Svetlana Matculevich

Guaranteed error bounds for linear algebra problems and a class of Picard–Lindelöf iteration methods

Master's Thesis
in Information Technology
February 13, 2012

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Svetlana Matculevich
Contact information: svmatsul@student.jyu.fi
Title: Luotettavat virherajat lineaarisille yhtälöryhmille ja tietyn tyypin Picard-Lindelöf -iteraatiomenetelmille
Työn nimi: Guaranteed error bounds for linear algebra problems and a class of Picard–Lindelöf iteration methods
Project: Master's Thesis in Information Technology
Supervisors: Pekka Neittaanmäki, Sergey Repin
Page count: 82
Abstract: This study focuses on iteration methods based on the Banach fixed point theorem and a posteriori error estimates of Ostrowski. Their application for systems of linear simultaneous equations, bounded linear operators, as well as integral and differential equations is considered. The study presents a new version of the Picard–Lindelöf method for ordinary differential equations (ODEs) supplied with guaranteed and explicitly computable upper bounds of the approximation error. The estimates derived in the thesis take into account interpolation and integration errors and, therefore, provide objective information on the accuracy of computed approximations.
Suomenkielinen tiivistelmä: Tässä tutkimuksessa tarkastellaan Banachin kiintopistelauseeseen perustuvia iteraatiomenetelmiä ja Ostrowskin a posteriori virhe-estimaatteja. Tutkimuksessa käsitellään virhe-estimaattien sovelluksia lineaarisen yhtälöryhmän, lineaaristen operaattoreiden sekä integraali- ja differentiaaliyhtälöiden tapauksissa. Tutkimuksessa luodaan eksplisiittisesti laskettavat virherajat Picard-Lindelöf -iteraatiomenetelmään differentiaaliyhtälöiden ratkaisemiseksi. Virherajat huomioivat sekä interpolaatiovirheen että numeerisesta integroinnista johtuvat virheen ja tarjoavat luotettavaa tietoa iteraation tuottaman approksimaation tarkkuudesta.
Keywords: Error estimates, iterative method, reliability, the Picard–Lindelöf method, guaranteed bounds
Avainsanat: Virhe-estimaatit, iteraatiomenetelmä, luotettavuus, Picard-Lindelöf -menetelmä, luotettavat rajat

Preface

Reliable error control of the evolutionary models is one of the most challenging and unexplored areas in applied mathematics. The Picard–Lindelöf method suggests one possible way of such an analysis. The goal of this thesis is to combine the Picard–Lindelöf method with Ostrowski's a posteriori estimates and derive a fully reliable guaranteed numerical solution for a class of ordinary differential equations. This idea has been realized and tested in numerical examples that confirm the theoretical results.

The obtained results exposed in the thesis were presented in two international conferences Reliable Methods of Mathematical Modeling on July $6-8$, 2011, in Lausanne, EPFL, Switzerland (RMMM 2011), and the 4th Workshop on Advanced Numerical Methods for Partial Differential Equation Analysis on August $22-24$, 2011 in The Euler International Mathematical Institute, St. Petersburg, Russia.

I would like to thank Professor Sergey Repin for the supervision and support in my scientific work. I also want to thank the dean of the department Professor Pekka Neittaanmäki for guidance and genuine interest in my thesis topic. Finally, I would like to thank my colleagues Olli Mali, Immanuel Anjam, and Marjaana Nokka from my research group for interesting work and fruitful discussions.

Lastly, I would like to thank the University of Jyväskylä for the support.

Mathematical notations

$\|\cdot\|$	L^2-norm
$\|\cdot\|_X$	L^2-norm in space X
$\{x_i\}_{i=1}^{\infty}$	infinite sequence
$\{x_i\}_{i=1}^{N}$	finite sequence
$\mathcal{A} : X \to Y$	mapping from space X to space Y
(X, d)	space with metric d
$\mathbb{R}^n$	n-dimensional Euclidean space
$\mathbb{M}_{n\times n}$	set of matrices of size $m \times n$
$\lambda_{min}(A)$	minimal value of eigenvalues of matrix A
$\lambda_{max}(A)$	maximal value of eigenvalues of matrix A
Ω	domain in $\mathbb{R}^n$ space
$(\cdot, \cdot)$	scalar multiplication
$\frac{d\cdot}{dt}$	derivative with respect to variable t
$\lvert\cdot\rvert$	module, absolute value
π	projection operator
CP^1	set of piecewise linear affine functions
$f'(t)\rvert_t$	derivative with respect to variable t

Glossary

estimate:	a calculated approximation of a result which is usable even if input data is incomplete or uncertain
reliability:	the ability of a mathematical object to perform and maintain its functions for every problem statement with any conditions
iterative methods:	a mathematical procedure that generates a sequence of improving approximate solutions for a class of problems
a posteriori:	afterwards
a priori:	previously
FDM:	a finite difference method
majorant:	an error upper bound in agreed norm
minorant:	an error lower bound in agreed norm
ODE:	an ordinary differential equation
PDE:	a partial differential equation
$A > 0$:	$(Ax, x) > 0 \ \forall\, x \in X, \ x \neq 0$, where $x \in H$ (finite-dimensional vector space)
$A \leq B$:	$(Ax, x) \leq (Bx, x)$

Definition. *Function f is continuous at the point t_0 if the following holds: for $\forall\ \varepsilon > 0$, however small, there exists some $\delta > 0$, such that for $|t - t_0| < \delta$ the value of $f(t)$ satisfies inequality $|f(t_0) - f(t)| < \varepsilon$.*

Definition. *Function f is Lipschitz continuous if there exists a real constant $\mathrm{K} \geq 0$ such that, for any x_1 and x_2 in X, the following holds:*

$$\|f(x_1) - f(x_2)\|_X \leq \mathrm{K}\|x_1 - x_2\|_X. \tag{1}$$

Constant K is called a Lipschitz constant for the function f.

Definition. *Bounded operator $\mathcal{T} : X \to Y$ is called contractive if its operator norm requires inequality $\|\mathcal{T}\| \leq 1$.*

Definition. *A piecewise linear function is a piecewise-defined function whose pieces are linear.*

Theorem (The Picard–Lindelöf theorem [2]). *Consider the initial value problem*

$$u'(t) = \varphi(u(t), t), u(t_0) = u_0, \quad t = [t_0 - \varepsilon, t_0 + \varepsilon], \tag{2}$$

where φ is Lipschitz continuous in u and continuous in t. Then, for $\varepsilon > 0$, there exists a unique solution $u(t)$ of (2) within a time interval $[t_0 - \varepsilon, t_0 + \varepsilon]$.

Theorem (The Banach theorem). *Let (X, d) be a non-empty complete metric space. Let $\mathcal{T} : X \to X$ be a q - contraction mapping on X, $q < 1$, such that $\|\mathcal{T}x - \mathcal{T}y\|_X \leq q\|x - y\|_X$ for $\forall\ x, y \in X$. Then, the $\mathcal{T}$ admits unique fixed point $x_\odot \in X$, i.e. $\mathcal{T}x_\odot = x_\odot$.*

Contents

1 Introduction

Iteration methods are widely used in numerical analysis. Convergence and a priori estimates, which follow from the Banach fixed point theorem, are well-studied.

This thesis is focused on guaranteed a posteriori error estimates for iteration schemes that arise in

(a) numerical linear algebra, and

(b) integration methods for ODEs based on the Picard–Lindelöf iteration method.

The guaranteed reliability of numerical solutions is also a very important question. If the error of a numerical solution can be reliably estimated, then it is possible to detect areas where errors are excessively high and modify (adapt) the grid (mesh) in order to minimize them.

In Chapter 2, the general iteration lgorithms is discussed. By analyzing the main results of the contractivity theory, we arrive at the a priori and a posteriori error estimate.

In Chapter 3, we continue discussion on the application of the obtained error bounds to the simplest linear algebra problems, such as iteration methods for systems of linear equations and the bounded linear operator theory. We combine the classified iteration schemes with Ostrowski's a posteriori error bounds and obtain the reliable iteration algorithms for solving linear algebra problems.

In Chapter 4, the main idea of the Picard–Lindelöf method is presented. It is justified with conditions which not only provide the convergence of the method but also allow the application of a posteriori error estimates. We discuss the application of the Ostrowski estimate to the problem formulated in this Chapter. However, these estimates cannot be directly used. In practical computations based on the Picard–Lindelöf method, we must take interpolation and integration errors into account. This analysis is done in Section 4.2 of Chapter 4. It leads to error bounds, which include interpolation and integration errors, which are derived in Section 4.4. The structure of the algorithm is exposed in Section 4.6, where the results of the numerical tests are presented.

2 Error bounds for iteration methods

Iteration methods are widely used in numerical analysis. They were developed because of specific features of linear systems of equations. Since systems of equations are mathematical tools used to describe essential parts of mathematical problems, it is important to develop reliable iteration methods to solve them. Therefore, in the current chapter the methods to control the accuracy of a sequence of approximate the solutions constructed by the iteration process are discussed.

2.1 General iteration algorithm

The general form to present the majority of mathematical problems is the following:

$$x_{\odot} = \mathcal{T} x_{\odot} + g, \tag{2.1}$$

where $\mathcal{T} : X \to X$ is a certain bounded operator, X is a Banach space, and $g \in X$. The function $x_{\odot}$ satisfying (2.1) is called the fixed point of the operator $\mathcal{T}$. Numerically (2.1) is approximated by

$$x_i = \mathcal{T} x_{i-1} + g, \quad \text{where } i = 1, \dots\,. \tag{2.2}$$

The iteration process (2.2) is formalized in Algorithm 1.

Algorithm 1 General iteration algorithm.

```
**Input:** $x_0 \in X$ {initial approximation}, $\varepsilon$ {required accuracy of approximation}
$i = 1$ {initial iteration step}
$x_i = \mathcal{T} x_{i-1} + g$ {$1^{st}$ element of sequence}
**while** $\|x_i - x_{i-1}\| > \varepsilon$ **do**
    $i = i + 1$
    $x_i = \mathcal{T} x_{i-1} + g$
**end while**
**Output:** $x_i$ {fixed point approximation}
```

2.1.1 Contractive operators

Reliable iteration algorithms can be constructed if the operator $\mathcal{T}$ possesses additional properties. We assume that $\mathcal{T} : S \rightarrow S$, where S is a closed nonempty set in X, and $\mathcal{T}$ is q-contractive, i.e.

$$\|\mathcal{T}x - \mathcal{T}y\|_X \leq q\|x - y\|_X, \forall x, y \in X, \quad 0 < q < 1, \tag{2.3}$$

where x and y are independent from each other. According to the Banach theorem [6], from the properties listed above follows that the sequence $\{x_i\}_{i=1}^{\infty}$ converges to a unique fixed point $x_\odot$. By using (2.3), we can obtain

$$\|x_{i+1} - x_i\|_X = \|\mathcal{T}x_i - \mathcal{T}x_{i-1}\|_X \leq q\|x_i - x_{i-1}\|_X \ \leq ... \leq q^i\|x_1 - x_0\|_X. \tag{2.4}$$

The Banach theorem also provides the inequality

$$\begin{aligned}\|x_{i+m} - x_i\|_X \leq & \|x_{i+m} - x_{i+m-1}\|_X + ... + \|x_{i+1} - x_i\|_X \leq \\ & \leq q^i(q^{m-1} + q^{m-2} + ... + 1)\|x_1 - x_0\|_X \leq \frac{q^i}{1-q}\|x_1 - x_0\|_X,\end{aligned} \tag{2.5}$$

where $m > 1$.

2.1.2 Error control by an a priori estimate

A reliable algorithm must include a stopping criterion based on the true values of the error, which is

$$e_i := \|x_i - x_\odot\|_X. \tag{2.6}$$

Taking (2.3) into account, we have

$$e_i = \|\mathcal{T}x_{i-1} - \mathcal{T}x_\odot\|_X \leq qe_{i-1} \leq q^i e_0. \tag{2.7}$$

From (2.7) we can conclude that the error tends to zero with the parameter q. However, the estimate (2.7) has only theoretical (asymptotical) meaning because e_0 is unknown. To get a computable bound, in (2.5) we fix i, tend $m \in \mathbb{N}$ to infinity, and obtain a new inequality

$$e_i \leq \frac{q^i}{1-q}\|x_1 - x_0\|_X := M_i^0. \tag{2.8}$$

Now Algorithm 1 can be represented with the error upper bounds included.

Algorithm 2 General iteration algorithm with a priori error upper bound

```
Input: $x_0 \in X$ {initial approximation}, $\varepsilon$ {required accuracy of approximation}
$i = 1$ {initial iteration step}
$x_i = \mathcal{T} x_{i-1} + g$ {$1^{st}$ element of sequence}
$M_i^0 = \frac{q}{1-q}\|x_i - x_{i-1}\|_X$ {upper bound for error $e_1$}
while $M_i^0 > \varepsilon$ do
  $i = i + 1$
  $x_i = \mathcal{T} x_{i-1} + g$
  $M_i^0 = q M_i^0$
end while
Output: $x_i$ {fixed point approximation}
        $M_i^0$ {upper bound of approximation error}
```

2.1.3 Error control by a posteriori estimates

We can obtain another upper bound for the error (2.6). We take the inequality (2.5), set $i = 1$ and obtain

$$\|x_{1+m} - x_1\|_X \leq \frac{q}{1-q}\|x_1 - x_0\|_X, \quad \text{where} \quad m \in \mathbb{N}. \tag{2.9}$$

Assuming that $x_{1+m} \to x_\odot$ as $m \to +\infty$, we get

$$\|x_\odot - x_1\|_X \leq \frac{q}{1-q}\|x_1 - x_0\|_X. \tag{2.10}$$

As soon as any element can be considered as the initial one, we take x_{i-1} as the 0-th term of the sequence and imply the relation (2.10)

$$e_i \leq \frac{q}{1-q}\|x_i - x_{i-1}\|_X. \tag{2.11}$$

A lower error bound follows from the triangle inequality

$$\|x_i - x_{i+1}\|_X \leq \|x_i - x_\odot\|_X + \|x_{i+1} - x_\odot\|_X \leq (1+q)\|x_i - x_\odot\|_X. \tag{2.12}$$

We obtain

$$e_i \geq \frac{1}{1+q}\|x_i - x_{i+1}\|_X. \tag{2.13}$$

After analyzing (2.13) and (2.11) we conclude that two-sided bounds of the derivation between x_i and $x_\odot$ can be computed by using three neighboring elements x_{i-1}, x_i, and x_{i+1}. The error estimates (2.11) and (2.13) have been derived by Ostrowski [13].

Henceforth, we call respectful upper and lower bounds "error majorant" and "error minorant" and denote them $M_i^{\oplus}$ and $M_i^{\ominus}$.

Let $\mathcal{T}$ be a q-contractive operator. Then, the following estimate holds:

$$\begin{aligned} e_i &\leq \frac{q}{1-q}\|x_i - x_{i-1}\|_X := M_i^{\oplus} \\ e_i &\geq \frac{1}{1+q}\|x_i - x_{i+1}\|_X := M_i^{\ominus} \end{aligned} \tag{2.14}$$

With the estimates (2.14) Algorithm 1 is changed the following way.

Algorithm 3 General iteration algorithm with a posteriori error control based on (2.14).

```
**Input:** $x_0 \in X$ {initial approximation}, $\varepsilon$ {required accuracy of approximation}
$i = 1$    {initial iteration step}
$x_i = \mathcal{T}x_{i-1} + g$    {$1^{st}$ element of sequence}
$M_i^{\oplus} = \frac{q}{1-q}\|x_i - x_{i-1}\|_X$    {upper bound for error $e_1$}
**while** $M_i^{\oplus} > \varepsilon$ **do**
  $i = i + 1$
  $x_i = \mathcal{T}x_{i-1} + g$
  $M_i^{\oplus} = \frac{q}{1-q}\|x_i - x_{i-1}\|_X$
  $M_i^{\ominus} = \frac{1}{1+q}\|x_i - x_{i+1}\|_X$
**end while**
**Output:** $x_i$    {fixed point approximation}
        $M_i^{\oplus}$ and $M_i^{\ominus}$    {upper and lower bound of approximation error}
```

In some cases it is easier to prove that a sequence of approximate solutions is obtained by a contractive mapping if we consider the operator

$$\hat{\mathcal{T}} = T^n := \underbrace{TT...T}_{n \text{ times}} \tag{2.15}$$

instead of T. The proposition 2.1.1 shows that we arrive at similar results.

Proposition 2.1.1. *Let $T : S \to S$ be a continuous mapping such that $\hat{\mathcal{T}}$, defined by (2.15) is a q-contractive mapping, $q \in (0, 1)$. Then,*

$$x = Tx \qquad \textit{and} \qquad \hat{x} = \hat{\mathcal{T}}\hat{x} \tag{2.16}$$

have the same fixed point, which is unique, i.e. $x = \hat{x}$, and can be found by the iteration procedure.

2.1.4 Advanced forms of a posteriori error bounds

The ratio of the upper and lower error bounds (2.14) exceeds $\frac{1+q}{1-q}$. If q is close to 1, the efficiency of the estimates deteriorates. However, there are advanced forms of error bounds that can be derived by using additional terms of the sequence $\{x_i\}_{i=1}^N$. For example,

$$\begin{aligned}\|x_i - x_\odot\|_X \leq \|x_i - x_{i+1}\|_X + \|x_{i+1} - x_\odot\|_X \leq \\ \leq \|x_i - x_{i+1}\|_X + \frac{q}{1-q}\|x_i - x_{i+1}\|_X, \quad (2.17)\end{aligned}$$

which leads to the upper bound

$$\|x_i - x_\odot\|_X \leq \frac{1}{1-q}\|x_i - x_{i+1}\|_X := M_i^{\oplus,1}(x_i, x_{i+1}). \quad (2.18)$$

Since

$$\frac{1}{1-q}\|x_i - x_{i+1}\|_X \leq \frac{q}{1-q}\|x_{i-1} - x_i\|_X, \quad (2.19)$$

then $M_i^{\oplus,1}(x_i, x_{i+1})$ is sharper than $M_i^{\oplus}$. The same idea can be applied to subsequences of $\{x_i\}_{i=1}^N$, so we obtain various upper bounds for $\|x_i - x_\odot\|_X$:

$$\|x_i - x_\odot\|_X \leq \frac{1}{1-q^l}\|x_i - x_{i+l}\|_X := M_i^{\oplus,l}(x_i, x_{i+l}), \quad l = 1, ..., L. \quad (2.20)$$

By using three sequential elements x_i, x_{i+1}, and x_{i+2}, we deduce another estimate

$$\begin{aligned}\|x_i - x_\odot\|_X &\leq \|x_i - x_{i+1}\|_X + \|x_{i+1} - x_{i+2}\|_X + \|x_{i+2} - x_\odot\|_X \leq \\ &\leq \|x_i - x_{i+1}\|_X + \|x_{i+1} - x_{i+2}\|_X + \frac{q}{1-q}\|x_{i+2} - x_{i+1}\|_X \leq \\ &\leq \|x_i - x_{i+1}\|_X + \frac{1}{1-q}\|x_{i+2} - x_{i+1}\|_X := M_i^{\oplus,1,2}(x_i, x_{i+1}, x_{i+2}). \quad (2.21)\end{aligned}$$

Error minorants can be improved by similar arguments. We have

$$\|x_i - x_{i+l}\|_X \leq \|x_i - x_\odot\|_X + \|x_{i+l} - x_\odot\|_X \leq (1 + q^l)\|x_i - x_\odot\|_X. \quad (2.22)$$

Therefore,

$$\|x_i - x_\odot\|_X \geq \frac{1}{1+q^l}\|x_i - x_{i+l}\|_X := M_i^{\ominus,l}(x_i, x_{i+l}). \quad (2.23)$$

The ratio of advanced upper and lower bounds in (2.20) and (2.23) is $\frac{1+q^l}{1-q^l}$. If q is close to 1, they provide much better estimates than (2.14).

3 Iteration methods for bounded linear operators

The general form of error estimates can be rewritten for problems with a bounded linear operator.

3.1 Iteration methods generated by bounded linear operators

Consider a bounded linear operator $\mathcal{L} : X \to X$, where X is a Banach space, and given $b \in X$. The iteration process is defined by the following relation:

$$x_j = \mathcal{L}x_{j-1} + b. \tag{3.1}$$

Let $x_\odot$ be a fixed point of (3.1) and

$$\|\mathcal{L}\| = q < 1. \tag{3.2}$$

By using the Banach theorem, it is easy to show that $x_j \to x_\odot$ as $j \to \infty$. Indeed, let $\bar{x}_j = x_j - x_\odot$. Then,

$$\bar{x}_j = \mathcal{L}x_{j-1} + b - x_\odot = \mathcal{L}(x_{j-1} - x_\odot) = \mathcal{L}\bar{x}_{j-1}. \tag{3.3}$$

Since $0_X = \mathcal{L}0_X$, we note that 0_X is a unique fixed point of $\mathcal{L}$. Therefore,

$$\|x_j - x_\odot\|_X = \|\bar{x}_j - 0_X\|_X \leq \frac{q^j}{1-q}\|\bar{x}_1 - \bar{x}_0\|_X = \frac{q^j}{1-q}\|R(x_0)\|_X := \mathcal{M}_j^0 \tag{3.4}$$

and

$$\|x_j - x_\odot\|_X \leq \frac{q}{1-q}\|R(x_{j-1})\|_X := \mathcal{M}_j^\oplus, \tag{3.5}$$

where $R(z) = \mathcal{L}z + b - z$. The lower bound of the error can be found analogously:

$$\|x_j - x_\odot\|_X \geq \frac{1}{1+q}\|x_{j+1} - x_j\|_X = \frac{1}{1+q}\|R(x_j)\|_X := \mathcal{M}_j^\ominus. \tag{3.6}$$

The estimates (3.5) and (3.6) are analogous to the error bounds (2.8), (2.11) and (2.13) obtained in the previous chapter. They are used for problems approximated by finite differences which are reduced to the system of the linear equations with a specific matrix (this application is discussed in Example 3.8).

