}^{M} (y_{i,k} - y_{guess,k})^2 \tag{3}$$

Where M is the total number of discretization points in the domain (by default, $M = 100$), $y_{i,k}$ is the current set of estimated values of $y(x_i)$ in the k-th iteration and $y_{guess,k}$ is the guess of $y(x)$ used to generate the k-th iteration by numerical integration of Eq. (1).

If the solutions are not similar (*i.e.*, the error is higher than the predefined tolerance `Tol`), a new guess y_{k+1} is generated by a linear combination of the current guess of $y(x)$ with the estimate generated from Eq. (1), that is,

$$y_{k+1} = a y_k + (1 - a) y_{guess,k} \tag{4}$$

The smoothing parameter a is equivalent to an under relaxation parameter in the context of iterative methods for linear systems [23], yet we show below that the iteration is robust for non-linear IDE as well. The process continues until the error in Eq. (3) is smaller than the predefined tolerance `Tol`. The final numerical solution is displayed in the Command Window of MATLAB® and saved in the file `ansidsolver.txt`.

In order to solve the integral component of Eq. (1), the method integrates the argument using the solver `quadl`, which uses an adaptive Lobbato quadrature. Because $y(x)$ is required in the argument of the integral, at each integration step, the method linearly interpolates (`interp1`) the guess of $y(x)$ for the specific values required by `quadl`.

The error control tolerances (*i.e.*, relative and absolute error tolerances) of the MATLAB® solvers `ode113` and `quadl` can be modified from their default values (*i.e.*, `RelTol` $= 1 \cdot 10^{-3}$, `AbsTol` $= 1 \cdot 10^{-6}$) by selecting a flag value in the call to `idsolver`. In the next section, we will illustrate the use of `idsolver` and the effect of changing those parameters to non-default values (*i.e.*, `RelTol` $=$ `AbsTol` $= 1 \cdot 10^{-8}$).

4. Illustrative examples

In this section, we illustrate the use of our solver `idsolver` and its performance solving different problems taken from the literature. All examples were solved using a dual-core 1.86 GHz PC running MATLAB® under Windows XP.

Example 1. The first-order, linear integro-differential equation with constant coefficients [24]:

$$\begin{aligned} y'(x) &= y(x) - 0.5x + \frac{1}{1+x} - \ln(1+x) \\ &\quad + \frac{1}{(\ln 2)^2} \int_0^1 \frac{x}{1+s} y(s)ds \quad x \in [0, 1] \\ y(0) &= 0 \end{aligned} \tag{5}$$

has the exact solution $y(x) = \ln(1 + x)$. In order to solve this example, Jaradat et al. [24] applied the Homotopy Analysis Method (HAM), a general analytic approach, which requires the construction of an homotopy-series solution. In contrast, we can directly solve Eq. (5) by calling our solver `idsolver`, as shown below:

```
xinterval = [0 1]; % Interval of the independent
variable x
n = 1; % Order of the IDE
InitCond = 0; % Initial condition
c = @(x,y) y-0.5*x+1/(1+x)-log(1+x);
d = @(x) 1/log(2)^2;
k = @(x,s) x./(1+s);
alpha = @(x) 0; % Lower limit of the integral
beta = @(x) 1; % Upper limit of the integral
Tol = 1e-8; % Tolerance
Flag = 0; % Setting MATLAB's default tolerances
idsolver(xinterval,n,InitCond,c,d,k,alpha,
beta,Tol,Flag)
```

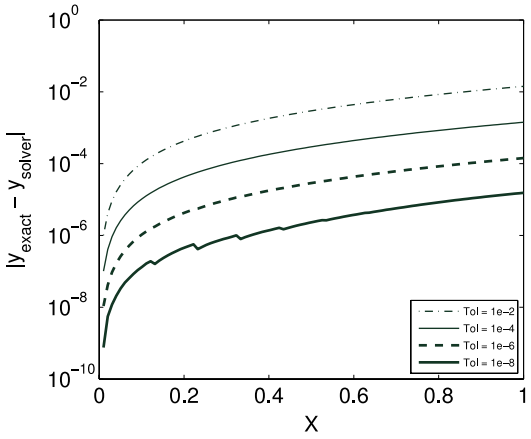

Fig. 1. Numerical performance of `idsolver` when solving Example 1 for different `Tol` values using MATLAB's default error tolerances.