Since the obtained results are applicable to problems approximated by systems of linear simultaneous equations, we consider different types of iteration schemes including guaranteed error bounds resulting from (2.8) and (2.11).

Consider the operator $\mathcal{L}$ in the set $X = \mathbb{R}^n$, which is defined by a non-degenerate matrix $A \in \mathbb{M}_{n\times n}$ decomposed as

$$A = L + D + R, \tag{3.7}$$

where L, R, and D are certain left, right triangular and diagonal matrixes, respectively. So the system of equations $Ax = f$ can be rewritten by L, D, and R matrices combination, for example, the Jacobi scheme

$$x^{k+1} = D^{-1}(L + R)x^k + D^{-1}f, \tag{3.8}$$

the Gauss–Seidel scheme

$$x^{k+1} = (L + D)^{-1}Rx^k + (L + D)^{-1}f \tag{3.9}$$

or a generalized form of (3.9), which is called the SOR scheme

$$(D + \omega L)\frac{x_{k+1} - x_k}{\omega} + Ax_k = f, \tag{3.10}$$

where $0 < \omega < 2$.

The listed methods are called one-step iteration methods. Otherwise, if x^{k+1} depends on more than one previous step, i.e. $x^{k+1} = F(x^k, x^{k-1}, ...)$, then a method is considered a multi-step iteration method. In general, a canonical form of a one-step iteration method is presented by

$$B^{k+1}\frac{x^{k+1} - x^k}{\tau_{k+1}} + Ax^k \ = f \ , k = 1, ..., n. \tag{3.11}$$

We discuss the applicability of the estimates (3.4) and (3.5) to several iteration schemes.

3.1.1 Stationary method

Consider stationary methods ($B_{k+1} = B$ and $\tau_{k+1} = \tau$)

$$B\frac{x_{k+1} - x_k}{\tau} + Ax_k \ = f. \tag{3.12}$$

Let the deviation between an approximate solution from $\{x_k\}_{k=1,...}$ and the exact solution be denoted as $\bar{x}_k = x_k - x_\odot, k = 1, ...$ Then from (3.12), the relation

$$B\frac{\bar{x}_k - \bar{x}_{k-1}}{\tau} + A\bar{x}_{k-1} = 0 \tag{3.13}$$

follows. It can be rewritten as

$$\bar{x}_k = (I - \tau B^{-1}A)\bar{x}_{k-1}. \tag{3.14}$$

The iteration process converges if and only if transformation matrix $S = I - \tau B^{-1}A$ satisfies the conditions of Theorem 3.1.1.

Theorem 3.1.1. *The iteration scheme (3.12) converges for any initial element if and only if all eigenvalues of $S = I - \tau B^{-1}A$ belong to the interval $(-1, 1)$.*

Now we can formulate a priori and a posteriori estimates. First, recall the following result.

Theorem 3.1.2. *Let A and B be symmetric matrices, positive definite matrices such that*

$$\mu_{min}B \leq A \leq \mu_{max}B,\ \mu_{min}, \mu_{max} > 0. \tag{3.15}$$

Then,

$$\|x_k - x_\odot\|_{A,B} \leq q^k \|x_0 - x_\odot\|_{A,B}, \tag{3.16}$$

where $q = \frac{1-\kappa}{1+\kappa}$, $\kappa = \frac{\mu_{min}}{\mu_{max}}$ and $\tau = \frac{2}{\mu_{min}+\mu_{max}}$ with norm $\|x\|_D := \sqrt{(Dx, x)},\ x \in H$.

Consider x_{k-1} as the starting element, then

$$\|\bar{x}_k\|_{A,B} = \|x_k - x_\odot\|_{A,B} \leq q\|x_{k-1} - x_\odot\|_{A,B}. \tag{3.17}$$

By applying (2.14) to (3.17), we obtain a posteriori and a priori bounds

$$\|\bar{x}_k\|_{A,B} \leq \frac{q}{1-q}\|\bar{x}_k - \bar{x}_{k-1}\|_{A,B} = \frac{q}{1-q}\|x_k - x_{k-1}\|_{A,B} \tag{3.18}$$

and

$$\|x_k - x_\odot\|_{A,B} \leq \frac{q^k}{1-q}\|x_1 - x_0\|_{A,B}. \tag{3.19}$$

In general, the reliable canonical one-step stationary iteration scheme with guaranteed bounds can be represented by Algorithm 4. [1]

[1]Algorithm 4 is applied below in Example 3.5 of the corresponding section.

Algorithm 4 General stationary iteration algorithm to solve system $Ax = f$ including guaranteed error bounds.

```
**Input:** $x_0 \in X$ {initial approximation}, $\varepsilon$ {required accuracy for approximation}, $N$ {system size}, $A$ {system matrix}, $f$ {system right part}, $B$ {constructed symmetric matrix}, $\lambda_{min}(B^{-1}A)$ {estimate for minimal eigenvalues of $B^{-1}A$}, $\lambda_{max}(B^{-1}A)$ {estimate for maximal eigenvalues of $B^{-1}A$}
$\tau = \frac{2}{\lambda_{max}(B^{-1}A)+\lambda_{min}(B^{-1}A)}$
$\kappa = \frac{\lambda_{min}(B^{-1}A)}{\lambda_{max}(B^{-1}A)}$
$q = \frac{1-\kappa}{1+\kappa}$
$L = I - \tau B^{-1}A$   {matrix of contractive operator}
$b = \tau B^{-1}f$   {shift vector}
$k = 1$   {initial iteration step}
$x_k = Lx_{k-1} + b$   {$1^{st}$ element of sequence}
$\mathcal{M}_k^{\oplus} = \frac{q}{1-q}\|x_k - x_{k-1}\|_{A,B}$   {a posteriori estimate for $\|\bar{x}_1\|_{A,B}$}
**while** $\mathcal{M}_k^{\oplus} > \varepsilon$ **do**
  $k = k + 1$
  $x_k = Lx_{k-1} + b$   {new element of the sequence}
  $\mathcal{M}_k^{0} = \frac{q^k}{1-q}\|x_1 - x_0\|_{A,B}$ {a priori estimate for $\|\bar{x}_k\|_{A,B}$}
  $\mathcal{M}_k^{\oplus} = \frac{q}{1-q}\|x_k - x_{k-1}\|_{A,B}$ {a posteriori estimate for $\|\bar{x}_k\|_{A,B}$}
**end while**
**Output:** $x_k$ {approximate solution}
            $\mathcal{M}_k^0$, $\mathcal{M}_k^{\oplus}$ {a priori and a posteriori error bounds}
```

3.1.2 The Richardson method

Consider the Richardson iteration scheme

$$\frac{x_{k+1} - x_k}{\tau_{k+1}} + Ax_k = f. \tag{3.20}$$

For (3.20) Theorem 3.1.2 implies the following corollary.

Corollary 3.1.3. *If $A = A^T > 0$ and $\tau = \frac{2}{\lambda_{min}(A)+\lambda_{max}(A)}$, the following inequality*

$$\|x_k - x_\odot\|_A \leq q^k \|x_0 - x_\odot\|_A, \tag{3.21}$$

holds, where $q = \frac{1-\kappa}{1+\kappa}$, $\kappa = \frac{\lambda_{min}}{\lambda_{max}}$.

Analogously, we apply the Ostrowski upper estimate with $q = \frac{1-\kappa}{1+\kappa}$ to obtain the a posteriori estimate

$$\|x_k - x_\odot\|_A \le \frac{q}{1-q}\|x_k - x_{k-1}\|_A. \tag{3.22}$$

The test of (3.22) is illustrated in the Example 3.6.

3.1.3 The Chebyshev method

We proceed to the schemes with variable step:

$$\frac{x_{k+1} - x_k}{\tau_{k+1}} + Ax_k = f. \tag{3.23}$$

In this method τ_k is selected in order to minimize the error $\bar{x}_k = x_k - x_\odot$.

Theorem 3.1.4. *Let A be a symmetric positive definite matrix with eigenvalues in $[\lambda_{min}, \lambda_{max}]$ and $\kappa = \frac{\lambda_{min}}{\lambda_{max}}$. Let n be a defined number of sub-iterations. Among iteration schemes (3.23) the minimal error on the n-th step is attained with the step*

$$\tau_k = \frac{\tau_0}{1 + \rho_0 t_k}, \quad k = 1, ..., n, \tag{3.24}$$

where

$$t_k = \cos\frac{(2k-1)\pi}{2n}, \quad \rho_0 = \frac{1-\kappa}{1+\kappa} \quad \text{and} \quad \tau_0 = \frac{2}{\lambda_{min} + \lambda_{max}}. \tag{3.25}$$

In this case,

$$\|x_n - x_\odot\| \le q_n \|x_0 - x_\odot\|, \tag{3.26}$$

where

$$q_n = \frac{2\rho_1^n}{1 + \rho_1^{2n}}, \qquad \rho_1 = \frac{1 - \sqrt{\kappa}}{1 + \sqrt{\kappa}}. \tag{3.27}$$

Remark 3.2. In the Chebyshev method we construct the next element with the help of a generated set of n parameters τ_k, where $k = 1, ..., n$:

$$L_n = (I - \tau_n A)(I - \tau_{n-1})...(I - \tau_2 A)(I - \tau_1 A). \tag{3.28}$$

Corollary 3.2.1. *In a particular case, if $n = 2$, so that we have x_0, x_1, and x_2. Then,*

$$\|\bar{x}_2\| \le \frac{q_2}{1-q_2}\|\bar{x}_2 - \bar{x}_0\| \le \frac{q_2}{1-q_2}\|x_2 - x_0\|. \tag{3.29}$$

Algorithm 5 Chebyshev iteration algorithm to solve system $Ax = f$ including guaranteed error bounds.

```
Input: $x_0 \in X$ {initial approximation}, $\varepsilon$ {required accuracy for approximation}, $N$ {system size}, $A$ {system matrix}, $f$ {system right part}, $\lambda_{min}(A)$ {estimate for minimal eigenvalue of $A$}, $\lambda_{max}(A)$ {estimate for maximal eigen value of $A$}, $n$ {accessorial set size, subcycle size}
$\tau_0 = \frac{2}{\lambda_{max}(A)+\lambda_{min}(A)}$
$\kappa = \frac{\lambda_{min}(A)}{\lambda_{max}(A)}$
$\rho_0 = \frac{1-\kappa}{1+\kappa}$
$\rho_1 = \frac{1-\sqrt{\kappa}}{1+\sqrt{\kappa}}$
$q(n) = \frac{2\rho_1^n}{1+\rho_1^{2n}}$
$x_1 = \texttt{chebyshev\_subcycle}(x_0, n, A, f, \tau_0, \rho_0)$
$i = 2$    {initial iteration step}
$\mathcal{M}_i^{\oplus} = \frac{q}{1-q}\|x_i - x_{i-1}\|_A$ {a posteriori estimate}
while  $\mathcal{M}_i^{\oplus} > \varepsilon$ do
    $x_i = \texttt{chebyshev\_subcycle}\ (x_{i-1}, n, \tau_0, \rho_0)$    {new element of the sequence}
    $\mathcal{M}_i^0 = \frac{q^i}{1-q}\|x_1 - x_0\|_A$ {a priori estimate}
    $\mathcal{M}_i^{\oplus} = \frac{q}{1-q}\|x_i - x_{i-1}\|_A$ {a posteriori estimate}
    $i = i + 1$
end while
Output: $x_i$ {approximate solution}
        $\mathcal{M}_k^0$, $\mathcal{M}_k^{\oplus}$ {a priori and a posteriori error bounds}
```

Algorithm 6 Chebyshev subcycle algorithm to construct new element of the sequence.

```
Input: $x_0$ {input vector}, $n$ {size of accessorial parameters set}, $A$ {system matrix}, $f$ {system right part}, $\tau_0$, $\rho_0$
$L_n = I$    {unitary matrix}
$b_n = \bar{0}$    {zero vector}
for $k = 1 : n$ do
    $t_k = \cos\frac{(2k-1)\pi}{2n}$
    $\tau_k = \frac{\tau_0}{1+\rho_0 t_k}$
    $L_n = (I - \tau_k A)L_n$
    $b_n = (I - \tau_k A)b_n + \tau_k f$
end for
$x = L_n x_0 + b_n$
Output: $x$ {output vector}
```

We describe the Chebyshev iteration scheme in Algorithm 5. The subprocedure `chebyshev_cycle()` is described in Algorithm 6. [2]

Remark 3.3. For any $A > 0$, we can obtain the following error estimate

$$\|x - \tilde{x}\| \leq \frac{1}{\lambda_{min}} \|A\tilde{x} - f\| := \mathcal{M}^{\lambda_{min}}, \tag{3.30}$$

which is the special case of the general estimate that is discussed in Example 3.10.

There is another well known estimate for the relative error based on the condition number of A:

$$\frac{\|x - \tilde{x}\|}{\|\tilde{x}\|} \leq cond(A) \frac{\|r\|}{\|f\|} := \mathcal{M}^{cond}, \tag{3.31}$$

where $r = A\tilde{x} - f$.

From [1] we can have the estimates based basically on the system residual

$$M^{lower} := \frac{\|r\|^2}{\|A^T r\|} \leq \|x - \tilde{x}\| \leq C(A) \frac{\|r\|^2}{\|A^T r\|} := M^{upper}, \tag{3.32}$$

where the constant is given as

$$C(A) = \sup_{\|y\|=1} \|A^T y\| \|A^{-1} y\|.$$

3.4 Examples

Example 3.5. In this example, we discuss the general stationary iteration method (3.12) with guaranteed upper bounds described by Algorithm 4. We consider the concrete system $Ax = f$. The matrices A and B are constructed by the MATLAB program listed in Listing A.1.

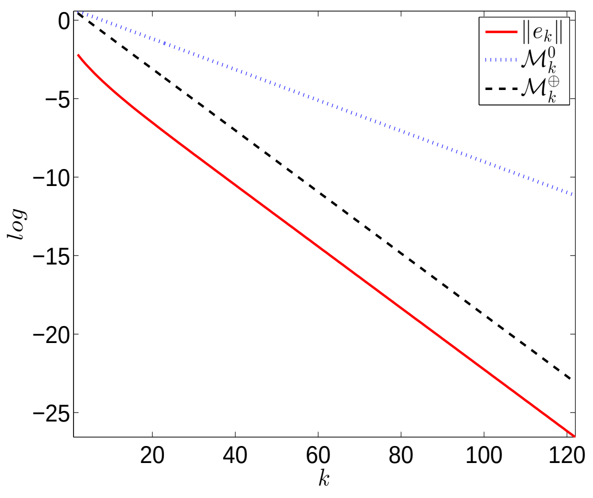
In numerical tests, size of systems is taken as $N = 10, 100, 1000$ and $\varepsilon = 10^{-8}$.

In Figure 3.1, we compare an a priori error estimate (3.19) and an a posteriori estimate (3.18) for $N = 10, 100, 1000$. It is easy to observe that for all N the difference between a priori and a posteriori estimates is significant.

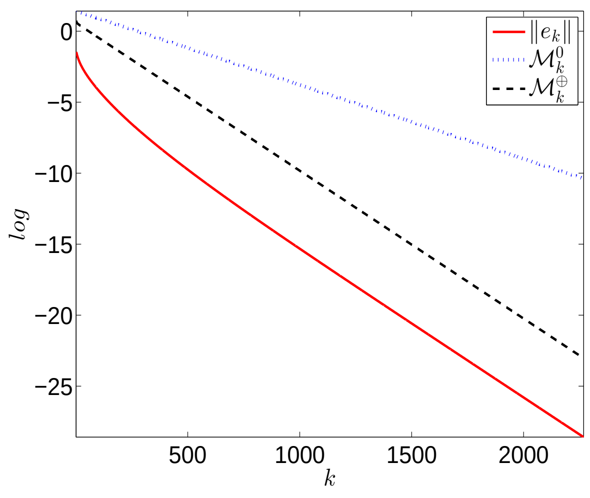
Example 3.6. We test the Richardson scheme (3.20) the same system. In this case, $N = 10, 100, 1000$ and condition numbers are $cond(A) = 10, 100, 1000$, respectively, $\varepsilon = 10^{-7}$.

The upper error bounds are shown in Figure 3.2. As in Example 3.5, the a posteriori bounds are sharper than the a priori bounds.

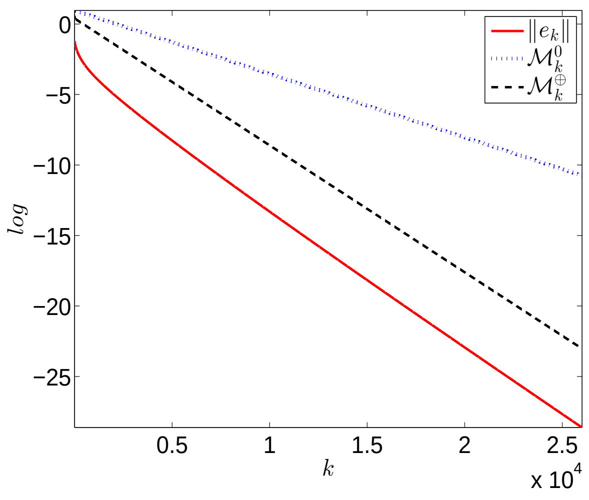
[2] The application of Algorithm 5 is discussed in Example 3.7.



(a) N = 10



(b) N = 100

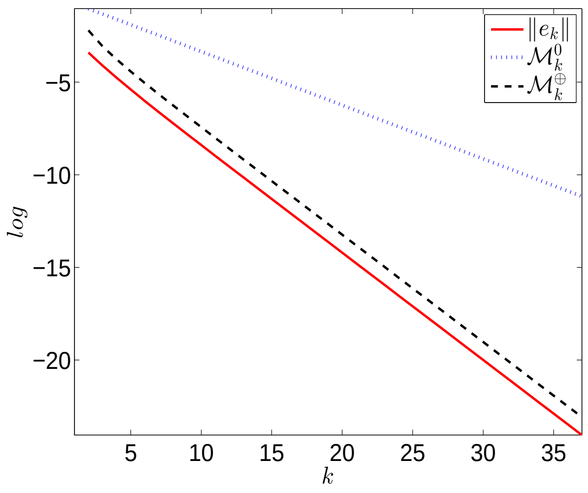


(c) N = 1000

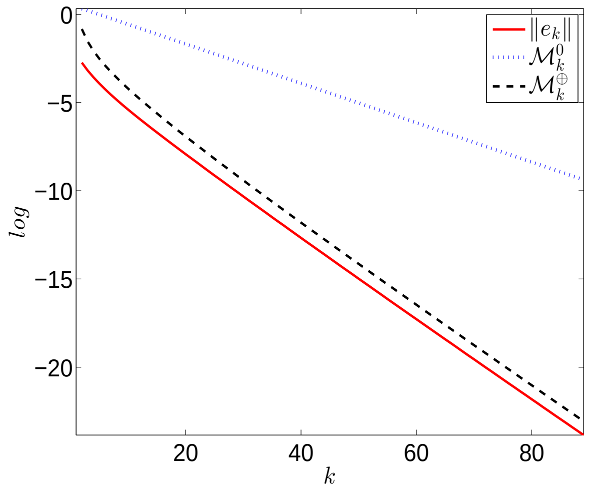
Figure 3.1: The a priori estimate $\mathcal{M}_k^0 = \frac{q^k}{1-q}\|x_0 - x_1\|_{A,B}$, the a posteriori estimate $\mathcal{M}_k^\oplus = \frac{q}{1-q}\|x_k - x_{k-1}\|_{A,B}$, and the true error $\|e_k\| = \|x_k - x_\odot\|_{A,B}$, where $k = 1, \dots$.

Example 3.7. In the third example we consider the same problem $Ax = f$. Systems with sizes $N = 10^k$, $k = 2, 3$ are tested. The set of accessorial parameters for each system (the size of the Chebyshev subcycle) is set by the formula $n = 2k + 1$, where $k = 2, 3$, respectively. The condition number for all matrices is equal to 10.

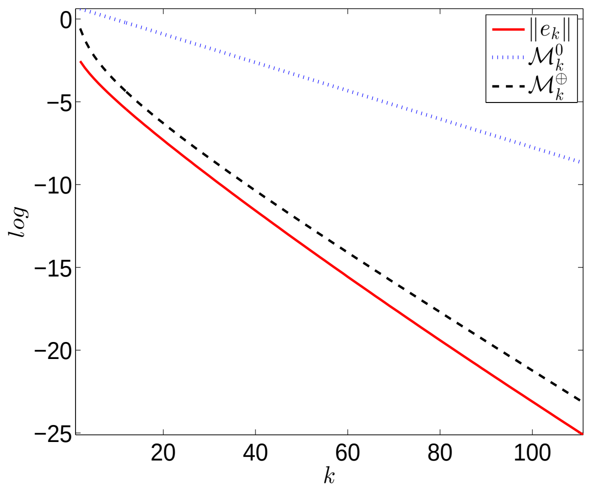
In Figures 3.3 and 3.4 it is easy to see that both estimates reproduce errors quite realistically. However, comparison of the efficiency indexes shows that a posteriori upper bounds are more accurate than a priori estimates.



(a) N = 10



(b) N = 100



(c) N = 1000

Figure 3.2: The a priori estimate $\mathcal{M}_k^0 = \frac{q^k}{1-q}\|x_0 - x_1\|_A$, the a posteriori estimate $\mathcal{M}_k^\oplus = \frac{q}{1-q}\|x_k - x_{k-1}\|_A$, and the true error $\|e_k\| = \|x_k - x_\odot\|_A$.

Example 3.8. In this example, the application of iteration schemes for a system of linear equations, which arise in finite-difference approximations of the problems, is discussed. We consider the following Dirichlet problem:

$$\begin{gathered} -a(x,y)\frac{\partial^2 u}{\partial x^2} - b(x,y)\frac{\partial^2 u}{\partial y^2} + c(x,y)u = f(x,y) \text{ in } \Omega = \ [x_0, x_M] \times [y_0, y_N] \\ u = \phi(x,y) \text{ on } \Gamma \end{gathered} \tag{3.33}$$

We approximate (3.33) by a second-order scheme with the so-called "cross" pattern:

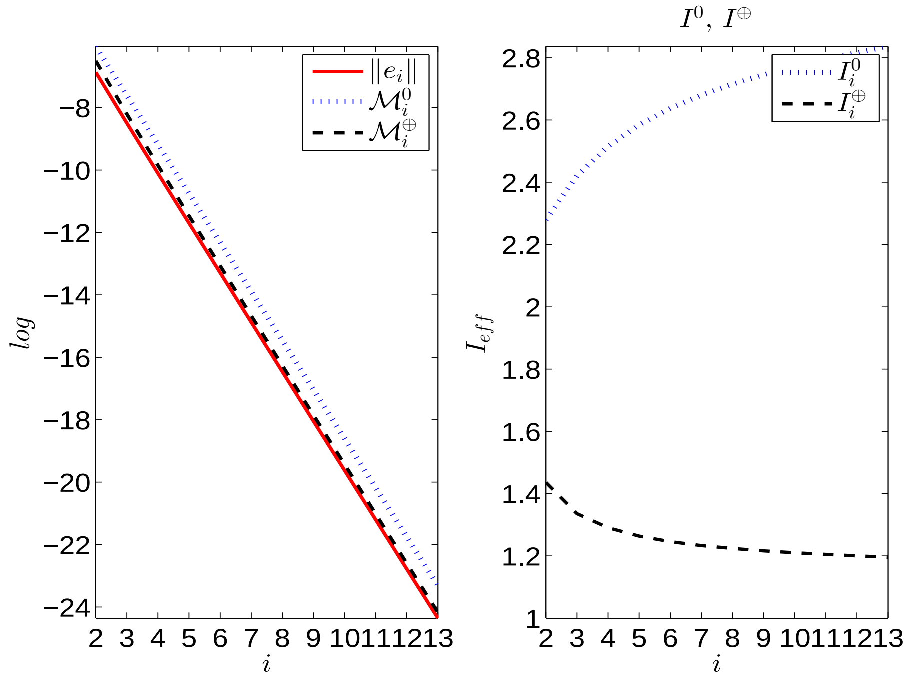


Figure 3.3: The a priori estimate $\mathcal{M}_i^0$, the a posteriori estimate $\mathcal{M}_i^\oplus$, the error $\|e_i\|$ and efficiency indexes for both estimates, $n = 100$.

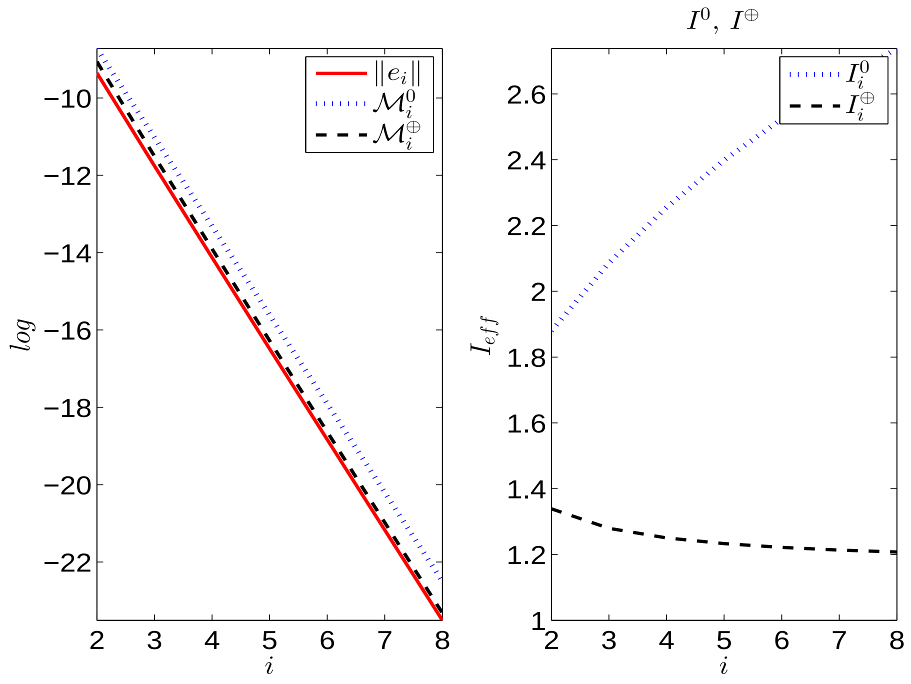


Figure 3.4: The a priori estimate $\mathcal{M}_i^0$, the a posteriori estimate $\mathcal{M}_i^\oplus$, the error $\|e_i\|$ and efficiency indexes for both estimates, $n = 1000$.

$$-a_{i,j}\frac{v_{i-1,j}-2v_{i,j}+v_{i+1,j}}{{h_x}^2}-b_{i,j}\frac{v_{i,j-1}-2v_{i,j}+v_{i,j+1}}{{h_y}^2}+c_{i,j}v_{i,j}=f_{i,j}. \tag{3.34}$$

We rewrite (3.34) for the internal nodes of the grid:

$$c_{i,j}v_{i-1,j}+e_{i,j}v_{i,j-1}+b_{i,j}v_{i,j}+d_{i,j}v_{i,j+1}+a_{i,j}v_{i+1,j}=g_{i,j}. \tag{3.35}$$

After proper renumbering (see the example in Figure 3.5 related to a simple 5×5 mesh), we obtain a system of linear equations with a block the tridiagonal matrix .

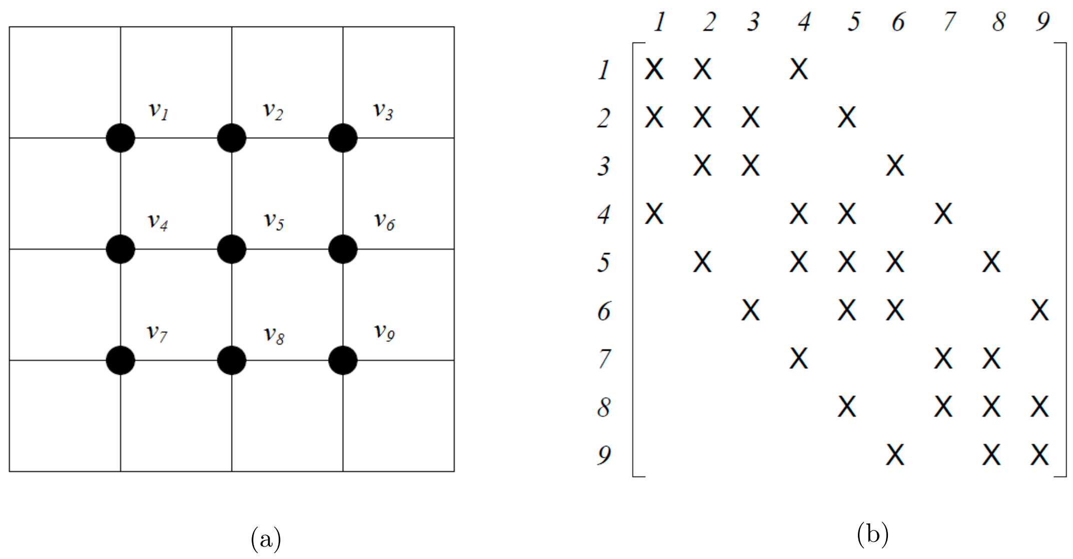


Figure 3.5: An example of renumbering on the mesh $[M\times N], M=5, N=5$.

We construct a test problem

$$\begin{aligned}-(x^2+y+1)\frac{\partial^2 u}{\partial x^2}-(x+y+2)\frac{\partial^2 u}{\partial y^2}+(x+y+4)u=f(x,y)\\ \Omega=[0.2,3.3]\times[0.5,4.2]\\ u=\phi(x,y) \text{ on } \Gamma\end{aligned} \tag{3.36}$$

with

$$\begin{aligned}f=-2(x^2+y+1)(y^2-4y)-2(x+y+2)(x^2-3x)+\\ +(x+y+4)(x^2-3x)(y^2-4y),\end{aligned}$$

and with the exact solution $u(x,y) = (x^2 - 3x)(y^2 - 4y)$. We use a regular mesh $[M \times N], M = 18, N = 20$ and solve the corresponding problem (associated with the matrix of size $n = 288$ and the condition number $cond(A) = 131.7$). The required accuracy is $\varepsilon = 10^{-10}$.

By using the SOR method (with $B = \frac{1}{\omega}D + L$), we obtain the transformation matrix S, where the parameter $q = 0.97712$.

In Figure 3.6 the error estimates (3.4), (3.5), and (3.6) are illustrated.

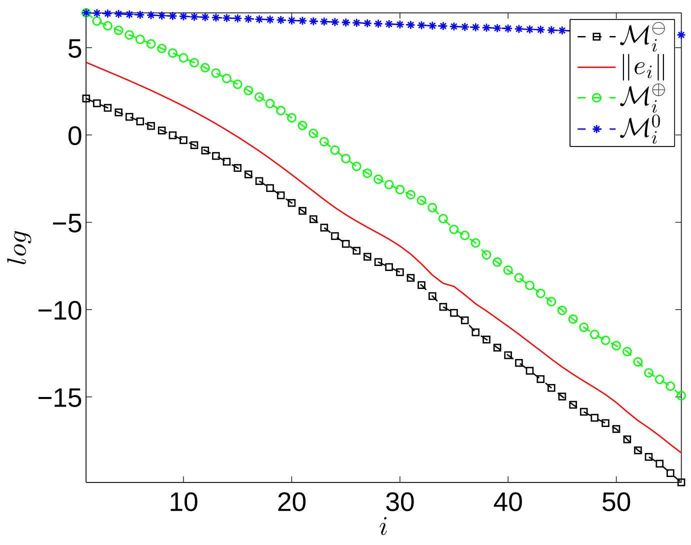


Figure 3.6: The error $\|e_i\|$ and the estimates $\mathcal{M}_i^{\ominus}$, $\mathcal{M}_i^{\oplus}$, $\mathcal{M}_i^{0}$ obtained by the SOR method, $n = 288$.

Example 3.9. We test the estimates discussed in Remark 3.3 on the problem considered in Example 3.8. All the discussed estimates are illustrated in Figure 3.7.

In the graphic the lower and upper bounds (3.32) are denoted by a 'dashed line/circled marker' and a 'dashed line/squared marker', respectively. The a priori majorant $\mathcal{M}_i^{cond}$ is marked by a 'dashed line/star marker'. We see that the most accurate estimate is $\mathcal{M}_i^{\lambda_{min}}$ denoted by the 'dot-dashed line/circled marker': it lies closer to the error ('line') compared to the other bounds.

Example 3.10. In this example, we discuss the case with $A = Q^* \Lambda Q$, where Q and Λ are an upper triangle matrix and a diagonal matrix, respectively. Let $\| x \| := (\Lambda x, x)$ and $\| x \|_* := (\Lambda^{-1} x, x)$. The error is measured in terms of the following norm:

$$\| Qx \| := (\Lambda Q x, Qx) = (Q^* \Lambda Q x, x). \tag{3.37}$$

Then, the general estimate for the error is

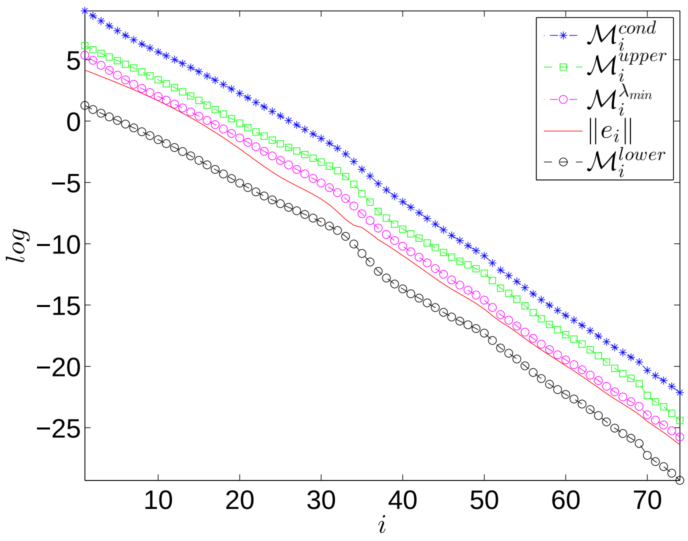


Figure 3.7: The estimates M_i^{lower}, M_i^{upper}, $M_i^{\lambda_{min}}$ and M_i^{cond} obtained to control the error $\|e_i\|$ during solving of the system using the SOR method, $n = 288$.

$$\| Q(x-\tilde{x}) \| \leq \| y - \Lambda Q\tilde{x} \|_* + \frac{1}{\lambda_{min}} \|Q^* y - f\|, \tag{3.38}$$

where flux $y = \Lambda Q\tilde{x} + \eta$. We square the inequality (3.38) and consider the right part of it.

$$\big(\| y - \Lambda Q\tilde{x} \|_* + \frac{1}{\lambda_{min}} \|Q^* y - f\|\big)^2 \leq$$
$$(1+\beta)\big(\Lambda^{-1}(y-\Lambda Q\tilde{x}), y - \Lambda Q\tilde{x}) + \frac{1}{\lambda_{min}}(1+\frac{1}{\beta})(Q^*y - f, Q^*y - f). \tag{3.39}$$

After substituting y to (3.39), we obtain

$$(1+\beta)(\Lambda^{-1}\eta,\eta) + \frac{1}{\lambda_{min}}(1+\frac{1}{\beta})(Q^*\eta + Q^*\Lambda Q\tilde{x} - f, Q^*\eta + Q^*\Lambda Q\tilde{x} - f) =$$
$$(1+\beta)(\Lambda^{-1}\eta,\eta) + \frac{1}{\lambda_{min}}(1+\frac{1}{\beta})((Q^*\eta, Q^*\eta) + 2(r, Q^*\eta) + (r,r)), \tag{3.40}$$

where $r := Q^*\Lambda Q\tilde{x} - f$. Consider (3.40) as a function of η and β

$$g(\eta,\beta) = (1+\beta)(\Lambda^{-1}\eta,\eta) + \frac{1}{\lambda_{min}}(1+\frac{1}{\beta})((Q^*\eta, Q^*\eta) + 2(r, Q^*\eta) + (r,r)), \tag{3.41}$$

which we need to minimize with respect to η and β. It is easy to see that

$$g'_\eta(\eta,\beta) = (1+\beta)\Lambda^{-1}\eta + (1+\frac{1}{\beta})\frac{1}{\lambda_{min}}(2QQ^*\eta + 2Qr + 2r) = 0. \tag{3.42}$$

From (3.42), it follows that

$$\eta = -\frac{1}{\lambda^2_{min}}(\Lambda^{-1} + \frac{1}{\beta\lambda^2_{min}}QQ^*s)^{-1}Qr. \tag{3.43}$$

To get a better accuracy, we optimize $q(\eta,\beta)$ with respect to β to find the best values

$$\begin{aligned}\beta = \sqrt{\frac{\kappa_2}{\kappa_1}},\ \ \kappa_1 &= (\Lambda^{-1}\eta,\eta),\\ \kappa_2 &= (Q^*\eta, Q^*\eta) + 2(r, Q^*\eta) + (r,r).\end{aligned} \tag{3.44}$$

Therefore, the general form of the estimate for the approximate solution $\tilde{x}$ has the form

$$\begin{aligned}|\!|\!| Q(x-\tilde{x}) |\!|\!| \leq \mathcal{M}^{gen} := \inf_{\beta>0,\eta} \Big\{ &(1+\beta)(\Lambda^{-1}\eta,\eta) + \\ &+ \frac{1}{\lambda_{min}}(1+\frac{1}{\beta})\big((Q^*\eta,Q^*\eta) + 2(r,Q^*\eta) + (r,r)\big)\Big\}, \text{ where } \beta > 0.\end{aligned} \tag{3.45}$$

In practice we often deal with problems that are are approximated by the following system

$$A = Q^*\Lambda Q, \tag{3.46}$$

where $Q \in M_{m\times n}$, $\Lambda \in M_{m\times m}$ and $m \ll n$.

The analysis of (3.43) shows that obtaining an optimal η, QQ^* is not time-consuming. Matrix that needs to be inverted belongs to $M_{m\times m}$, where m is considerably small.

In the example below we construct two different matrixes A_1 and A_2 with $n = 50$ from unitary matrices Q_1 and Q_2 (by using a similar function to the one in Listing A.1).

We compare the behavior of (3.45) with $y_i = \tilde{x}_iQ_i + \eta_i,\ i = 1,2$ in two different cases. One case is when $\eta_i = 0$ with the parameter $\beta \to \infty$, another one is when $\eta \neq 0$ and $\mathcal{M}^{gen}(\eta,\beta)$ is optimized according to η and β. We solve systems with matrices A_i, $i = 1,2$, the conditional numbers $cond(A_i) = 50$ and the right part of the system $f = \bar{1}$ by the Jacobi iteration method and obtain the following Figures 3.8 and 3.9. In the graphics for two different systems we show the efficiency index of the general estimate (optimized with respect to η and β) and the worse efficiency index of (3.30). In both cases we have got better efficiency indeces for a general estimate with flux $y_i = \tilde{x}_iQ_i + \eta_i$, where $\eta_i \neq 0$, $i = 1,2$.

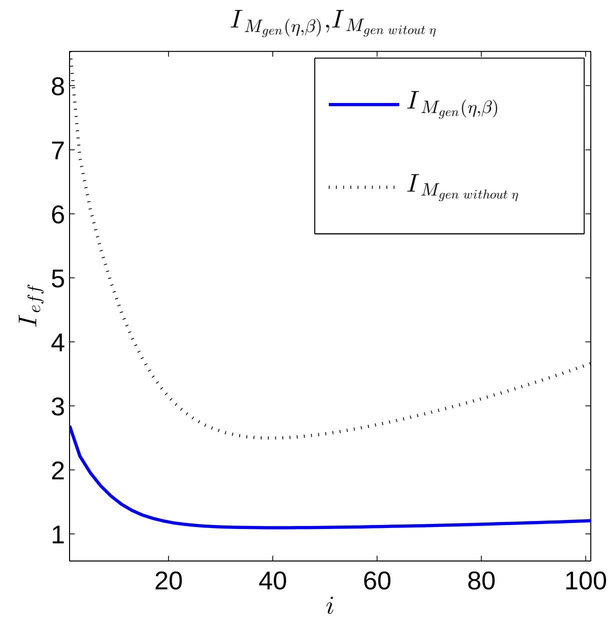


Figure 3.8: Efficiency indeces for general error estimates with and without optimization $(A_1 = Q_1' \Lambda Q_1)$.

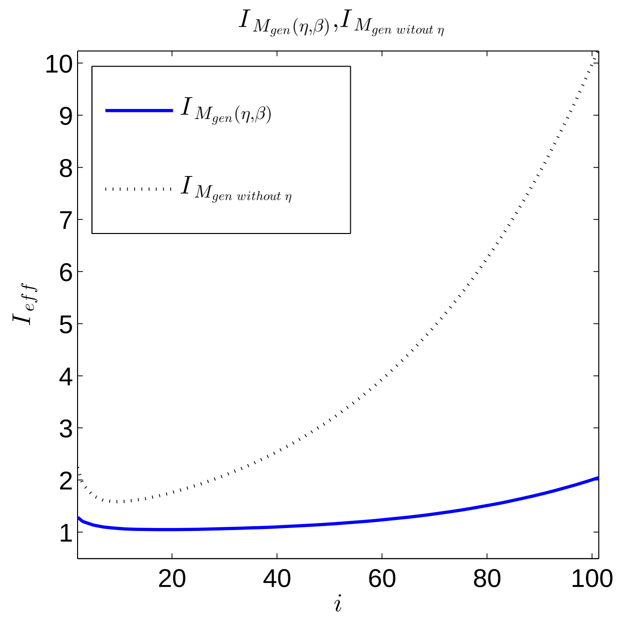


Figure 3.9: Efficiency indeces for general error estimates with and without optimization $(A_2 = Q_2' \Lambda Q_2)$.

4 The Picard–Lindelöf iteration method

In this chapter, we discuss a new version of the Picard–Lindelöf method for solving the Cauchy problem

$$\frac{du}{dt} = \varphi(u(t),\ t),\ u(t_0) = u_0, \tag{4.1}$$

where the solution $u(t)$ (which may be a scalar or vector function) must be found in the interval $[t_0,\ t_K]$.

The existence and uniqueness of the solutions follow from the Picard–Lindelöf theorem and the Picard's existence theorem or from the Cauchy–Lipschitz theorem (see [2, pp. 1–15], [5]).