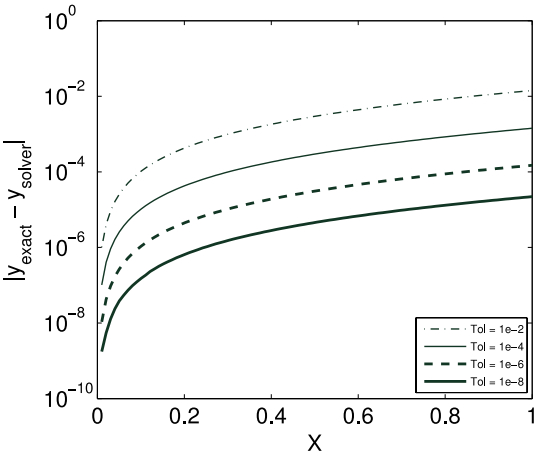

Fig. 2. Numerical performance of `idsolver` when solving Example 1 for different `Tol` values, when all error tolerances were set to $1 \cdot 10^{-8}$ in `ode113` and `quadl`.

In this case, the solver required 22 iterations and 1.2 s of computations. Fig. 1 shows the absolute error between the exact solution and the one computed by `idsolver` for different tolerances (`Tol`) using the default error control values of MATLAB®. As expected, the error decreased as the tolerance `Tol` decreased. Given that the analytical solution of Eq. (5) is a monotonically increasing function, the absolute difference between the numerical and exact solution follows the same trend. Fig. 2 shows the computed error when all default `ode113` and `quadl` tolerances (*i.e.*, `AbsTol` and `RelTol`) were changed to a value of $1 \cdot 10^{-8}$ (*i.e.*, `Flag = 1`). A closer look at Figs. 1 and 2, shows that stringent error controls did not improve the numerical solution of Eq. (5), especially at $x = 1$. Nonetheless, the errors are smoother on [0, 1] when all tolerances are set to a common, low value such as $1 \cdot 10^{-8}$.

Example 2. The first-order integro-differential equation with constant coefficients [24]:

$$\begin{aligned} y'(x) &= y(x) - \cos(2\pi x) - 2\pi \sin(2\pi x) \\ &\quad - \frac{1}{2}\sin(4\pi x) + \int_0^1 \sin(4\pi x + 2\pi s) y(s)ds \\ y(0) &= 1 \quad x \in [0, 1] \end{aligned} \tag{6}$$

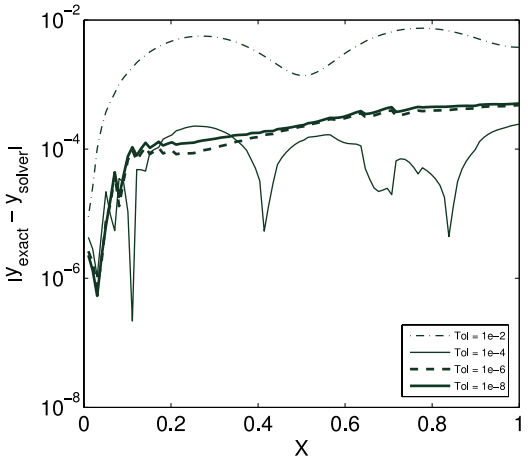

Fig. 3. Numerical performance of `idsolver` when solving Example 2 for different `Tol` values using MATLAB's default error tolerances.