The problem (4.1) can be numerically solved by various well-known methods (e.g., the methods of Runge–Kutta and Adams). Typically, the methods are furnished by a priori asymptotic estimates which show theoretical properties of the iteration algorithm. However, this estimates may have mainly a qualitative meaning and do not provide all necessary information about the error bounds. This is the goal of a posteriori error estimation methods. We deduce such type of estimates and suggest a version of the Picard–Lindelöf method as a tool for constructing a fully reliable approximation of (4.1).

The Picard-Lindellöf iteration is one of the efficient known numerical methods for ODEs. It can also be used not only for ODEs but also for t-dependent algebraic and functional equations (see, e.g. [11] and [12]). It was shown that the speed of convergence is quite independent of the step sizes. Numerical methods based on Picard–Lindelöf iterations for dynamical processes (the so-called waveform relaxation in the context of electrical networks) are discussed in [4].

The approach discussed in this paper is based on two-sided a posteriori estimates derived by Ostrowski [13] (see also systematic exposition presented in the books [7], [14]). The algorithm includes natural adaptation of the integration step and provides guaranteed bounds for the accuracy on the time interval $[t_0,\ t_K]$.

In Section 4.1, we present the main idea of the Picard–Lindelöf method and obtain the conditions, which not only provide convergence of the method but also allow to apply a posteriori error estimates. However, these estimates cannot be directly used. In practice computations based on the Picard–Lindelöf method we must take into account interpolation and integration errors. This analysis is done in Section 4.2. It leads to the error bounds, derived in Section 4.4, which include the interpolation and integration errors. The structure of the algorithm is exposed in Section 4.6, where results of numerical tests are presented.

4.1 The Picard–Lindelöf method

Assume that the function $\varphi(\xi(t),\ t)$ (which is allowed to be a vector-valued function) in (4.1) is continuous with respect to both variables in terms of continuous norm

$$\|u\|_{C([t_k,\,t_{k+1}])} := \max_{t\in[t_k,\,t_{k+1}]} |u(t)| \tag{4.2}$$

and satisfies the Lipschitz condition in the form

$$\|\varphi(u_2,\ t_2) - \varphi(u_1,\ t_1)\|_{C([t_1,\,t_2])} \leq \mathrm{L}_1\|u_2 - u_1\|_{C([t_1,\,t_2])} + \mathrm{L}_2|t_2 - t_1|,$$
$$\forall (u_1,\ t_1), (u_2,\ t_2)\ \in Q, \tag{4.3}$$

where L_1, L_2 are Lipschitz constants, and

$$Q := \Big\{(\xi,\ t) \mid \xi \in \mathrm{U},\ t_0 \leq t \leq t_N\Big\}. \tag{4.4}$$

U is the set of possible values of u which comes from an a priori analysis of the problem (it is clear that $u_0 \in \mathrm{U}$).

In the Picard–Lindelöf method, we represent the differential equation in the integral form

$$u(t) = \int_{t_0}^{t} \varphi(u(s),\ s)ds + u_0. \tag{4.5}$$

Now, the exact solution is a fixed point of (4.5), which can be found by the iteration method

$$u_j(t) = \int_{t_0}^{t} \varphi(u_{j-1}(s),\ s)ds + u_0. \tag{4.6}$$

We write the form $u_j = \mathcal{T}u_{j-1} + u_0$, where $\mathcal{T} : X \to X$ is the integral operator.

It is easy to show that the operator

$$\mathcal{T}u := \int_{t_k}^{t} \varphi(u(\tau), \tau)d\tau + u_{0,\,k} \tag{4.7}$$

is q-contractive on $I_k = [t_k,\ t_{k+1}]$, where I_k is a subinterval of the mesh $\mathcal{F}_K = \bigcup\limits_{k=0}^{K-1} [t_k, t_{k+1}]$ defined on the interval $[t_0,\ t_K]$, with respect to the norm $\|u\|_{C(I_k)}$, if condition

$$q := \mathrm{L}_1(t_{k+1} - t_k) < 1 \tag{4.8}$$

is provided.

Therefore, if the interval $[t_{k+1}, t_k]$ is small enough, then the solution can be found by the iteration procedure. In the next sections, we call this method the Adaptive Picard–Lindelöf (APL) method.

4.2 Application of the Ostrowski estimates

For the considered problem, the Ostrowski estimate reads as follows:

Theorem 4.2.1 ([13])**.** *Assume, that (4.8) is satisfied on $I_k := [t_k,\ t_{k+1}]$. Then, the following estimate holds:*

$$M_j^{\ominus} := \frac{1}{1+q}\|u_j - u_{j+1}\|_{C(I_k)} \leq \|u - u_j\|_{C(I_k)} \leq \frac{q}{1-q}\|u_j - u_{j-1}\|_{C(I_k)} =: M_j^{\oplus}. \tag{4.9}$$

Remark 4.3. It is possible to derive more accurate error bounds for $\|u - u_j\|_{C(I_k)}$ by using additional elements of the sequence $\{u_j\}_{j=1}^{\infty}$ that have indeces greater than j:

$$\|u - u_j\|_{C(I_k)} \leq M_j^{\oplus,p} := \frac{1}{1-q^p}\|u_j - u_{j+p}\|_{C(I_k)}. \tag{4.10}$$

By the mathematical induction method it can be proved that the optimal form of the majorant and minorant based on P correspondent elements of the sequence are as follows:

$$\begin{aligned} M_j^{\ominus,P} &:= \sup_{p=1,\dots,P}\left\{\frac{1}{1+q^p}\|u_j - u_{j+p}\|_{C(I_k)}\right\}, \\ M_j^{\oplus,P} &:= \inf_{p=1,\dots,P}\left\{\frac{1}{1-q^p}\|u_j - u_{j+p}\|_{C(I_k)}\right\}. \end{aligned} \tag{4.11}$$

However, estimates (4.9) cannot directly used because numerical approximations include the interpolation and integration errors, which must be taken into account by fully reliable schemes.

Let us discuss this issue within the paradigm of a the first step of APL:

$$u_1(t) = \int\limits_{t_0}^{t} \varphi(u_0(\tau),\ \tau)d\tau,\ t \in [t_0,\ t_1], \tag{4.12}$$

where u_0 is the initial approximation defined as a piecewise affine function on the mesh $\Omega_{S_k} = \bigcup\limits_{s=0}^{S_k-1} [z_s, z_{s+1}]$ on the interval $[t_0,\ t_1]$.

If $q < 1$ and u_1 is computed exactly, then

$$\|u_1(t) - u(t)\|_{C([t_0,\, t_1])} \leq \frac{q}{1-q}\|u_1(t) - u_0(t)\|_{C([t_0,\, t_1])}. \tag{4.13}$$

However, in general, u_1 is approximated by a piecewise affine continuous function

$$\bar{u}_1(t) = \pi u_1 \in CP^1([z_s,\ z_{s+1}]), \quad s = 0, ..., S_k - 1, \tag{4.14}$$

where π is the projection operator $\pi : C \to CP^1([t_0,\ t_1])$ satisfying the relation $\pi\ u(z_s) = \bar{u}(z_s)$. Thus, on the right-hand side of (4.13) we can estimate as follows:

$$\|u_1(t) - u_0(t)\|_{C([t_0,\, t_1])} \leq \|\bar{u}_1(t) - u_0(t)\|_{C([t_0,\, t_1])} + \|\bar{u}_1(t) - u_1(t)\|_{C([t_0\ t_1])}, \tag{4.15}$$

Here $\|\bar{u}_1(t) - u_1(t)\|_{C([t_0,\, t_1])} = \|\bar{e}_1\|_{C([t_0,\, t_1])}$ is an interpolation error. In general, this term is unknown, but we can estimate it by using an interpolation error estimate.

Numerical integration generates other errors which must be taken into account. Indeed, the values $\bar{u}(z_s), s = 0, ..., S_k$ can not be found exactly. Hence, at every node z_s instead of $\bar{u}_1(z_s)$ we have $\widehat{u}_1(z_s)$. Now, (4.15) implies

$$\begin{aligned}\|u_1(t) - u_0(t)\|_{C([t_0,\, t_1])} \leq\ & \|\widehat{u}_1(t) - u_0(t)\|_{C([t_0,\, t_1])} + \\ & + \|\widehat{u}_1(t) - \bar{u}_1(t)\|_{C([t_0,\, t_1])} + \|\bar{u}_1(t) - u_1(t)\|_{C([t_0,\, t_1])},\end{aligned} \tag{4.16}$$

where $\|\widehat{u}_1(t) - \bar{u}_1(t)\|_{C([t_0,\, t_1])} = \|\widehat{e}_1\|_{C([t_0,\, t_1])}$ is the integration error.

4.4 Estimates of interpolation and integration errors

4.4.1 Interpolation error

We study the difference between u_1 and $\bar{u}_1$, where $\bar{u}_1$ is the linear interpolant of u_1 defined at points $\{z_s\}_{s=0}^{S_k}$:

$$u_1(z_s) = \bar{u}_1(z_s) = \int\limits_0^{z_s} \varphi(u_0(t), t)dt. \tag{4.17}$$

For all $z \in [z_s,\ z_{s+1}]$,

$$\bar{u}_1(z) = u_1(z_s) + \frac{u_1(z_{s+1}) - u_1(z_s)}{\Delta_s}(z - z_s). \tag{4.18}$$

Then,

$$\begin{aligned}
&\bar{e} = \bar{u}_1(z) - u_1(z) = \\
&\quad = \left[\int\limits_0^{z_s} \varphi(u_0(t),\ t)dt + \frac{\int\limits_{z_s}^{z_{s+1}} \varphi(u_0(t),\ t)dt}{\Delta_s}(z - z_s) \right] - \int\limits_0^{z} \varphi(u_0(t),\ t)dt = \\
&\qquad = \frac{z - z_s}{\Delta_s} \int\limits_{z_s}^{z_{s+1}} \varphi(u_0(t),\ t)dt - \int\limits_{z_s}^{z} \varphi(u_0(t),\ t)dt.
\end{aligned} \tag{4.19}$$

Taking into account that u_0 is affinely interpolated, consider the last integral in right-hand side of (4.19)

$$\int\limits_{z_s}^{z} \varphi(u_0(t),\ t)dt = \int\limits_{z_s}^{z} \varphi(u_{0,\,s} + \frac{u_{0,\,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t)dt. \tag{4.20}$$

Define

$$\lambda = \frac{t - z_s}{\Delta_s} = \frac{t - z_s}{z_{s+1} - z_s}, \tag{4.21}$$

where z_s and z_{s+1} are nodes of the mesh defined in Section 4.2. Substitute $t = z_s + (z_{s+1} - z_s)\lambda$ to $\varphi(u_0(t), t)$

$$\begin{aligned}
&\varphi(u_{0,\,s} + \frac{u_{0,\,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) = \\
&\quad = \varphi(u_{0,\,s} + (u_{0,\,s+1} - u_{0,\,s})\lambda,\ z_s + \lambda(z_{s+1} - z_s)) = \\
&\qquad = \varphi(\lambda u_{0,\,s+1} + (1 - \lambda)u_{0,\,s},\ \lambda z_{s+1} + (1 - \lambda)z_s).
\end{aligned} \tag{4.22}$$

Let

$$\widetilde{\varphi}_{[s,\,s+1]} := \varphi_s + \frac{\varphi_{s+1} - \varphi_s}{\Delta_s}(t - z_s), \tag{4.23}$$

where $\varphi_s = \varphi(u_{0,\,s}, z_s)$ and $\varphi_{s+1} = \varphi(u_{0,\,s+1}, z_{s+1})$. Using (4.21), we rewrite (4.23)

$$\widetilde{\varphi}_{[s,\,s+1]} = \varphi_s + (\varphi_{s+1} - \varphi_s)\lambda = \lambda\varphi_{s+1} + (1-\lambda)\varphi_s. \tag{4.24}$$

Therethrough, we can derive the following estimate with help of (4.24) and (4.3):

$$\begin{aligned}
&\big|\varphi(u_{0,\,s} + \frac{u_{0,\,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) - \widetilde{\varphi}_{[s,\,s+1]}\big| \le \\
&\quad\le \big|\varphi(\lambda u_{0,\,s+1} + (1-\lambda)u_{0,\,s},\ \lambda z_{s+1} + (1-\lambda)z_s) - \lambda\varphi_{s+1} + (1-\lambda)\varphi_s\big| \le \\
&\quad\le (1-\lambda)\Big[\mathrm{L}_{1,\,s}|\lambda u_{0,\,s+1} + (1-\lambda)u_{0,\,s} - u_{0,\,s}| + \\
&\qquad\qquad\qquad + \mathrm{L}_{2,\,s}|\lambda z_{s+1} + (1-\lambda)z_s - z_s|\ \Big] + \\
&\qquad +\ \lambda\Big[\mathrm{L}_{1,\,s}|\lambda u_{0,\,s+1} + (1-\lambda)u_{0,\,s} - u_{0,\,s+1}| + \\
&\qquad\qquad\qquad + \mathrm{L}_{2,\,s}|\lambda z_{s+1} + (1-\lambda)z_s - z_{s+1}|\ \Big] \le \\
&\quad\le 2\lambda(1-\lambda)\Big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}|z_{s+1} - z_s|\ \Big] \\
&\quad\le 2\frac{(z_{s+1} - t)(t - z_s)}{\Delta_s^2}\Big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \Big]
\end{aligned} \tag{4.25}$$

We decompose (4.20)

$$\begin{aligned}
&\int\limits_{z_s}^{z} \varphi(u_0(t),\ t)dt = \\
&\quad = \int\limits_{z_s}^{z} \widetilde{\varphi}_{[s,\,s+1]}(t)dt + \int\limits_{z_s}^{z} \big[\varphi(u_{0,\,s} + \frac{u_{0,\,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) - \widetilde{\varphi}_{[s,\,s+1]}\big]dt.
\end{aligned} \tag{4.26}$$

Let us denote the first integral in the right hand side of (4.26) by $\widetilde{i}_s(z)$. Then,

$$\widetilde{i}_s(z) := \int\limits_{z_s}^{z} \big(\varphi_s + \frac{\varphi_{s+1} - \varphi_s}{\Delta_s}(t - z_s)\big)dt = (z - z_s)\big[\varphi_s + \frac{\varphi_{s+1} - \varphi_s}{2\Delta_s}(z - z_s)\big]. \tag{4.27}$$

The second integral in the right hand side of (4.26) is estimated with the help of (4.25):

$$\int\limits_{z_s}^{z} \Big|\varphi(u_{0,\,s} + \frac{u_{0,\,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) - \widetilde{\varphi}_{[s,\,s+1]}\Big|dt \leq$$

$$\leq \frac{2\Big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \Big]}{\Delta_s^2} \int\limits_{z_s}^{z} (t - z_s)(z_{s+1} - t)dt =$$

$$= \frac{2\Big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \Big]}{\Delta_s^2} \int\limits_{z_s}^{z} (t - z_s)(z_s + \Delta_s - t)dt = \tag{4.28}$$

$$= \frac{2\Big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \Big]}{\Delta_s^2}(z - z_s)^2\big[\frac{\Delta_s}{2} - \frac{z - z_s}{3}\big] =$$

$$= \frac{\Big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \Big]}{3\Delta_s^2}(z - z_s)^2(2z_s + 3\Delta_s - 2z).$$

Since

$$\max_{z\,\in\,[z_s,\,z_{s+1}]}(z - z_s)^2(2z_s + 3\Delta_s - 2z) = \Delta_s^3, \tag{4.29}$$

we find that

$$\int\limits_{z_s}^{z} \Big|\varphi(u_{0,\,s} + \frac{u_{0,\,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) - \widetilde{\varphi}_{[s,\,s+1]}\Big|dt \leq$$

$$\leq \frac{\big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \big]\Delta_s^3}{3\Delta_s^2} =$$

$$= \frac{\big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\ \big]\Delta_s}{3}. \tag{4.30}$$

We represent the interpolation error (4.19) by using (4.27),

$$\widetilde{u}_1(z) - u_1(z) = \frac{z - z_s}{\Delta_s}\int\limits_{z_s}^{z_{s+1}} \varphi(u_0(t),\ t)dt - \int\limits_{z_s}^{z} \varphi(u_0(t),\ t)dt =$$

$$= \frac{z - z_s}{\Delta_s}\,\widetilde{i}_s(z_{s+1}) - \widetilde{i}_s(z) + \varepsilon_1(z) + \varepsilon_2(z), \tag{4.31}$$

where

$$\varepsilon_1 = \int\limits_{z_s}^{z_{s+1}} \left|\varphi(u_{0,\,s} + \frac{u_{0,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) - \widetilde{\varphi}_{[s,s+1]}\right| dt,$$
$$\varepsilon_2 = \int\limits_{z_s}^{z} \left|\varphi(u_{0,\,s} + \frac{u_{0,s+1} - u_{0,\,s}}{\Delta_s}(t - z_s),\ t) - \widetilde{\varphi}_{[s,s+1]}\right| dt. \tag{4.32}$$

Thus, we estimate the interpolation error as follows:

$$\widehat{e} = \|\bar{u}_1(z) - u_1(z)\|_{C([z_s,\, z_{s+1}])} \leq$$
$$\leq \max_{z\in[z_s,\, z_{s+1}]} \left|\frac{z - z_s}{\Delta_s}\widetilde{i}_s(z_{s+1}) - \widetilde{i}_s(z)\right| + \max_{z\in[z_s,\, z_{s+1}]} |\varepsilon_1(z) + \varepsilon_2(z)|. \tag{4.33}$$

For the first term in the right hand side of (4.33) we have (see (4.27))

$$\max_{z\in[z_s,\, z_{s+1}]} \left|\frac{z - z_s}{\Delta_s}\widetilde{i}_s(z_{s+1}) - \widetilde{i}_s(z)\right| dt \leq \frac{|\varphi_{s+1} - \varphi_s|}{2\Delta_s} \max_{z\in[z_s,\, z_{s+1}]} |(z - z_s)(z_{s+1} - z)| \leq$$
$$\leq \frac{|\varphi_{s+1} - \varphi_s|}{2\Delta_s}\frac{\Delta_s^2}{4} = \frac{1}{8}|\varphi_{s+1} - \varphi_s|\Delta_s. \tag{4.34}$$

For the second term, we have (see (4.30))

$$\max_{z\in[z_s,\, z_{s+1}]} |\varepsilon_1(z) + \varepsilon_2(z)| \leq 2\frac{\Delta_s\big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\,\big]}{3}. \tag{4.35}$$

Hence, the overall estimate of the interpolation error has the form

$$\|\bar{u}_1(z) - u_1(z)\|_{C([z_s,\, z_{s+1}])} \leq$$
$$\leq \frac{\varphi_{s+1} - \varphi_s}{8}\Delta_s + \frac{2}{3}\Delta_s\big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\,\Delta_s\,\big]. \tag{4.36}$$

4.4.2 Integration error estimate

Assume that $f(t)$ is the Lipschitz function with constant L. Obtain the guaranteed bounds for the integration errors of the quadrature formula

$$\int\limits_a^b f(x)dx \simeq \frac{f(a) + f(b)}{2}(b - a) \tag{4.37}$$

Define four lines from Figure 4.1:

(1) $y = L(x - a) + f(a)$;

(2) $y = L(b - x) + f(b)$;

(3) $y = L(a - x) + f(a)$;

(4) $y = L(x - b) + f(b)$.

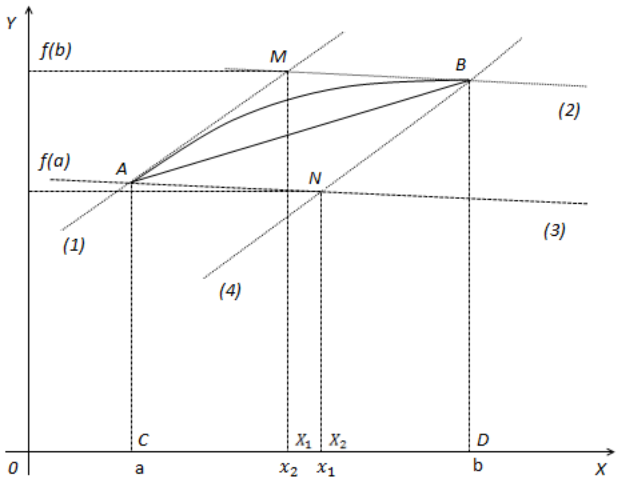


Figure 4.1: Illustration of the Lipschitz function $f(x)$ bounded by lines $y = Lx + b$.

Point x_1 is obtained by the intersection of the lines (3) and (4):

$$x_1 = \frac{a+b}{2} + \frac{f(a) - f(b)}{2L}. \tag{4.38}$$

Analogously, $x_2 = (1) \cap (2)$:

$$x_2 = \frac{a+b}{2} + \frac{f(b) - f(a)}{2L}. \tag{4.39}$$

Then,

$$\begin{aligned} S_{AMBDC} &= S_{AMX_2C} + S_{MBDX_2} = \\ &= \Big[2f(a) + L(x_2 - a)\Big]\frac{(x_2 - a)}{2} + \Big[2f(b) + \mathrm{L}(b - x_2)\Big]\frac{(b - x_2)}{2} \end{aligned} \tag{4.40}$$

$$\begin{aligned} S_{ANBDC} &= S_{ANX_1C} + S_{NBDX_1} = \\ &= \Big[2f(a) + \mathrm{L}(a - x_1)\Big]\frac{(x_1 - a)}{2} + \Big[2f(b) + \mathrm{L}(x_1 - b)\Big]\frac{(b - x_1)}{2} \end{aligned} \tag{4.41}$$

$$
\begin{aligned}
S_{est} &= \frac{S_{AMBDC} - S_{ANBDC}}{2} \\
&= \frac{1}{2}\big(f(a)(x_2 - x_1) + f(b)(x_1 - x_2)\big) + \\
&\qquad + \frac{\mathrm{L}}{4}\Big[(x_2 - a)^2 + (x_2 - b)^2 + (x_1 - a)^2 + (x_1 - b)^2\Big] = \\
&= \frac{1}{2}(x_2 - x_1)\big(f(a) - f(b)\big) + \\
&\qquad + \frac{\mathrm{L}}{4}\Big[(x_2 - a)^2 + (x_2 - b)^2 + (x_1 - a)^2 + (x_1 - b)^2\Big]
\end{aligned}
\tag{4.42}
$$

From (4.38) and (4.39), we deduce the following relations:

$$
\begin{aligned}
x_2 - x_1 &= \frac{f(b) - f(a)}{L}, \\
x_2 - a &= \frac{a + b}{2} - a + \frac{f(b) - f(a)}{2L} = \frac{b - a}{2} + \frac{f(b) - f(a)}{2L}, \\
x_2 - b &= \frac{a - b}{2} + \frac{f(b) - f(a)}{2L}, \\
x_1 - a &= \frac{b - a}{2} + \frac{f(a) - f(b)}{2L}, \\
x_2 - b &= \frac{a - b}{2} + \frac{f(a) - f(b)}{2L}.
\end{aligned}
\tag{4.43}
$$

Then,

$$
\begin{aligned}
S_{est} &= -\frac{[f(b) - f(a)]^2}{2L} + \frac{L}{4}\left[(b - a)^2 + \frac{(f(a) - f(b))^2}{Lx^2}\right] = \\
&= \frac{L}{4}(b - a)^2 - \frac{1}{4L}[f(b) - f(a)]^2.
\end{aligned}
\tag{4.44}
$$

We recall that $a = z_s$, $b = z_{s+1}$ and $\Delta_s = z_{s+1} - z_s$. Hence,

$$
S_{est} = \frac{\mathrm{L}}{4}\Delta_s^2 - \frac{1}{4\mathrm{L}_s}[\varphi_{s+1} - \varphi_s]^2, \tag{4.45}
$$

where $\mathrm{L}_s = \mathrm{L}_{1,\,s}\, l_s + \mathrm{L}_{2,\,s}$ (here, l_s is slope of the piecewise function on every interval $[z_s,\ z_{s+1}]$, $s = 0, ..., S_k - 1$).

Thus, the integration error can be estimated

$$
\|\widehat{u}_1(t) - \bar{u}_1(t)\|_{C([z_s,\, z_{s+1}])} \le \frac{\mathrm{L}_s}{4}\Delta_s^2 - \frac{1}{4\mathrm{L}_s}[\varphi_{s+1} - \varphi_s]^2. \tag{4.46}
$$

4.4.3 Guaranteed error bounds for the Picard–Lindelöf method

Thus, on every subinterval $[z_s,\ z_{s+1}]$ the interpolation error can be estimated with help of (4.36). Then, for whole interval $[t_0,\ t_1] := \bigcup\limits_{s=0}^{S_k-1}[z_s,\ z_{s+1}]$ the interpolation error estimate is the following:

$$\begin{aligned}\|\bar{u}_1(t) - u_1(t)\|_{C([t_0,\,t_1])} &\leq \\ \leq \sum_{s=0,\dots,S_k-1} \frac{\varphi_{s+1}-\varphi_s}{8}\Delta_s &+ \frac{2}{3}\big[\mathrm{L}_{1,\,s}|u_{0,\,s+1} - u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\,\big]\Delta_s. \qquad (4.47)\end{aligned}$$

Analogously, for the integration error:

$$\|\bar{u}_1(t) - \widehat{u}_1(t)\|_{C([t_0,\,t_1])} \leq \sum_{s=0,\dots,S_k-1} \frac{\mathrm{L}_s}{2}\Delta_s^2 - \frac{1}{2\mathrm{L}_s}[\varphi_{s+1}-\varphi_s]^2. \qquad (4.48)$$

Then, the inequality (4.16) implies the estimate:

$$\begin{aligned}\|u_1(t) - u_0(t)\|_{C([t_0,\,t_1])} \leq\ & \|\widehat{u}_1(t) - u_0(t)\|_{C([t_0,\,t_1])} + \\ &+ \sum_{s=0,\dots,S_k-1}\left(\frac{\varphi_{s+1}-\varphi_s}{8}\Delta_s + \frac{2}{3}\Delta_s\big[\mathrm{L}_{1,\,s}|u_{0,\,s+1}-u_{0,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\,\big]\right) + \\ &+ \sum_{s=0,\dots,S_k-1}\left(\frac{\mathrm{L}_s}{2}\Delta_s^2 - \frac{1}{2\mathrm{L}_\mathrm{s}}[\varphi_{s+1}-\varphi_s]^2\right). \qquad (4.49)\end{aligned}$$

After j steps of the iterations we obtain

$$\begin{aligned}\|u_{j+1}(t) - u_j(t)\|_{C([t_0,\,t_1])} \leq \mathrm{M}^{\oplus,1}_{j+1}(\widehat{u}_j) :=& \\ \|\widehat{u}_{j+1}(t) - \widehat{u}_j(t)\|_{C([t_0,\,t_1])} &+ E^1_{interp} + E^1_{integr}, \qquad (4.50)\end{aligned}$$

where

$$\begin{aligned}E^1_{interp} := \sum_{s=0,\dots,S_k-1}\Bigg(&\frac{\varphi(\widehat{u}_{j,\,s+1}, z_{s+1}) - \varphi(\widehat{u}_{j,\,s}, z_s)}{8}\Delta_s + \\ &+ \frac{2}{3}\Delta_s\big[\mathrm{L}_{1,\,s}|\widehat{u}_{j,\,s+1} - \widehat{u}_{j,\,s}| + \mathrm{L}_{2,\,s}\Delta_s\,\big]\Bigg) \qquad (4.51)\end{aligned}$$

and

$$E^1_{integr} := \sum_{s=0,\dots,S_k-1} \left(\frac{\mathrm{L_s}}{2}\Delta_s^2 - \frac{1}{2\mathrm{L_s}} \left[\varphi(\widehat{u}_{j,\,s+1}, z_{s+1}) - \varphi(\widehat{u}_{j,\,s}, z_s)\right]^2 \right), \qquad (4.52)$$

where for $j = 0$ the function $\widehat{u}_j$ is taken as piecewise affine interpolation of u_0, and for $j \geq 1$ it is taken from the previous iteration step.

The quantity $\mathrm{M}_j^{\oplus,1}$ is fully computable, and it shows the overall error associated with the step number j on the first interval.

Remark 4.5. Estimate of the overall error related to the interval $[t_0,\ t_K]$ includes all errors computed on the intervals. In other words the error associated with $[t_0, t_{k-1}]$ is appended to the error on $[t_{k-1},\ t_k]$ (which formally follows from the fact that the initial condition on $[t_{k-1},\ t_k]$ includes the errors on the previous intervals).

Thus, we have shown that for the problem (4.1) with the Lipschitz function φ fully guaranteed and computable bounds can be indeed derived, i.e. for every finite time interval $[t_0,\ t_K]$ and for every a priori required accuracy ε an approximate solution of the problem can be found by using the APL method discussed above.

4.6 APL algorithm and numerical examples

Let ε be a required accuracy of the approximate solution. Then, practical computation can be performed by Algorithm 7.

In general, the algorithm should start with the generation of a suitable mesh (i.e., select time intervals). Here, we do not discuss this question in detail, but only note that the *Mesh Guaranteed Procedure* must adapt the mesh to the nature of $\varphi(u(t), t)$, which requires information about U (see (4.4)). In practise, such an information can be obtained by solving the problem (4.1) numerically with the help of some heuristic (e.g., Runge–Kutta) method on a coarse mesh.

The APL algorithm is a cycle over all the intervals of the mesh $\mathcal{F}_K = \bigcup\limits_{k=0}^{K-1} [t_k,\ t_{k+1}]$. On each subinterval, the algorithm is realized as a subcycle (whose index is j). In the subcycle, we apply the PL method and try to find an approximation that meets the accuracy requirements imposed (i.e., the accuracy must be higher than ε^k). Initial data are taken from the previous step (for the first step, the initial condition is defined by u_0).

After computing an approximation on $[t_k,\ t_{k+1}]$ we use our majorant and find a guaranteed upper bound (which includes the interpolation and integration errors).

Algorithm 7 Algorithm of APL method

```
Input: $\varepsilon$ {required accuracy on the interval}, $u_0$ {input initial boundary condition}
$\mathcal{F}_K = \bigcup_{k=0}^{K-1} [t_k,\ t_{k+1}]$ { constructed by *Mesh Generation Procedure*}
$\varepsilon^k = \frac{\varepsilon}{K}$ {obtain accuracy of the approximate solution on interval $[t_k,\ t_{k+1}]$}
$\Omega_{S_k} = \bigcup_{s=0}^{S_k-1} [z_s,\ z_{s+1}]$ {initial mesh for each subinterval}
for $k = 1$ to $K$ do
  $j = 0$
  do
    if $k = 1$
      $a = u_0$
    else
      $a = v^{k-1}(t_{k-1})$
    endif
    $v_j^k = \textit{Integration Procedure}(\varphi, v_{j-1}^k, S_k) + a$
    calculate $E_{interp}^k$ and $E_{integr}^k$ by using (4.51) and (4.52)
    $M_j^{\oplus,k} = \|v_j^k - v_{j-1}^k\|_{C([t_{k-1},\ t_k])} + E_{interp}^k + E_{integr}^k$
    $e_j^{\oplus} = \frac{q}{1-q} M_j^{\oplus,k}$
    if $E_{interp}^k + E_{integr}^k > \varepsilon_k$
      $S_k = 2\,S_k$ {refine the mesh $\Omega_{S_k}$}
    endif
    $j = j + 1$
  while $e_j^{\oplus} > \varepsilon^k$
  $v^k = v_j^k$ {the approximate solution on the interval $[t_{k-1},\ t_k]$}
  $e^{\oplus,k} = e_j^{\oplus}$ {error bound achieved for the interval $[t_{k-1},\ t_k]$}
end for
Output: $\{v^k\}_{k=1}^K$ {the approximate solution}
        $\{e^{\oplus,k}\}_{k=1}^K$ {error bounds estimates on sub intervals}
```

Iterations are continued unless the required accuracy ε^k has been achieved. After that we save the results and proceed to the next interval.

Note that in Algorithm 1, we do not discuss in detail the process of integration on an interval, which is performed on a local mesh with a certain amount of subintervals (whose size is Δ_s). In principle, it may happen that the desired level of accuracy, ε^k, is not achieved with the Δ_s selected. This fact will be easily detected because the interpolation and integration errors will dominate and do not allow the overall error to decrease below ε^k. In this case, Δ_s must be reduced, and computations on the corresponding interval must be repeated.

Example 4.7. Consider the problem

$$\frac{du}{dt} = 4u\,t\,\sin(8t), \quad t \in [0, 3/2],$$
$$u(0) = u_0 = 1 \tag{4.53}$$

with the exact solution $u = e^{\frac{1}{16}\sin(8t) - \frac{1}{2}t\cos(8t)}$.

In Figure 4.2, we depict the error (bold dots), the error bounds computed by the Ostrowski estimates (dotted line) and by the advanced form of the estimate (dashed line). In order to make the results more transparent, we depict the approximate solution together with the zone which contains the exact solution (see Figures 4.3a and 4.4a). The form of this (shaded) zone is determined by the a posteriori estimates.

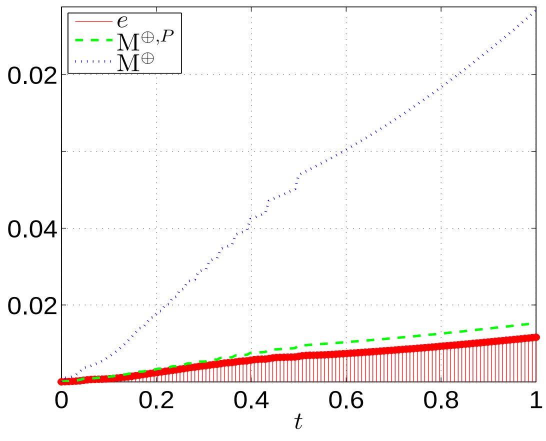
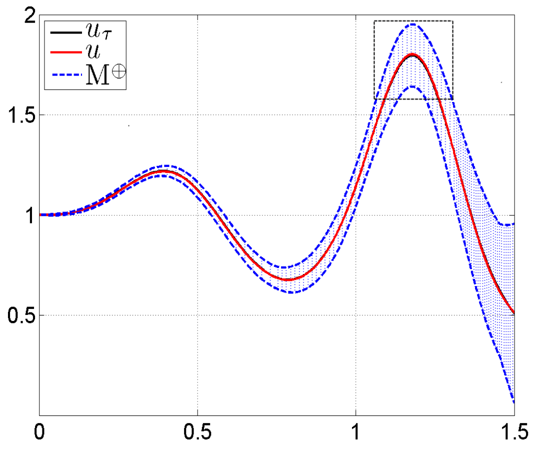


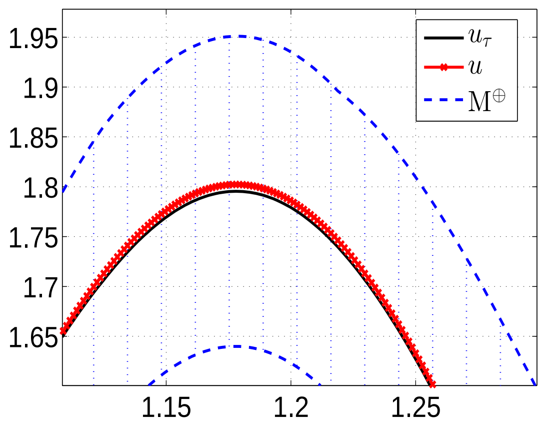
Figure 4.2: The error and yhe error majorants.

Thus, the APL method computes two-sided guaranteed bounds containing the exact solution. It may happen that the desired level of accuracy has been exceeded at some moment $t' < t_K$ and further Picard–Lindelöf iterations are unable to reduce the error. This situation may arise if the amount of internal points used for numerical integration on each interval is too small. In this case, we must enlarge the number of internal nodes (which will reduce the integration and interpolation errors) and repeat the computations. Numerical results illustrated in Figures 4.3a and 4.4a show that the advanced majorant provides much sharper bounds of the deviation.

Values of the components of the estimate (first term, *estimate of* $\|\bar{e}\|$ and *estimate of* $\|\widehat{e}\|$ from (4.50)) are presented in Table 4.1. We see that in this example the values of S_k were selected properly, so that the interpolation and integration error estimates are insignificant with respect to the first term.

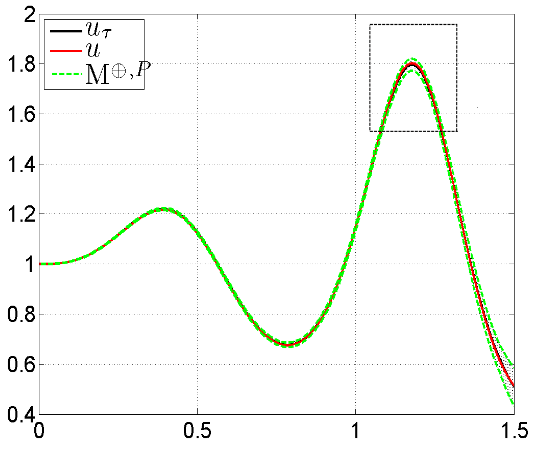


(a)

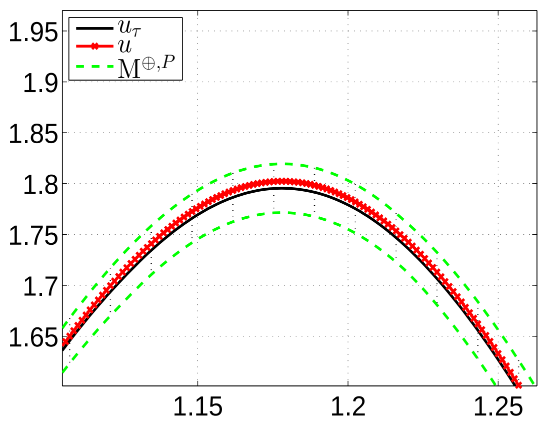


(b)

Figure 4.3: (a) The exact and the approximate solutions with guaranteed bounds of the deviation computed by Ostrowski estimate. (b) Zoomed interval of the exact and the approximate solutions with bounds of the deviation computed by the majorant.



(a)



(b)

Figure 4.4: (a) The exact and the approximate solutions with the guaranteed bounds of the deviation computed by the advanced form of estimate. (b) Zoomed interval of the exact and the approximate solutions with bounds of the deviation computed by the majorant.

Example 4.8. The APL method works with stiff problems as well. Consider the classical stiff equation

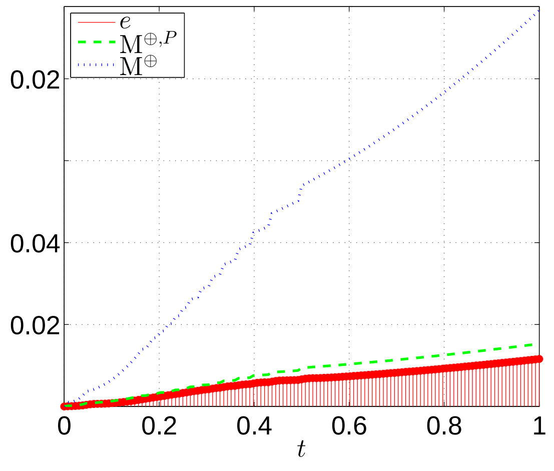
Table 4.1: Components of the general estimate.

estimate of $\|e_j\|$	*estimate of* $\|\bar{e_j}\|$	*estimate of* $\|\hat{e_j}\|$
2.2658e-002	8.6160e-008	9.5725e-008
4.6095e-002	1.8847e-007	5.8148e-007
5.4949e-002	2.5299e-007	5.9301e-007
7.4818e-002	2.5768e-007	2.3618e-006
9.5993e-002	3.0190e-007	2.3699e-006
1.0302e-001	3.4216e-007	2.3807e-006
1.5427e-001	4.8963e-007	2.4320e-006
1.5647e-001	6.1877e-007	2.4999e-006
2.3495e-001	9.4891e-007	2.6183e-006
2.7145e-001	9.8935e-007	2.6328e-006
3.0533e-001	9.9923e-007	2.6373e-006
3.2838e-001	1.0158e-006	2.6404e-006
4.4629e-001	1.0182e-006	2.6517e-006

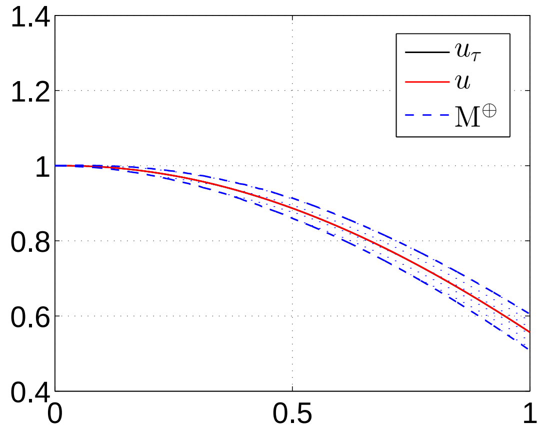
$$\begin{aligned} \frac{du}{dt} &= 50\cos(t) - 50u, \quad t = [0,1], \\ u(0) &= u_0 = 1 \end{aligned} \tag{4.54}$$

with the exact solution $u = \frac{1}{2501}e^{-50t} + \frac{2500}{2501}\cos(t) + \frac{50}{2501}\sin(t)$.

Analogously to the previous example, in Figure 4.5a the general error (lines with dots on the top) estimated by the Ostrowski estimate (dotted line) and the advanced form of estimate (dashed line) are illustrated. Another way to depict obtained results is shown in Figure 4.5b.



(a)



(b)

Figure 4.5: (a) The error and the error majorants. (b) The exact and the approximate solutions with the guaranteed deviation bound.

Example 4.9. The APL method can be also applied to stiff systems of ODEs. As an example, we consider the system

$$\begin{cases} \frac{du_1}{dt} = 998u_1 + 1998u_2, \\ \frac{du_2}{dt} = -999u_1 - 1999u_2, \\ u_1(t_0) = 1, u_2(t_0) = 1, \\ t \in [0,\ 5 \cdot 10^{-3}] \end{cases}$$

with the exact solutions $u_1 = 4e^{-t} - 3e^{-1000t}$ and $u_2 = -2e^{-t} + 3e^{-1000t}$. In Figures 4.6a, 4.6b, 4.7a and 4.7b, we present the same type of information (behavior of the solution and guaranteed bounds) as in the previous examples.

We note that for stiff equations getting an approximate solution with the guaranteed and sharp error bounds requires much larger expenditures than in relatively simple Examples 4.7 and 4.8. This results is not surprising, because (as it is quite natural to expect) for such type of problems, fully reliable computations will be much more expensive.