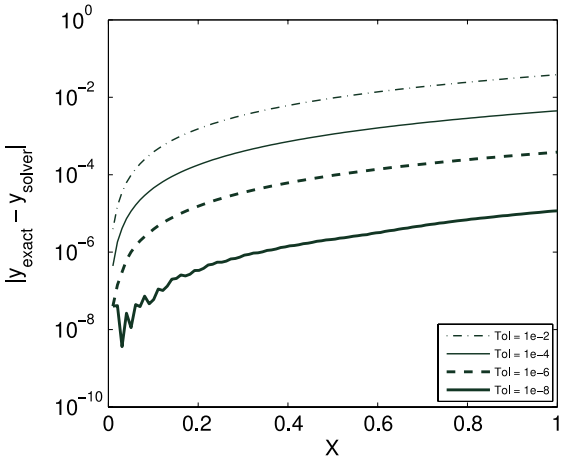

Fig. 5. Numerical performance of `idsolver` when solving Example 3 for different `Tol` values using MATLAB's default error tolerances.

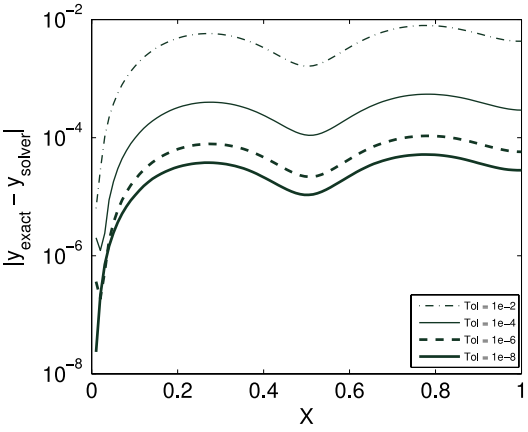

Fig. 4. Numerical performance of `idsolver` when solving Example 2 for different `Tol` values, when all error tolerances were set to $1 \cdot 10^{-8}$ in `ode113` and `quadl`.

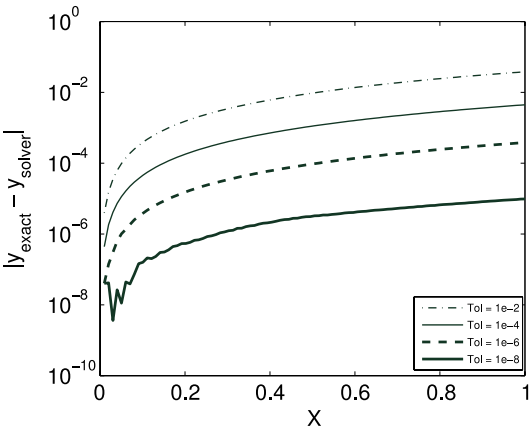

Fig. 6. Numerical performance of `idsolver` when solving Example 3 for different `Tol` values, when all error tolerances were set to $1 \cdot 10^{-8}$ in `ode113` and `quadl`.

has the exact solution $y(x) = \cos(2\pi x)$. As in Example 1, Jaradat et al. [24] have used the HAM method to solve the above equation. In our case, we have solved (6) by calling the solver `idsolver` as shown in the Appendix. In the current example, when `Tol` $= 1 \cdot 10^{-8}$, our solver required 14 iterations and 2.7 s of computations to achieve convergence. Fig. 3 shows the absolute error between the exact solution and the one computed by `idsolver` using MATLAB®'s default error control properties. In this case, the numerical solution of Eq. (6) was not improved when the `idsolver` tolerance `Tol` was decreased below 10^{-6}. In this example the global error in the numerical solution is controlled by the default errors in the quadrature and ODE solver. In fact, when we set all MATLAB® tolerances (*i.e.*, `AbsTol` and `RelTol`) to a value of $1 \cdot 10^{-8}$ (*i.e.*, `Flag = 1`), the numerical solution of Eq. (6) indeed improves monotonously when we decrease the `Tol` value, see Fig. 4.

Example 3. The first order non-linear integro-differential equation with constant coefficients (modified from example 1 in [20]):

$$y'(x) = 1 - \frac{29}{60}x + \int_0^1 (xs)y^2(s)ds \quad x \in [0, 1]$$
$$y(0) = 1 \tag{7}$$

has the exact solution $y(x) = (1+x+x^2)$. A simple code modification was necessary to perform in the `idsolver` routine in order to handle the non-linear integral term $y^2(s)$. The code used to invoke the `idsolver` is shown in the Appendix. In this example, when `Tol` $= 1 \cdot 10^{-8}$, the modified solver required 47 iterations and 2.0 s of computations to reach convergence. When we set stringent MATLAB® tolerances (*i.e.*, `Flag = 1`) in this example, we do not get an improvement in the numerical solution of Eq. (7) (compare Figs. 5 and 6). In this non-linear example, the local error has a monotonous increase in local errors as x increases above 0.2. The magnitude of those local errors may be modified by the user-defined tolerance `Tol`, as can be seen in Figs. 5 and 6.