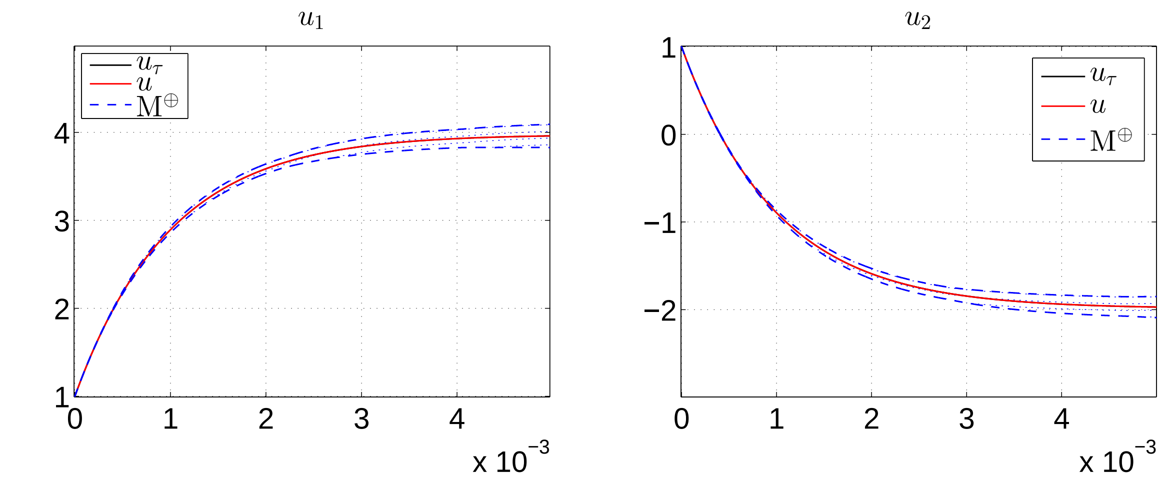


Figure 4.6: The exact solutions and the approximate solutions of the system, and the guaranteed error bounds computed by Ostrowski method.

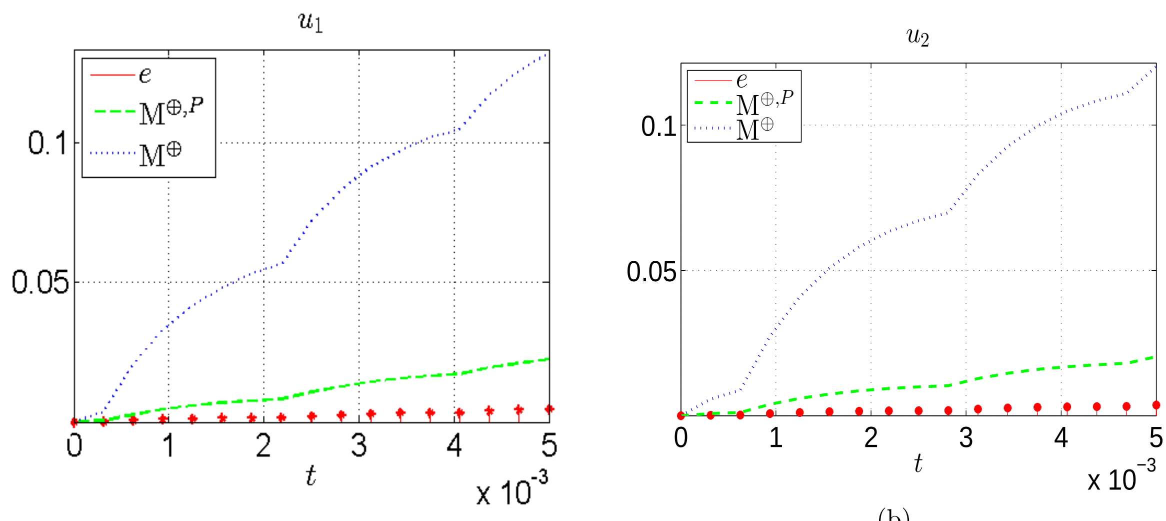


Figure 4.7: The error and error majorants for solutions u_1, u_2 of the system.

A Listing

A.1 Listing 1. Example 3.5

In the current listing we illustrate the MATLAB function that constructs symmetric matrices based only on the input size of the required matrices.

Listing A.1: A function to construct two symmetric matrices A and B.

```
function [A, B] = construct_symmetric_matrix_A_B(n)

% CONTRUCT_SYMMETRIC_A_B
% This function constructs two symmetric matrices A and B with size [n x n]
%
% SYNTAX: [A, B] = construct_symmetric_A(n)
%
% IN:   n   size of symmetric matrices
%
% OUT:    A, B  constructed matrices

% construct diagonal matrix with element from 1 to 50 on diagonal
D = diag(linspace(1, 50, n));
% construct matrix with size [n, n] filled with random number less that n
C = rand(n);

[Q1, ~] = qr(C);
[Q2, ~] = qr(C);

A = Q1' * D * Q1;

% get maximal eigen value of matrix A
mu = max(eig(A));
B = mu * Q2' * D * Q2;

end
```

A.2 Listing 2. Example 3.6

Describe the MATLAB code that was implemented to test the Richardson iteration scheme 3.20 in Example 3.6.

The main function `richardson_guaranteed_iteration_scheme_test()` contains construction of the problem including a system matrix, a system right part and a system size initialization. Here, we also define an accuracy level. In addition, the function includes technical procedures to open a file for outputting the obtained results.

Listing A.2: The main function for defining the problem and submitting the iteration process.

```
function richardson_guaranteed_iteration_scheme_test()

% RICHARDSON_GUARANTEED_ITERATION_SCHEME_TEST
% This function generates symmetric matrices A of different size to solve
% system A*x = f with Richardson guaranteed iteration scheme

% SYNTAX:   richardson_guaranteed_iteration_scheme_test()
% INPUT:    -
% OUTPUT:   -

% clean console window before submitting the function
clc;
warning off all;

% set the format of output data
format long e;

% init the example number
example_number = 5;

% open tex-file to output results
filename = strcat('../out/numerical-results-example-', num2str(example_number), '.tex');
texfile_id = fopen(filename, 'wt');

% set the required accuracy
eps   = 1e-10;

% cycle for testing systems of different size
for i = 1 : 3

  % set the initial data
    n   = 10^i;                            % system size
  x_0 = zeros(n, 1);                     % initial guess
  f   = ones(n, 1);                      % system right part
  A   = construct_symmetric_matrix(n);    % system matrix

  % get true error, a priori and a posteriori estimate
  [x, error, m_apriori, m_Ostr] = reliable_richardson_iteration_method(n, A, f, x_0, eps);

  % output true error and estimates to graphics and opened tex-file
  output_result_estimates_graphics(error, m_apriori, m_Ostr, n, example_number, texfile_id
      );
  output_result_estimates_to_texfile(texfile_id, error, m_apriori, m_Ostr);
end

% close all opened documents
fclose('all');

end
```

In order to test different problems, we insert a cycle over the system size n. We illustrate in the graphics by the function
`output_result_estimates_graphics` and output to the file by
`output_result_estimates_to_texfile` the results obtained after submission of the function `reliable_richardson_iteration_method` .

In the function `reliable_richardson_iteration_method` we solve a system of equations by the Richardson scheme including guaranteed a priori and a posteriori error bounds.

Listing A.3: The main function for defining the problem and submitting the iteration process.

```
function [x, error, m_apriori, m_ostr] = reliable_richardson_iteration_method(n, A, f, x_0
    , eps)
%
% RELIABLE_RICHARDSON_ITERATION_METHOD
% This function solves the system A*x = f with Richardson iteration scheme
% and control the reliability of the approximate solution by Ostrowski a
% posteriori and a priori estimates.

% SYNTAX:   [x, error, m_apriori, m_ostr] = ...
%                   reliable_richardson_iteration_method(n, A, f, x_0, eps)
%
% INPUT:    n           size of system
%           A           matrix of system
%           f           right part of system
%           eps         accuracy level between x_k and x_{k-1}, where k = 1,
%                       .., stoping criterion
% OUTPUT:   x           solution of the problem
%           error       truth error
%           m_apriori   a priori error estimate
%           m_ostr      a posteriori error estimate

% check if the input data has the minimum amount argument to solve the
% problem
if nargin < 3
    ERROR('Too few arguments');
end
% check if the rest of data is defined
if nargin < 5 || isempty(eps) == 1
    eps = 1e-5;
end
if nargin < 4 || isempty(x_0) == 1
    x_0 = ones(n, 1);
end

% check if matrix of the system is squared
if size(A,1) ~= size(A,2)
    ERROR('Matrix A should be square.');
elseif size(A,1) ~= size(f,1)
    ERROR('Mismatch between dimensions of A and f');
end
```

```
% init a priori, a posteriori estimates and true error storages
m_apriori = [];
m_ostr = [];
error = [];

% init unitary matrix
I = eye(n);

% set required paramaters
eigen_values = eig(A);
lambda_min = min(eigen_values);
lambda_max = max(eigen_values);

tau = 2/(lambda_min + lambda_max);
kappa = lambda_min / lambda_max;
q = (1 - kappa) / (1 + kappa);

% construct contractive operator and shift vector
L = I - tau * A;
b = tau * f;

% define exact solution for construction of model error
x_exact = A^(-1) * f;
% calculate first element of sequence of approximate solutions
x_1 = L * x_0 + f;
x_prev = x_1;

% init data before cycle
difference = 1;
k = 2;

while difference > eps

    % construct the next approximation from sequence
    x = L * x_prev + b;

    % calculate error, a priori estimate and Ostrowski estimate on the current step
    error(k)     =                 matrix_norm(x - x_exact, A);
    m_apriori(k) = (q^k/(1-q))   * matrix_norm(x_1 - x_0, A);
    m_ostr(k)    = (q / (1 - q)) * matrix_norm(x - x_prev, A);

    % calculate relative error between previous and current step
    difference = norm(x - x_prev) / norm(x);

    % go to the next step
    x_prev = x;
    k = k + 1;
end
end
```

In the beginning of the function `reliable_richardson_iteration_method(n, A, f, x_0, eps)` we check the correctness of the input data. Then, we initialize parameters which are used in the iteration process. We obtain the spectrum of A by the MATLAB function `eig(A)`, which returns a vector containing the eigenvalues of an input square matrix. The minimal and maximal eigenvectors are used in obtaining parameters τ, κ and q. Then, we need to construct the contractive operator L and shift the vector b before the iteration process submission. For a priori majorant construction, we need to know x_1 in advance.

On every iteration: we calculate a new approximation (3.1) with L and b defined before the cycle, construct a model error of this approximation and upper bounds to control it. We take the Ostrowski upper bound as a stopping criterion.

A.3 Listing 3. Example 3.7

For the third example, the main function is similar to (A.2), where we define a problem statement and submit the reliable Chebyshev iteration process. Consider only the Chebyshev iteration scheme implementation (Listing A.4).

Listing A.4: A guaranteed Chebyshev iteration scheme function.

```
function [x, error, m_apriori, m_ostr] = reliable_chebyshev_iteration_method(N, A, f, x_0,
     eps, n)
%
% RELIABLE_CHEBYSHEV_ITERATION_METHOD
% This function solves the system A*x = f with chebyshev iteration scheme
% and control the reliability of the approximate solution by Ostrowski a
% posteriori and a priori estimates.

% SYNTAX:   [x, error, m_apriori, m_ostr] =
%                   reliable_chebyshev_iteration_method(N, A, f, x_0, eps, n)
%
% INPUT:    N       size of system
%           A       matrix of system
%           f       right part of system
%           eps     accuracy level between x_k and x_{k-1}, where k = 1, ...
%           n       subcycle size, auxillary set size

% check if the input data has the minimum amount argument to solve the problem
if nargin < 3
    ERROR('Too few arguments');
end
% check if the rest of data is defined
if nargin < 6 || isempty(n) == 1
    n = 100;
end
if nargin < 5 || isempty(eps) == 1
    eps = 1e-5;
end
```

```
if nargin < 4 || isempty(x_0) == 1
    x_0 = ones(N, 1);
end

% check if matrix of the system is squared
if size(A,1) ~= size(A,2)
    ERROR('Matrix A should be square.');
elseif size(A,1) ~= size(f,1)
    ERROR('Mismatch between dimensions of A and f');
end

% init a priori, a posteriori estimates and error storages
error = [];
m_apriori = [];
m_ostr = [];

% set required parameters
eigen_values = eig(A);
lambda_min = min(eigen_values);
lambda_max = max(eigen_values);

tau_0 = 2/(lambda_min + lambda_max);
kappa = lambda_min / lambda_max;
rho_0 = (1 - kappa) / (1 + kappa);
rho_1 = (1 - sqrt(kappa)) / (1 + sqrt(kappa));
q_n = (2 * rho_1^n) / (1 + rho_1^(2*n));

% define exact solution for construction of true error
x_exact= A \ f;

% construct the first element of the sequence
x_1 = chebyshev_subcycle(x_0, n, A, f, tau_0, rho_0);
x_prev = x_1;
m_Ostr = q_n / (1 - q_n) * norm(x_1 - x_0);

% init cycle counter
i = 2;

while m_Ostr > eps

    % construct the next approximation from sequence
    x = chebyshev_subcycle(x_prev, n, A, f, tau_0, rho_0);

    % calculate error, a priori estimate and Ostrowski estimate on the current step
    error(i) = norm(x - x_exact);
    m_apriori(i) = q_n^i / (1 - q_n) * norm(x_1 - x_0);
    m_ostr(i) = q_n / (1 - q_n) * norm(x - x_prev);

    % define cycle stopping criterion
    m_Ostr = m_ostr(i);

    x_prev = x;
    i = i + 1;
end
end
```

Firstly, we check correctness of the input data. Secondly, we initialize storages for true error and their estimates. Then, it is necessary to define parameters τ_0, κ, ρ_0, ρ_1, $q(n)$ used in (3.23) before the cycle is submitted. We also construct x_1 for calculation of the a priori estimate and finally submit the iteration process.

New approximations are obtained by the Chebyshev subcycle which is encapsulated to a single function in Listing A.5.

Listing A.5: Chebyshev subcycle.

```
function x = chebyshev_subcycle(x_0, n, A, f, tau_0, rho_0)

% CHEBYSHEV_SUBCYCLE
% This function returns the new constructed term of the iteration sequence
% using a generated set of Chebyshev parameters

% SYNTAX:   x = chebyshev_subcycle(x_0, n)
% INPUT:    x_0     input vector
%           n       size of chebyshev parameter set
% OUTPUT:   x       output vector

% get length of the vector
N   = length(x_0);

% set initial value to the constractive operator L_n and shift vector b_n
L_n = eye(N);
b_n = zeros(N, 1);

% init unitary matrix
I = eye(N);

% init storages for auxillary parameters
t = zeros(1, n);
tau = zeros(1, n);

for k = 1 : n
    % generate a set of parameters for output x calculation
    t(k) = cos((2*k - 1)*pi/(2*n));
    tau(k) = tau_0 / (1 + rho_0 * t(k));

    % construct the constractive operator L_n and shift vector b_n
    L_n = (I - tau(k) * A) * L_n;
    b_n = (I - tau(k) * A) * b_n + tau(k) * f;
end

% construct new output vector
x = L_n * x_0 + b_n;

end
```

This function is presented by an n-size cycle, where we generate auxiliary parameters for construction of a transformation operator and shift vector.

A.4 Listing 4. Example 3.8

Next, we discuss the code implemented to provide an approximation of a given problem by the finite-difference method and to solve the obtained system of equations by the reliable SOR iteration method.

Listing A.6: Main function of problem statement and submitting the solver.

```
function reliable_iteration_methods_for_fdm_test()

% GUARANTEED_ITERATION_METHODS_FOR_FDM_TEST
% This function defines the Dirichlet problem, solve it by the finite difference method
% (with a reliable iteration scheme) and compares the obtained true error with a priori
% and a posteriori estimates

% SYNTAX:   reliable_iteration_methods_for_fdm_test()
% INPUT:    -
% OUTPUT:   -

% clean console window before submitting the function
clc;
format short;

% init the example number
example_number = 2;

% open file to output the results
filename = strcat('../out/numerical-results-example-', num2str(example_number), '.tex');
texfile_id = fopen(filename, 'wt');

% problem statement
u = @(x, y) (x^2 - 3*x) * (y^2 - 4*y);
a = @(x, y) x^2 + y + 1;
b = @(x, y) x + y + 2;
c = @(x, y) x + y + 4;

f = @(x, y) - (x^2 + y + 1) * 2 * (y^2 - 4*y) ...
            - (x + y + 2)   * 2 * (x^2 - 3*x) ...
            + (x + y + 4)   * (x^2 - 3*x) * (y^2 - 4*y);

% set domain boundary
x0 =  0.2;
xM =  3.3;
y0 =  0.5;
yN =  4.2;

% set mesh size [M x N]
M = 18;
N = 20;

% set the accuracy level
eps = 1e-5;

% get parameters of the grid defined on [x0, xM] x [y0, yN] domain with size M x N
[x, y, xh, yh] = get_mesh_parameters(x0, xM, y0, yN, M, N);
```

```
% get renumbered sets of the coefficients
[A, B, C, D, E, G] = construct_coefficients(f, a, b, c, x, y, xh, yh);

% get the projection of the exact solution to the mesh [x x y]
U = make_function_mesh(u, x, y);

% define the accuracy level for the Jacobi iteration method
eps_iter = 1e-6;
% get solution and parameter rho by the Jacobi iteration method
[V, rhoJ] = jacobi_iteration_scheme(U, A, B, C, D, E, G, M, N, eps_iter);
% define optimal parameter omega related from rhoJ
omega_opt = 2/(1 + sqrt(1 - rhoJ^2));

fprintf('Reliable SOR method\n\n');
format short;

% define the Dirichlet boundary condition
u0(1 : M, 1) = U(1 : M, 1);
u0(1 : M, N) = U(1 : M, N);
u0(1, 1 : N) = U(1, 1 : N);
u0(M, 1 : N) = U(M, 1 : N);

% construct a new system matrix and right part with renumbering unknown variable nodes
[Matrix, RightPart] = construct_system_from_problem_statement(u0, a, b, c, f, x, y, xh, yh
    , N, M);

% construct a contractive operator and shift vector for the SOR iteration process
[L, l, q] = get_contractive_operator_and_shift_based_on_SOR(Matrix, RightPart, M, N,
    omega_opt);

% define accuracy level for the reliable SOR iteration method
eps_iter = 1e-10;

[V, error, M_minus, M_plus, M0_plus, M_upper, M_lower, M_lambda_min, M_cond] = ...
    reliable_sor_iteration_scheme(u0, U, eps_iter, L, l, q, M, N, Matrix, RightPart);

% output result estimates to the graphics, console or tex-file
output_result_estimates_graphics(error, M_plus, M0_plus, M_minus, ...
                                        M_upper, M_lower, M_lambda_min, M_cond, ...
                                        (M - 2)*(N - 2), example_number, texfile_id);
output_result_estimates_to_console(error, M_plus, M0_plus, M_minus, ...
                                          M_lambda_min, M_cond, M_upper, M_lower);
output_result_estimates_to_texfile(texfile_id, error, M_plus, M0_plus, M_minus, ...
                                                      M_lambda_min, M_cond, M_upper,
                                                          M_lower);
end
```

In the main function we define the problem statement (3.33), which contains the definition of $a(x,y)$, $b(x,y)$, $c(x,y)$ and $f(x,y)$. Here, we set the domain boundary A, B, C, D and mesh sizes M, N. Finally, we init the accuracy of the approximate solution ε that we want to obtain.

The next step is problem approximation. First, we construct the mesh with size $[N \times M]$ (see Listing A.7).

Listing A.7: Function to generate the required mesh.

```
function [x, y, xh, yh] = get_mesh_parameters(A, B, C, D, M, N)
%
% GET_MESH_PERAMETERS
% This function constructs mesh [x x y] with steps xh, yh from input
% boundaries of the domain [A, B] x [C D] with size M x N
%
% SYNTAX:   [x, y, xh, yh] = get_mesh_parameters(A, B, C, D, M, N)
% INPUT:    A   lower bound of the interval on Ox-axis
%           B   upper bound of the interval on Ox-axis
%           C   lower bound of the interval on Oy-axis
%           D   upper bound of the interval on Oy-axis
% OUTPUT:
%           xh  step on the Ox-axis for interval [A, B]
%           yh  step on the Oy-axis for interval [C, D]
%           x   set of node on the interval [A, B] with step xh
%           y   set of node on the interval [A, B] with step xh

% define the mesh horizontal and vertical steps
xh = (B - A) / (M-1);
yh = (D - C) / (N-1);

% define horizontal and vertical node sets
x  = linspace(A, B, M);
y  = linspace(C, D, N);

end
```

Now, we need to renumber the internal nodes in the order from top to down, from right to left as it was explained in Example 3.8 (Figure 3.5). The renumbering procedure is shown in Listing A.8.

Listing A.8: Function to renumber the internal nodes of the set.

```
function [A, B, C, D, E, G] = construct_coefficients(f, a, b, c, x, y, xh, yh)
%
% CONSTRUCT_COEFFICIENTS
% This function constructs vectors with coefficients taken from the  problem
% statement
% SYNTAX:   [A, B, C, D, E, G] = construct_coefficients(f, a, b, c, x, y, xh, yh)
% INPUT:    f                   right-part function
%           a, b, c             function coefficients from problem statement
%           x                   horizontal part of mesh [x x y]
%           y                   vertical part of mesh [x x y]
%           xh                  step on the Ox-axis of the mesh [x x y]
%           yh                  step on the Oy-axis of the mesh [x x y]
% OUTPUT:   A, B, C, D, E, G    data vectors with coefficients

% get size of the mesh
M = length(x);
N = length(y);
```

```
% define storages for coefficients
A = zeros(M * N, 1);
B = zeros(M * N, 1);
C = zeros(M * N, 1);
D = zeros(M * N, 1);
E = zeros(M * N, 1);
G = zeros(M * N, 1);

% define counter variable
k = 1;

for i = 1 : M
    for j = 1 : N
        % contruct coefficient based on the fd scheme 'cross'
        vertical = -a(x(i), y(j))/xh^2;
        horizontal = -b(x(i), y(j))/yh^2;

        A(k) = vertical;
        C(k) = vertical;
        D(k) = horizontal;
        E(k) = horizontal;
        B(k) = -2 * (horizontal + vertical) + c(x(i), y(j));
        G(k) = f(x(i), y(j));
        k = k + 1;
    end
end

end
```

We project the solution $u(x, y)$ to the mesh $[x \times y]$, which is obtained by the function from Listing A.9, in order to compare this solution with the obtained approximations.

Listing A.9: Function that constructs projection of the exact function to the mesh.

```
function U = make_function_mesh(u, x, y)

% MAKE_FUNCTION_MESH
% This function projects input exact function to the mesh [x x y].
%
% SYNTAX:   U = make_function_mesh(u, x, y)
% INPUT:    u exact function
%           x   x-direction component of the mesh
%           y   y-direction component of the mesh
% OUTPUT:   U   function u projected to mesh [x x y]

% get size of the mesh
M = length(x);
N = length(y);

% define storage for projection function
U = zeros(M, N);

for i = 1 : M
    for j = 1 : N
       U(i, j) = u(x(i), y(j));
    end
end

end
```

Then, we submit the Jacobi iteration scheme to obtain ω_{opt}, which minimizes the SOR scheme iteration steps. The Jacobi method is presented in Listing A.10.

Listing A.10: Jacobi iteration scheme function.

```
function [V, rho] = jacobi_iteration_scheme(U, A, B, C, D, E, G, M, N, eps)
%
% JACOBI_ITERATION_SCHEME
% This function solves the system A*x = f rewritten in the data blocks A,
% B, C, D, E, G with the Jacobi iteration scheme

% SYNTAX:   [V, rho] = jacobi_iteration_scheme(U, A, B, C, D, E, G, M, N, eps_iter)
%
% INPUT:    U                   projection of the exact solution to the mesh
%           A, B, C, D, E, G    data block with matrix components
%           M                   mesh Ox-axis dimension
%           N                   mesh Oy-axis dimension
%           eps                 accuracy level between x_k and x_{k-1}, where k = 1,
%                               .., stopping criterion
% OUTPUT:
%           V       solution of the problem
%           rho     iteration process spectral radius

% define storages for solutions
Vprev    = ones(M,N);
V        = zeros(M,N);

% define accuracy criterion
diffence = 1;
step     = 1;
% define storages for error
diffs   = [];

while diffence > eps

    % calculate values inside area
    for i = 2 : M-1
        for j = 2 : N-1
            k = (i - 1) * N + j;
            V(i, j) = (G(k) - (C(k) * Vprev(i - 1, j) + ...
                               D(k) * Vprev(i, j + 1) + ...
                               E(k) * Vprev(i, j - 1) + ...
                               A(k) * Vprev(i + 1, j))) / B(k);
        end
    end

    % calculate boundary conditions
    V(1 : M, 1) = U(1 : M, 1);
    V(1 : M, N) = U(1 : M, N);
    V(1, 1 : N) = U(1, 1 : N);
    V(M, 1 : N) = U(M, 1 : N);
    % calculate difference
    diffs = [diffs norm(V - Vprev)];    %#ok<AGROW>
    diffence = diffs(end) / norm(V);

    % go to the next step
    Vprev = V;
    step = step + 1;
end

% calculate spectral radius
rho = diffs(end) / diffs(end - 1);

end
```

To apply the iteration scheme, we need to reconstruct the block matrix and the right part of the FDM approximation (the procedure is presented in Listing A.11).

Listing A.11: Function to construct block matrix and right part from FDM problem approximation.

```
function [A, F] = construct_system_from_problem_statement(u0, a, b, c, f, x, y, xh, yh, N,
     M)
%
% CONSTRUCT_SYSTEM_FROM_PROBLEM_STATEMENT
% This function constructs system that follows from the problem approximation
% by the finite difference method
%
% SYNTAX:   [A, F] = construct_system_from_problem_statement(u0, a, b, c, f, x, y, xh, yh,
     N, M)
% INPUT:    u0          function on boundary (Dirichlet boundary condition)
%           a, b, c     function coefficients from problem statement
%           f           right-part function
%           x           horizontal part of mesh [x x y]
%           y           vertical part of mesh [x x y]
%           xh          step on the Ox-axis of the mesh [x x y]
%           yh          step on the Oy-axis of the mesh [x x y]
%           N           Ox-axis size of the mesh
%           M           Oy-axis size of the mesh
% OUTPUT:   A           block matrix constructed from problem approximation
%           F           right part constructed from problem approximation

% define storages for data
A = zeros((N-2)*(M-2));
F = zeros((N-2)*(M-2), 1);
G = zeros((N-2)*(M-2), 1);

% define counter
k = 1;
% define shift for renumbered problem
N_ = N - 2;

for i = 2 : M-1
    for j = 2 : N-1
        % define repeated components
        vertical = -a(x(i), y(j))/xh^2;
        horizontal   = -b(x(i), y(j))/yh^2;
        central = -2 * (horizontal + vertical) + c(x(i), y(j));

        g = f(x(i), y(j));
        A(k, k) = central;
        if i == 2 && j == 2                 % left-up corner node
            A(k, k+1) = horizontal;
            A(k, k+N_) = vertical;
            F(k) = g - horizontal*u0(i, j-1) - vertical*u0(i-1, j);
        elseif i > 2 && i < M-1 && j == 2   % left boundary nodes
            A(k, k-N_) = vertical;
            A(k, k+1) = horizontal;
            A(k, k+N_) = vertical;
            F(k) = g - horizontal*u0(i, j-1);
        elseif i == M-1 && j == 2           % left-down node
            A(k, k-N_) = vertical;
            A(k, k+1) = horizontal;
            F(k) = g - horizontal*u0(i, j-1) - vertical*u0(i+1, j);
        elseif i == M-1 && j > 2 && j < N-1 % down boundary nodes
            A(k, k-1) = horizontal;
            A(k, k+1) = horizontal;
            A(k, k-N_) = vertical;
            F(k) = g - vertical*u0(i+1, j);
        elseif i == M-1 && j == N-1         % right-down node
            A(k, k-1) = horizontal;
            A(k, k-N_) = vertical;
            F(k) = g - horizontal*u0(i, j+1) - vertical*u0(i+1, j);
        elseif i > 2 && i < M-1 && j == N-1 % right boundary nodes
            A(k, k-1) = horizontal;
            A(k, k+N_) = vertical;
            A(k, k-N_) = vertical;
            F(k) = g - horizontal*u0(i, j+1);
        elseif i == 2 && j == N-1           % right-up node
            A(k, k-1) = horizontal;
            A(k, k+N_) = vertical;
            F(k) = g - horizontal*u0(i, j+1) - vertical*u0(i-1, j);
        elseif i == 2 && j > 2 && j < N-1    % up boundary nodes
            A(k, k-1) = horizontal;
            A(k, k+1) = horizontal;
            A(k, k+N_) = vertical;
            F(k) = g - vertical*u0(i-1, j);
        else                                 % internal nodes
            A(k, k-1) = horizontal;
            A(k, k+1) = horizontal;
            A(k, k+N_) = vertical;
            A(k, k-N_) = vertical;
            F(k) = g;
        end
        G(k) = g;
        k = k + 1;
    end
end
end
```

We also encapsulated construction of the contractive operator and shift vector to the separate function (Listing A.12).

Listing A.12: Function to construct contractive operator and shift vector for iteration process.

```
function [L, b, q] = get_contractive_operator_and_shift_based_on_SOR(A, f, M, N, omega)
%
% GET_CONTRACTIVE_OPERATOR_AND_SHIFT_BASED_ON_SOR
% This function constructs the contractive operator and the shift-vector based on the SOR
    iteration scheme

% SYNTAX:   [L, b, q] = get_contractive_operator_and_shift_based_on_SOR(A, f, M, N, omega)
% INPUT:    A       system matrix
%           f       system right part
%           M       size of x - direction of mesh
%           N       size of y - direction of mesh
%           omega   parameter in interval (0, 2)
% OUTPUT:   L       matrix of contractive operator
%           b       shift vector
%           q       contractivity parameter of operator L

% define size of contractive operator
n = (M - 2)*(N - 2);

% init unit matrix
I = eye(n);
% get diagonal component of matrix A
D = diag(diag(A));
% get lower triangular part of matrix A without diagonal
LA = tril(A, -1);
% define matrix according to SOR scheme
B = D/omega + LA;

% construct contractive operator for iteration process
L = I - B \ A;
% construct shift vector for iteration process
b = B \ f;
% calculate contractivity parameter
q = norm(L);
end
```

The reliable iteration process itself is included in the function `reliable_sor_iteration_scheme` from Listing A.13. This obtained results are saved as graphics and tex-file and presented in the command window (`output_result_estimates_graphics`, `output_result_estimates_to_console` and `output_result_estimates_to_texfile` functions).

Listing A.13: Function to renumber internal nodes of the set.

```
function [V, error, M_minus, M_plus, M0_plus, M_upper, M_lower, M_lambda_min, M_cond] =
    ...
    reliable_sor_iteration_scheme(u0, U, eps, L, b, q, M, N, A, f)

% RELIABLE_SOR_ITERATION_METHOD
% This function solves the system A*x = f with SOR iteration scheme
% and controls the reliability of the approximate solution by Ostrowski a
% posteriori, a priori and some estimates based on the residual.

% SYNTAX:   [V, error, M_minus, M_plus, M0_plus, i_upper, i_lower, i_Rep, i_cond] = ...
%           reliable_sor_iteration_method(u0, U, eps_iter, L, b, q, M, N, A, f)
%
% INPUT:    u0          initial guess in iteration process
%           U         projection of the exact solution to the mesh
%           f           right part of system
%           eps         accuracy level between x_k and x_{k-1}, where k = 1,
%                       .., stoping criterion
%           L           contractive operator matrix
%           b           shift vector
%           q           contractivity parameter
%           M           size of x - direction of mesh
%           N           size of y - direction of mesh
%           A           system matrix
%           f           system right part
% OUTPUT:
%           V               solution of the problem
%           error           true error
%           M_minus         a posteriori lower error estimate
%           M_plus          a posteriori upper error estimate
%           M0_plus         a priori error estimate
%           M_upper         upper error bound based on residual
%           M_lower         lower error bound based on residual
%           M_lambda_min    upper error bound based on residual and mininum eigenvalue
%           M_cond          upper error bound based on condition number of
%                           system matrix

% init solutions storages
Vprev    = zeros(M, N);
V        = zeros(M, N);
% defined auxiliary sizes of the problem after renumbering
M_ = M - 2;
N_ = N - 2;
% init error and estimates storages
error   = [];
M_plus  = [];
M0_plus = [];
M_minus = [];
M_upper = [];
M_lower = [];
M_cond = [];
M_lambda_min = [];

% get solutions in 1D
x0       = convert_solution_from_2D_to_1D(V(2:M-1, 2:N-1), N_, M_);
x_prev   = convert_solution_from_2D_to_1D(Vprev(2:M-1, 2:N-1), N_, M_);
```

```
x_exact  = convert_solution_from_2D_to_1D(U(2:M-1, 2:N-1), N_, M_);

% define the error upper bound as accuracy criterion
Maj = (q/(1 - q)) * norm(R(x0, L, b));
% init iterator
i = 1;

while Maj > eps

    % construct next step of the approximation sequence
    x = L * x_prev + b;

    % get solution in 2D
    V = convert_solution_from_1D_to_2D(x, u0, N, M);
    Vprev = convert_solution_from_1D_to_2D(x_prev, u0, N, M);

    % construct true error
    error(i) = norm(x - x_exact);

    % get error estimates based on contractivity
    [M_minus(i), M_plus(i), M0_plus(i)] = ...
        get_contractivity_estimates(x, x_prev, x0, i, L, b, q);

    % get error estimate based on residual of the problem
    [M_upper(i), M_lower(i), M_lambda_min(i), M_cond(i)] = ...
        get_estimates_based_on_residual(x, x_exact, A, f);

    % set the accuracy criterion
    Maj = M_plus(i);

    x_prev = x;
    i = i + 1;
end
end
```

Finally, we present functions, which construct estimates included in Listings A.14 and A.15.

Listing A.14: Function to construct estimates based on the contractive properties of iteration operator.

```
function [M_minus, M_plus, M0_plus] = get_contractivity_estimates(x, x_prev, x_0, i, L, b,
     q)
%
% GET_CONTRACTIVITY_ESTIMATES
% This function constructs error estimates based on the contractive properties of L from
    input data

% SYNTAX:   [M_minus, M_plus, M0_plus] = get_contractivity_estimates(x, x_prev, x_0, i, L,
     b, q)
% INPUT:    x           current input approximation
%           x_prev      previous input approximation
%           x_0         initial input approximation
%           i           step number
%           L           contractive operator
%           b           shift vector
%            q           contractive parameter
% OUTPUT:    M_minus     a posteriori lower bound
%            M_plus      a posteriori upper bound
%            M0_plus     a priori upper bound

% construct error estimates
M_plus  = (q  /(1-q)) * norm(R(x_prev, L, b));
M0_plus = (q^i/(1-q)) * norm(R(x_0, L, b));
M_minus = (1  /(1+q)) * norm(R(x, L, b));
end
```

Listing A.15: Function to construct estimates based on the residual.

```
function [M_upper, M_lower, M_lambda_min, M_cond] = get_estimates_based_on_residual(x,
    x_exact, A, f)
%
% GET_ESTIMATES_BASED_ON_RESIDUAL
% This function constructs estimates based on the residual of system A*x = b and
    properties of matrix A
%
% SYNTAX:   [M_upper, M_lower, M_lambda_min, M_cond] = ...
%               get_estimates_based_on_residual(x, x_exact, A, b)
% INPUT:     x                approximate solution of the system
%            x_exact          exact solution of the system
%            A                system matrix
%            f                system right-part
% OUTPUT:    M_upper          error upper bound based on residual
%            M_lower          error lower bound based on residual
%            M_lambda_min     error upper bound based on residual and
%                             lambda_min
%            M_cond           error lower bound based on conditional number
%                             of system matrix

% define parameter
p = 2;
q = p /(p - 1);

% define the residual of the problem
r = A*x - f;

% define minimal eigenvalue of matrix
lambda_min = min(eig(A));

% construct auxiliary term and constant
a_p = norm(r, p)^2 / norm(A'*r, q);
C_p = norm(A', q) * norm(A^(-1), p);

% construct error estimates
M_upper = C_p * a_p;
M_lower = a_p;
M_lambda_min = 1/lambda_min * norm(r, p);
M_cond = cond(A, p) * norm(x_exact, p) * norm(r, p) / norm(f, p);
end
```

A.5 Listing 5. Example 3.10

Consider the code that was implemented to test the general form of the upper bound (3.45) in Example 3.10. In the main function, we define matrices Q by loading them from the text file (generated and saved in advance).

Listing A.16: Function to test general estimate.

```
function general_estimate_test()

% GENERAL_ESTIMATE_LAMBDA_MIN_COMPARISON_TEST
% This function tests behavior of the general estimate and compares it to
% the well known estimate M_lambda_min.
%
% SYNTAX:   general_estimate_test()
% INPUT:  -
% OUTPUT:   -

% clear the console screen
clc;
warning off all;
% define format of console output data
format long e;

% define example number
example_number = 9;
% define matrix number
matrix_number = 9;

% open file to output results
filename = strcat('../out/numerical-results-example-', num2str(example_number), '.tex');
texfile_id = fopen(filename, 'wt');

% set the required accuracy
eps = 1e-10;

% size of the system and unknown vector
n = 50;

% construct matrix A = Q' * D * Q;
Q = load_matrix(matrix_number);
Lambda = diag(linspace(1, n, n));
% construct the right part
f = ones(n, 1);

% construct error and general form of estimate during solution search by iteration Jacobi
    method
[i_m_gen_with_eta, i_m_gen_without_eta] =
    jacobi_method_with_general_and_lambda_min_estimate(n, Q, Lambda, f, eps);

% output obtained results to the graphics
output_result_effectivity_indexes_graphics(i_m_gen_with_eta, i_m_gen_without_eta, ...
                                           n, example_number, matrix_number, texfile_id);
end
```

The estimate (3.45) is tested on the sequence of approximate solutions generated by the Jacobi iteration scheme. The function `jacobi_method_with_general_and_lambda_min_estimate` has the same structure as the iteration function discussed before. We discuss the function which we use to construct the general estimate (see Listing A.17).

Listing A.17: Function to construct general estimate.

```
function e = get_m_gen_with_eta(x, Q, Lambda, y, f, lambda, eta)
%
% GET_M_GEN_WITH_ETA
% This funcition contruct general form of estimate with optimized shift
% in flux y = Lambda * Q * x + eta and optimized parameter beta.
%
% SYNTAX:   e = get_m_gen_with_eta(x, Q, Lambda, y, f, lambda, eta)
%
% INPUT:    x       approximate solution
%           Q       upper diagonal unitary matrix
%           Lambda  diagonal matrix
%           y       flux vector
%           f       system right part
%           lambda  minimal eigenvalue
%           eta     optimized shift vector in flux y
%
% OUTPUT:   e       general form of error estimate

% compute residual of system Q' * Lambda * Q * x = f, where x is
% approximate solution
residual = r(Q, Lambda, x, f);

% compute optimal parameter beta
C1 = (Lambda^(-1) * eta)' * eta;
C2 = ((Q * Q' * eta)' * eta + 2 * eta' * (Q * residual) + residual'*residual)/lambda^2;
beta = sqrt(C2/C1);

%
v = residual + Q' * eta;
e = (1 + beta)*((Lambda^(-1) * eta)' * eta) + (1 + 1/beta)/lambda^2 * (v' * v);

end
```

We calculate the optimal β according to the formula (3.44) before the estimate construction w. To get the optimal η, we use the function from Listing A.18.

Listing A.18: Function to construct optimal η.

```
function eta = get_optimal_eta(Lambda, Q, beta, lambda, x, f)
%
% GET_OPTIMAL_ETA
% This function constructs optimized eta for general estimate.
%
% SYNTAX:   eta = get_eta(Lambda, Q, beta, lambda, x, f)
%
% INPUT:    Lambda  diagonal matrix
%           Q       upper diagonal unitary matrix
%           beta    positive parameter
%           lambda  minimal eigenvalue
%           x       approximate solution
%           f       system right part
% OUTPUT:   eta     optimized vector

eps = 10^(-5);
n = length(x);

% get eta with cycle coordinate search procedure
eta = cycle_coordinate_search(@(eta)f_eta(eta, x, Q, Lambda, f, beta, lambda), ones(n, 1),
     eps);

end
```

It is implemented using the cycle coordinate search algorithm (Listing A.19) with golden 1D search included.

Listing A.19: Cycle coordinate search function.

```
function x = cycle_coordinate_search(f, x_0, eps)
%
% CYCLE_COORDINATE_SEARCH
% This function takes input function, initial vector and accuracy level eps
% and returns optimal argument for function f found with cycle coordinate
% search
%
% SYNTAX:   x = cycle_coordinate_search(f, x_0, eps)
% INPUT:  f     function to minimize
%       x_0   initial point
%       eps   accuracy level
% OUTPUT:   x   optimization point

% set the dimension of the problem according to the input vector x_0
n = length(x_0);

% construct set of basic functions
e = eye(n);

% init valiables before cycle
x = x_0;
x_prev = zeros(n, 1);
k = 1;
lambda_min = -5;
lambda_max = 5;
```

```
while norm(x - x_prev) > eps

    % remember x before cycle of coordinate descent
    x_prev = x;

    % cycle of coordinate descent
    for i = 1 : n
        % get optimal parameter lambda with golden section search algorithm
        lambda = golden_section_search(@(lambda)(f(x + e(:, i) * lambda)), lambda_min,
            lambda_max, 10^(-10));
        % transform vector in one-coordinate direction
        x = x + lambda * e(:, i);
    end
    k = k + 1;
end
end
```

A.6 Listing 6. Example 4.7

Consider implementation of the reliable Picard-Lindelöf method applied to the problem (4.53) in Example 4.7. Again, the main function contains the problem definition (see Listing A.20).

Listing A.20: Function to test reliable Picard-Lindelöf method.

```
function picardLindelofMethodForEquationsTest()
%
% picardLindelofMethodForEquationsTest
% This function defines test problem, data required to solve it with interation
% Picard-Lindelof method and submits the solver
%
% SYNTAX:   picardLindelofMethodForEquationsTest()
% INPUT:    -
% OUTPUT:   -

tic;

clc;
warning off all;
format long e;

% define symbolic variables
syms Phi u t

% open tex-file for results of calculation
file_name = strcat('../out/numerical-results-for-example-13.tex');
texfile_id = fopen(file_name, 'wt');

example_number = 13;

% define problem statement: right part of problem, time interval and initial condition
```

```
symPhi = u*4*t*sin(8*t);
timeInterval = [0, 3/2];
u_0 = 1;

% get 2nd derivative of right part and exact solution of problem
[symd2Phidudt, symUExact] = problemAnalisys(symPhi, timeInterval, u_0);

% set the required accuracy
E = 10^(-1);

% set the required constant
q = 0.5;

% output problem statement to console and tex-file
outputProblemStatementToConsole(example_number, symPhi, timeInterval, u_0, symUExact)
outputProblemStatementToTexFile(texfile_id, example_number, symPhi, timeInterval, u_0,
    symUExact);

% submit Picard-Lindelof iteration method
picardLindelofSolver(texfile_id, example_number, ...
           u_0, ...
           symd2Phidudt, ...
           timeInterval(1), timeInterval(2), ...
           E, q);

% close all opened files
fclose('all');

toc;

end
```

Here, the function `picardLindelofSolver` is responsible for solving the problem.