Example 4. The first order integro-differential equation with variable limits [21]:

$$y'(x) = x(1+x^{1/2})e^{-x^{1/2}} - (x^2+x+1)e^{-x} + \int_x^{\sqrt{x}} (xs)y(s)ds \quad x \in [0, 1]$$
$$y(0) = 1 \tag{8}$$

has the exact solution $y(x) = e^{-x}$. When we choose `Tol` $= 1 \cdot 10^{-8}$, `idsolver` requires 12 iterations and 0.5 s of computations to decrease the error below `Tol`. A comparison between Figs. 7 and 8 shows that decreasing the default MATLAB® tolerances (*i.e.*, `Flag = 1`) clearly improves the numerical solution of $y(x)$, particularly between $x = 0$ and $x = 0.2$. Thus decreasing the

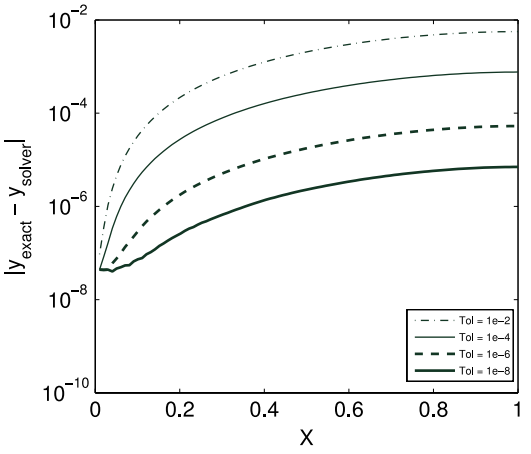

Fig. 7. Numerical performance of `idsolver` when solving Example 4 for different `Tol` values using MATLAB's default error tolerances.

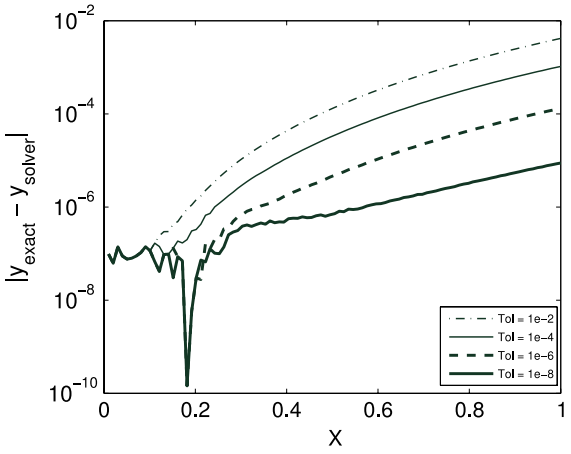

Fig. 9. Numerical performance of `idsolver` when solving Example 5 for different `Tol` values using MATLAB's default error tolerances.

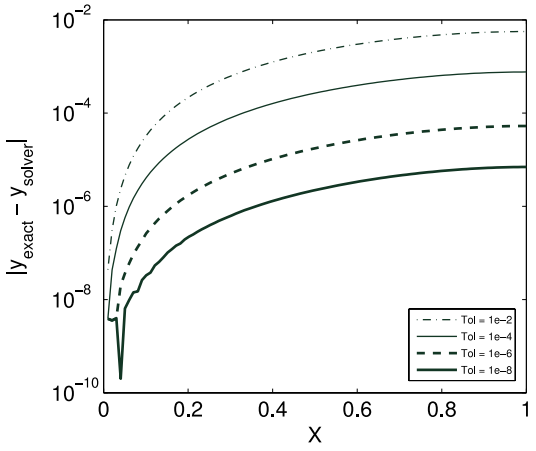

Fig. 8. Numerical performance of `idsolver` when solving Example 4 for different `Tol` values, when all error tolerances were set to $1 \cdot 10^{-8}$ in `ode113` and `quadl`.

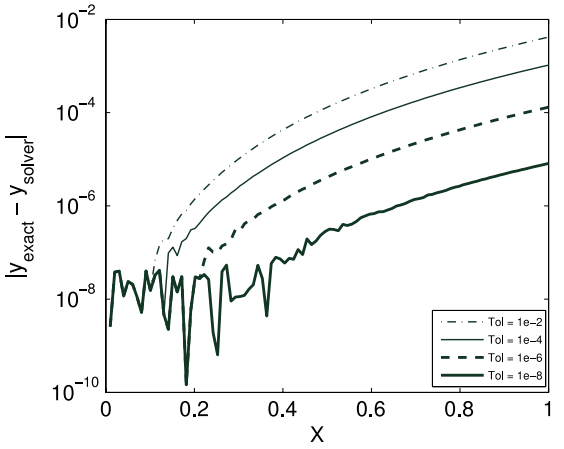

Fig. 10. Numerical performance of `idsolver` when solving Example 5 for different `Tol` values, when all error tolerances were set to $1 \cdot 10^{-8}$ in `ode113` and `quadl`.

`idsolver` tolerance (`Tol`) significantly increases the accuracy of the numerical solution, reaching in the worst case, a local error of $1 \cdot 10^{-5}$.

Example 5. The fourth order integro-differential equation with constant coefficients (adapted from example 3 in [20]):

$$y^{(4)}(x) = e^x - x + \int_0^1 (xs)y(s)ds \quad x \in [0, 1]$$
$$y(0) = 1, \qquad y^{(1)}(0) = 1, \qquad y^{(2)}(0) = 1, \qquad y^{(3)}(0) = 1 \tag{9}$$

has the exact solution $y(x) = e^x$.

In this example, our solver required only 10 iterations and 0.7 s of computations to attain convergence. The code used to solve this example is shown in the Appendix. Figs. 9 (`Flag = 0`) and 10 (`Flag = 1`) show that solution accuracy improved after setting MATLAB® tolerances stringently; both figures show that as `Tol` values decreased the accuracy of the numerical solution improved.

5. Conclusions

Our user-oriented IDE solver has a robust behavior for a wide range of integro-differential equations with short computation times and exhibiting a good accuracy when using a `Tol` value of $1 \cdot 10^{-8}$. Convergence of the numerical solution was monotonous when all the routine's tolerance parameters were alike and they were decreased altogether.

We do not recommend using default MATLAB® error tolerances and strongly suggest users plotting numerical results for decreasing values of `Tol` until no further improvements are observed. Our solver did not run into stability problems for the ODE solver chosen, yet for stiff ODE we could include stiff IVP solvers in `idsolver` to accommodate those dynamic features.

In some examples tested in the present work, we have found an irregular shape of the local error as a function of x. This behavior depends upon error propagation features that are problem-dependent – thus we cannot predict this outcome from the numerical method – sometimes it happens, sometimes it does not. Nonetheless, numerical errors can be monotonously reduced by choosing a single tolerance parameter when calling the solver.

Our future efforts will be devoted to extending our technique to systems of integro-differential equations and integro-differential equations with boundary value conditions.

Appendix

The following code was used to solve Example 2:

```
xinterval = [0 1];
n = 1;
InitCond = 1;
```

```
c = @(x,y) y-cos(2*pi*x)-2*pi*sin(2*pi*x)
-0.5*sin(4*pi*x);
d = @(x) 1;
k = @(x,s) sin(4*pi*x+2*pi*s);
alpha = @(x) 0;
beta = @(x) 1;
Tol = 1e-8;
Flag = 0; % Setting MATLAB's default tolerances
idsolver(xinterval,n,InitCond,c,d,k,alpha,
beta,Tol,Flag)
```

The following code was used to solve Example 3:

```
xinterval = [0 1];
n = 1;
InitCond = 1;
c = @(x,y) 1-29*x/60;
d = @(x) 1;
k = @(x,s) x.*s;
alpha = @(x) 0;
beta = @(x) 1;
Tol = 1e-8;
Flag = 0; % Setting MATLAB's default tolerances
idsolverEx3(xinterval,n,InitCond,c,d,k,alpha,
beta,Tol,Flag)
```

The following code was used to solve Example 4:

```
xinterval = [0 1];
n = 1;
InitCond = 1;
c = @(x,y) x*(1+x^0.5)*exp(-x^0.5)-(x^2+x+1)*exp(-x);
d = @(x) 1;
k = @(x,s) x.*s;
alpha = @(x) x;
beta = @(x) x 0.5;
Tol = 1e-8;
Flag = 0; % Setting MATLAB's default tolerances
idsolver(xinterval,n,InitCond,c,d,k,alpha,
beta,Tol,Flag)
```

Finally, the following code was used to solve Example 5:

```
xinterval = [0 1];
n = 4;
InitCond = [1 1 1 1];
c = @(x,y) exp(x)-x;
d = @(x) 1;
k = @(x,s) x.*s;
alpha = @(x) 0;
beta = @(x) 1;
Tol = 1e-8;
Flag = 0; % Setting MATLAB's default tolerances
idsolver(xinterval,n,InitCond,c,d,k,alpha,
beta,Tol,Flag)
```

References

[1] G. Astarita, R. Ocone, AIChE J. 34 (1988) 1299–1309.
[2] N.V. Mantzaris, J.J. Liou, P. Daoutidis, F. Srienc, J. Biotech. 71 (1999) 157–174.
[3] Z. Jackiewicz, M. Rahman, B.D. Welfert, Appl. Math. Comput. 195 (2008) 523–536.
[4] J.P. Boyd, J. Comput. Phys. 189 (2003) 98–110.
[5] M. Dehghan, F. Shakeri, Prog. Electromagn. Res. 78 (2008) 361–376.
[6] E. Emmrich, O. Weckner, Math. Mech. Solids 12 (2007) 363–384.
[7] A.S. Lodge, M. Renardy, J.A. Nohel, Viscoelasticity and Rheology, Academic Press, Orlando, Fla, 1985, University of Wisconsin–Madison, Mathematics Research Center.
[8] J. Medlock, M. Kot, Math. Biosci. 184 (2003) 201–222.
[9] G. Astarita, AIChE J. 35 (1989) 529–532.
[10] A. Yildirim, Comput. Math. Appl. 56 (2008) 3175–3180.
[11] E. Deeba, S.A. Khuri, S.S. Xie, Appl. Math. Comput. 115 (2000) 123–131.
[12] A.M. Wazwaz, Appl. Math. Comput. 118 (2001) 327–342.
[13] A. Akyüz, M. Sezer, Int. J. Comput. Math. 72 (1999) 491–507.
[14] M.T. Kajani, M. Ghasemi, E. Babolian, Appl. Math. Comput. 180 (2006) 569–574.
[15] T.A. Driscoll, J. Comput. Phys. 229 (2010) 5980–5998.
[16] S.M. Hosseini, S. Shahmorad, Appl. Math. Model. 27 (2003) 145–154.
[17] S.M. Hosseini, S. Shahmorad, Appl. Math. Comput. 136 (2003) 559–570.
[18] J. Pour-Mahmoud, M.Y. Rahimi-Ardabili, S. Shahmorad, Appl. Math. Comput. 168 (2005) 465–478.
[19] A. Arikoglu, I. Ozkol, Appl. Math. Comput. 168 (2005) 1145–1158.
[20] P. Darania, A. Ebadian, Appl. Math. Comput. 188 (2007) 657–668.
[21] H. Jorquera, Comput. Phys. Comm. 86 (1995) 91–96.
[22] H.A. van der Vorst, SIAM J. Sci. Stat. Comput. 13 (1992) 631–644.
[23] O. Axelsson, Iterative Solution Methods, Cambridge University Press, Cambridge, England; New York, 1994.
[24] H. Jaradat, O. Alsayyed, S. Al-Shara, J. Math. Stat. 4 (2008) 250–254.