Listing A.21: Function to solve ODE with guaranteed accuracy.

```
function picardLindelofSolver(file_id, example_number, u_0, symd2Phidudt, t_0, t_N, E, q)
%
% picardLindelofSolver
% This function solves the problem with guaranteed Picard-Lindelof
% iteration method
%
% INPUT:    file_id         open tex-file id
%           example_number  example number used to save the results
%           u_0             Dirichlet boundary coundition
%           symd2Phidudt    symbolic function, 2nd derivative of the right part of problem
%                           with respect to variables 'u' and 't'
%           t_0             upper border of problem time interval
%           t_N             lower border of problem time interval
%           E               required accuracy level
%           q               required contractivity constant
% OUTPUT:   -

% define symbolic variables
syms u tau t
```

```
Solution = struct('f', 0);

% get time-grid based on right part function properties and
% required properties for solution

dPhyduMonotonyPoints = [0, ...
                        0.253594729763804, ...
                        0.392699081698724, ...
                        0.614147554929360, ...
                        0.785398163397448, ...
                        0.997333214051655, ...
                        1.178097245096172, ...
                        1.38569230081213];
dPhydtMonotonyPoints = [0, ...
                        0.134609248288975, ...
                        0.253594729763804, ...
                        0.455449645928175, ...
                        0.614147554929360, ...
                        0.822291716590292, ...
                        0.997333214051655, ...
                        1.20369504291218, ...
                        1.38569230081213];

[N, t, ~] = dichotomyMethod(q, t_0, t_N, dPhyduMonotonyPoints);

k        = 2;
eps_k    = E / N;
m        = 70;       % size of mesh on interval [t_i, t_{i+1}], i = 0, ... , N-1
m_int    = 70000;    % size of auxiliary mesh on interval [z_j, z_{j+1}], j = 0, ..., m-1
iter_num = 30;       % overstated value of the possible iterations number

outputDichotomyResultsToConsole(N, eps_k);
outputFragmentationDetails(m, m_int);
outputDichotomyResultsToTexFile(file_id, N, eps_k);

globalUExact = [];
globalTime   = [];

globalUCurr = [globalUExact uExact(t_0)];
globalTime  = [globalTime t_0];

errorArray       = zeros(1, N + 1);
intErrorArray    = zeros(1, N + 1);
majorantArray    = zeros(1, N + 1);
optMajorantArray = zeros(1, N + 1);

fprintf('\n3. Iterative cycle:\n');

while t(k-1) < t_N

    fprintf('\nInterval #%i', k-1);

    if k == 11
        fprintf('Ops!');
    end
    % initial data for function and time grid creation
```

```
  a = t(k-1);
  b = t(k);

  s = 1;

  % construct mesh for integration and interpolation
  t_tau = linspace(a, b, m);
  t_int_tau = linspace(a, b, (m - 1) * (m_int - 1) + 1);

  u_curr_tau   = zeros(1, m);

  % projection of the exact solution on the mesh
  if k == 2
      u_start = @(t)(uStart(t));
  else
      u_start = inline(sym(u_0));
  end

  u_prev_tau(1, :) = u_start(t_tau);
  u_prev_int_tau(1, :) = u_start(t_int_tau);
  phi_prev_int_tau = Phi(u_prev_int_tau, t_int_tau);

  % projection of the exact solution on the mesh
u_exact_tau = uExact(t_tau);


  %profile on
  L1 = getLipschitzConstantL1(symd2Phidudt, t_int_tau, dPhyduMonotonyPoints);

  % initialization of data
  er           = max(abs(u_prev_tau - u_exact_tau));
  Error        = zeros(1, iter_num);
  Majorant     = zeros(1, iter_num);
  Est_e_interp = zeros(1, iter_num);
  Est_e_integr = zeros(1, iter_num);
  D            = zeros(1, iter_num);

  solution   = struct('u', 0);
  right_part = struct('phi', 0);

  while er > eps_k

  [L2, l] = getLipschitzConstantL2(u_prev_int_tau, t_int_tau, dPhydtMonotonyPoints);
      L = L1 + getMultiplicationResult(L2, l);

      % 1. calculate new function by numerical integration
      [u_curr_tau, Est_e_integr(s)] = numericalIntegral(t_tau, t_int_tau,
          phi_prev_int_tau, u_0, m_int, L);

      % 2. calculate middle points with help of interpolation with
      % spline using function values that have been obtained on previous step
      u_curr_int_tau = getPiecewiseLinearInterp(t_tau, t_int_tau, u_curr_tau, m_int);

      % 3. calculate next step right part function values
      %phi_curr_tau     = phi_inline(t_tau,     u_curr_tau);
      phi_curr_int_tau = Phi(u_curr_int_tau, t_int_tau);
```

```
    % 4. computation of error and majorant
    [D, Error, Majorant, Est_e_interp, Est_e_integr, er] = ...
        calculateErrorMajorant(D, Error, Majorant, Est_e_interp, Est_e_integr, ...
                               s, u_prev_tau, u_curr_tau, phi_prev_int_tau, t_int_tau,
                                    L, q, u_exact_tau);
    % output error and majorant to console
    fprintf('\ter = %1.4e\n\teps_k = %1.4e\n', Error(s), eps_k);

    outputErrorMajorantToConsole(Error, Majorant, s);

    u_prev_tau     = u_curr_tau;
    u_prev_int_tau = u_curr_int_tau;
    phi_prev_int_tau = phi_curr_int_tau;

    solution(s).u     = u_curr_tau;
    right_part(s).phi = phi_curr_int_tau;

    s = s + 1;
end

error_size = s - 1;

% calculate additional solution for sequence %of the approximate solutions
% to find opimized upper bound for the error of approximate solution on the previous
    step
[L2, l] = getLipschitzConstantL2(u_prev_int_tau, t_int_tau, dPhydtMonotonyPoints);
L = L1 + getMultiplicationResult(L2, l);

[u_add_tau, Est_e_integr(s)] = numericalIntegral(t_tau, t_int_tau, phi_prev_int_tau,
    u_0, m_int, L);
solution(s).u     = u_add_tau;

% calculate optimized majorant
[optMajorant] = getOptimizedMajorant(error_size, t_int_tau, solution, right_part, L, q
    , Est_e_integr, s);

errorArray(k)       =                         Error(error_size);
intErrorArray(k)    = intErrorArray(k-1)    + Error(error_size);
majorantArray(k)    = majorantArray(k-1)    + Majorant(error_size);
optMajorantArray(k) = optMajorantArray(k-1) + optMajorant(error_size);

one = ones(1, length(t_tau));
if k == 2
    globalTime = t_tau;
    globalUCurr = u_curr_tau;
    globalUExact = u_exact_tau;

    globalError =  one * Error(error_size);
    globalMajorant = one * Majorant(error_size);
    globalOptMajorant = one * optMajorant(error_size);
    globalInterpEst(k - 1) = (q / (1 - q)) * Est_e_interp(error_size);
    globalIntegrEst(k - 1) = (q / (1 - q)) * Est_e_integr(error_size);
    globalOstrEst(k - 1)   = (q / (1 - q)) * D(error_size);

else
```

```
        additionError = getAddition(globalError(length(globalError)), globalError(length(
            globalError)) + Error(error_size), length(t_tau));
        additionMajorant = getAddition(globalMajorant(length(globalMajorant)),
            globalMajorant(length(globalMajorant)) + Majorant(error_size), length(t_tau));
        additionOptMajorant = getAddition(globalOptMajorant(length(globalOptMajorant)),
            globalOptMajorant(length(globalOptMajorant)) + optMajorant(error_size), length
            (t_tau));

        globalTime = [globalTime(1: length(globalTime) - 1) t_tau];
        globalUCurr = [globalUCurr(1: length(globalUCurr) - 1) u_curr_tau];
        globalUExact = [globalUExact(1: length(globalUExact) - 1) u_exact_tau];

        globalError       = [globalError(1: length(globalError) - 1) additionError];
        globalMajorant    = [globalMajorant(1: length(globalMajorant) - 1)
            additionMajorant];
        globalOptMajorant = [globalOptMajorant(1: length(globalOptMajorant) - 1)
            additionOptMajorant];

        globalInterpEst(k - 1) = globalInterpEst(k - 2) + (q / (1 - q)) * Est_e_interp(
            error_size);
        globalIntegrEst(k - 1) = globalIntegrEst(k - 2) + (q / (1 - q)) * Est_e_integr(
            error_size);
        globalOstrEst(k - 1)   = globalOstrEst(k - 2) + (q / (1 - q)) * D(error_size);
    end

    % output in tex-file table with error and different estimates
    outputIterativeErrorOptMajMajToTexFile(file_id, error_size, Error, optMajorant,
        Majorant)

    Solution(k-1).f = u_curr_tau;
    u_0             = u_curr_tau(length(u_curr_tau));

    k = k + 1;
end

outputPartitionsOfErrors(t, globalInterpEst, globalIntegrEst, globalOstrEst,
    example_number);
fprintf('\n 4. Result presentation\n');

% output in tex-file table with global error, global majorant and
% optimazed form of global majorant
outputGlobalErrorOptMajMajToTexFile(file_id, errorArray, optMajorantArray, majorantArray);
plotSolutionWithErrorBars(Solution, t, m, errorArray, example_number, file_id);
plotErrorDiagramForArticle(t, intErrorArray, optMajorantArray, majorantArray,
    example_number, file_id);
plotExactSolutionWithIntegralErrorMajorantForArticle(Solution, m, t, intErrorArray,
    optMajorantArray, majorantArray, example_number, file_id);
plotResultForArticle(globalTime, globalUCurr, globalUExact, globalError, globalOptMajorant
    , globalMajorant, example_number, file_id);

end
```

We first define sets of the zeros of the function $\frac{d\varphi}{du}$ and $\frac{d\varphi}{dt}$, which are used in numerical norm calculation (this information should be provided with the problem). Then, we submit the function `dichotomyMethod` to the obtained time-mesh, which will be used in the iteration process. We apply the integration operator on every time interval of the mesh obtained by `dichotomyMethod`.

Listing A.22: Function to obtain time-mesh for applying the Picard-Lindelöf method.

```
function [N, t, arrayL] = dichotomyMethod(q, t_0, t_N, monotonyPoints)
%
% dichotomyMethod
% This function generates adaptive time mesh according to dichotomy method
%
% SYNTAX:  [N, t, arrayL] = dichotomyMethod(q, t_0, t_N, monotonyPoints)
% INPUT:   q                required contractivity level constant
%          t_0              lower bound of time interval
%          t_N              upper bound of time interval
%          monotonyPoints   monotony intervals on fucntion dPhidu
% OUTPUT:  N                number of time interval
%          t                adaptive time mesh
%          arrayL           Lipschitz constants on every time mesh interval

% define structure linkedList
listObj = linkedList([t_0 t_N]);
index = 1;
arrayL = [];

% continue iteration process till reaching the last node of time mesh
while listObj.getItem(index) ~= t_N
    a     = listObj.getItem(index);
    b     = listObj.getItem(index + 1);
    L     = numericNorm(@(t)(absdPhidu(t)), a, b, monotonyPoints);
    kappa = L * (b - a);
    if kappa > q
        c     = (a + b) / 2;
        listObj.addAfter(c, index);
    else
        arrayL(index) = L;
        index = index + 1;
    end
end
t = listObj.getList();
N = length(t) - 1;

end
```

The algorithm is already formalized in Algorithm 7. We decrease the time interval until the moment it satisfies the q-contractivity condition (4.8) for the operator (4.7).

Next, we submit the cycle over every subinterval of the obtained time-mesh. Estimation of the Lipschitz constants L_1 and L_2 is the most time consuming procedure. On every step of this cycle, we consider the time interval $[t_i, t_{i+1}]$ as an independent unit. In order to interpolate and integrate on the i-th interval, we define the refined time mesh t_τ and t_τ^{int}, respectfully. After preparing vectors for approximation u and φ we submit the Picard-Lindelöf cycle. It contains the procedure of the numerical integration function φ and construction of the function u on the mesh t_τ (function `numericalIntegral`, see Listing A.23) and interpolation of the function u to the refined mesh t_τ^{int} (function `getPiecewiseLinearInterp`, see Listing A.24).

Listing A.23: Function to integrate φ and obtain function u on the mesh t_τ.

```
function [u_tau, est] = numericalIntegral(t_tau, t_int_tau, phi_int_tau, u_0, m_int, L)
%
% numericalIntegral
% This function calculates numerical integral using trapezoid quadrature
% and constructs estimate of integration error
%
% SYNTAX:    [u_tau, est] = numericalIntegral(t_tau, t_int_tau, phi_int_tau, u_0, m_int, L)
% INPUT:     t_tau           time mesh
%            t_int_tau       refined time mesh for integration
%            phi_int_tau     projection of function Phi on refined time mesh
%            u_0             Dirichlet boundary condition
%            m_int           number of integation points
%            L               Lipschitz constant
% OUTPUT:    u_tau           approximate function on time mesh t_tau
%            est             integration error estimate

m = length(t_tau);

% set storage for new approximate solution
u_tau = zeros(1, m);
% define step of integration refined mesh
h_int = t_int_tau(2) - t_int_tau(1);
% initialize start value
u_tau(1) = u_0;
est = 0;

for i = 1 : m - 1
    % remember start and end indexes
    i_start = (i - 1)*(m_int - 1) + 1;
    i_end = i*(m_int - 1) + 1;

    % define set of function values
    f = phi_int_tau(i_start : i_end);
    % define time-mesh
    t = t_int_tau(i_start : i_end);
    % get Lipschitz constant
    lipschitzConstant = L(i_start : i_end-1);

    % calculate next value of the approximate solution
    u_tau(i + 1) = trapezoidQuadrature(f, h_int) + u_0;
    % calculate integration error
    est = est + getTrapezoidQuadratureEstimate(f, t, lipschitzConstant);

    % init next step initial value
    u_0 = u_tau(i + 1);
end
end
```

```
function int = trapezoidQuadrature(f, h)
n = length(f);
int = h*(f(1) + f(n) + sum(f(2 : n - 1)));
end

function est = getTrapezoidQuadratureEstimate(f, t, L)
est = 0;
for i = 1 : length(t) - 1
  est = est + abs(L(i)*(t(i + 1) - t(i))^2 - (f(i + 1) - f(i))^2/L(i))/4  ;
end
end

function Int = threeByEightInt(x, y)
h = x(4) - x(1);
if length(y) == 4
  Int = (h/8) * (y(1) + 3*(y(2) + y(3)) + y(4));
else
  firstSum = 0;
  secondSum = 0;
  i = 2;
  while i <= length(y) - 2
    firstSum = firstSum + y(i) + y(i + 1);
    i = i + 3;
  end
  i = 4;
  while i <= length(y) - 3
    secondSum = secondSum + y(i);
    i = i + 3;
  end
  Int = (h/8)*(y(1) + y(length(y)) + 3*firstSum + 2*secondSum);
end
end
```

Listing A.24: Function to obtain u interpolated to the refined mesh t_{τ}^{int}.

```
function u_int_tau = getPiecewiseLinearInterp(t_tau, t_int_tau, u_tau, m_int)

% getPiecewiseLinearInterp
% This function interpolates input approximate function by  piecewise linear functions
%
% SYNTAX:   u_int_tau = getPiecewiseLinearInterp(c, t_int_tau, u_tau, m_int)
% INPUT:    t_tau       time mesh
%           t_int_tau   refined time mesh
%           u_tau       approximate solution projected on the mesh t_tau
%           m_int       number of points on every interval [t_tau(i)
%                       t_tau(i+1)], i = 0, N-1
% OUTPUT:   u_int_tau

% define storage for interpolated solution
u_int_tau = zeros(1, length(t_int_tau));
% init solution in the first node of the mesh
u_int_tau(1) = u_tau(1);

% submit the cycle to interpolate u_tau on every time interval [t_tau(i),
% t_tau(i+1)], i = 1, ..., n
for i = 1 : length(t_tau) - 1
```

```
    k = (u_tau(i + 1) - u_tau(i))/(t_tau(i + 1) - t_tau(i));
    b = (u_tau(i) * t_tau(i + 1) - u_tau(i + 1) * t_tau(i))/(t_tau(i + 1) - t_tau(i));

    % extend function u_tau on the interval [t_tau(i), t_tau(i+1)], i = 1,
    % ..., n by linear function y_i = x_i * k + b
    for j = 2 : m_int
        index = (i - 1) * (m_int - 1) + j;
        u_int_tau(index) = t_int_tau(index) * k + b;
    end
end

end
```

Finally, we can obtain the error bounds after construction of a new approximation. This is provided by the function `calculateErrorMajorant` from Listing A.24.

Listing A.25: Function to obtain error bounds.

```
function [D, Error, Majorant, Est_e_interp, Est_e_integr, e] = ...
            calculateErrorMajorant(D, Error, Majorant, Est_e_interp, Est_e_integr, s, ...
            u_prev_tau, u_curr_tau, phi_prev_int_tau, t_int_tau, L, q, u_exact_tau)

% calculateErrorMajorant
% This function constructs error bounds for obtained approximation
%
% SYNTAX:   [D, Error, Majorant, Est_e_interp, Est_e_integr, e] = ...
%               calculateErrorMajorant(D, Error, Majorant, Est_e_interp, Est_e_integr, ...
%               s, u_prev_tau, u_curr_tau, phi_prev_int_tau, t_int_tau, L, q, u_exact_tau)
% INPUT:    D                   array of differences between the exact solution
%                               and approximations
%           Error               array of true errors for approximations
%           Majorant            array of majorant for Error array
%           Est_e_interp        interpolation error bound
%           Est_e_integr        integration error bound
%           s                   current iteration
%           u_prev_tau          approximate solution on the previous step
%           u_curr_tau          approximate solution on the current step
%           phi_prev_int_tau    function phi projected on refined time-mesh
%           t_int_tau           refined time-mesh
%           L                   Lipschitz constant of function Phi
%           q                   contractivity parameter
%           u_exact_tau         exact solution projected on time-mesh
% OUTPUT:   D                   updated for the current iteration differences between
%                               the exact solution and approximations
%           Error               updated for the current iteration array of true errors
%                               for approximations
%           Majorant            updated for the current iteration array of majorant
%                               for Error array
%           Est_e_interp        updated for the current iteration interpolation
%                               error bound
%           Est_e_integr        updated for the current iteration integration
%                               error bound
%           s                   current iteration

% get the maximal absolute value of difference, continous norm for error
Error(s) = max(abs(u_curr_tau - u_exact_tau));
```

```
% get partition of the error estimate
D(s) = max(abs(u_prev_tau - u_curr_tau));
% get interpolation error estimate
Est_e_interp(s) = getInterpolationErrorEstimate(t_int_tau, phi_prev_int_tau, L);
% get the general error estimate
Majorant(s) = (q / (1 - q)) * (D(s) + Est_e_integr(s) + Est_e_interp(s));
e = Error(s);

end
```

We use the procedure `getOptimizedMajorant` to obtain the advanced form of the majorant (see Listing A.26).

Listing A.26: Function to obtain advanced error bounds.

```
function optMajorant = getOptimizedMajorant(error_size, t_int_tau, solution, right_part, L
    , q, Est_e_integr, s)
%
% getOptimizedMajorant
% This function calculates the optimized error estimate based on additional
% terms of the sequence
%
% SYNTAX:   optMajorant = getOptimizedMajorant(error_size, t_int_tau, solution, right_part
    , L, q, Est_e_integr, s)
% INPUT:    error_size      size of error
%           t_int_tau       refined time mesh
%           solution        approximate solution storage
%           right_part      right part solution storage
%           L               Lipschitz constant of function Phi
%           q               contractivity constant
%           Est_e_integr    integration error estimate
%           s               current step
% OUTPUT:   optMajorant     optimized error majorant

% define storage for estimate
optMajorant = zeros(1, error_size);

for l = 1 : error_size
    % get partition of error estimate, interpolation error estimate
    e_int_est = getInterpolationErrorEstimate(t_int_tau, right_part(l).phi, L);
    % get optimal error estimate
    optMajorant(l) = (1/(1 - q^(error_size + 1 - l)))*(max(abs(solution(l).u - solution(
        error_size + 1).u)) + e_int_est + Est_e_integr(s));
end

end
```

At the end of every iteration we save the obtained results to the global storages. This results will be written to the result graphics and files.

B References

[1] G. Auchmuty. *A posteriori error estimates for linear equations.* Numer. Math., 61(1992), p. 1 – 6.

[2] E. A. Coddington, N. Levinson, *Theory of Ordinary Differential Equations*, McGraw-Hill, New York, 1972.

[3] K. Eriksson, D. Estep, P. Hansbo, C. Johnson, *Computational Differential Equations*, Cambridge, Univesity press, 1996.

[4] T. Eirola, A. M. Krasnosel'skii, M. A. Krasnosel'skii, N. A. Kuznetsov, O. Nevanlinna, *Incomplete corrections in nonlinear problems*, Elsevie, Nonlinear Analysis, Vol. 25, Num. 7, 1995, p. 717 – 728.

[5] E. Lindelöf, *Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre.* Comptes rendus hebdomadaires des séances de l'Académie des sciences, Vol. 114, 1894, p. 454 – 457.

[6] A. N. Kolmogorov, A. N. Fomin, *Elements of the theory of functions and functional analysis, Vol. 1, Metric and Normed Spaces.* Publishing house "Nauka", Moscow, 1976.

[7] P. Neittaanmäki, S. Repin, *Reliable methods for computer simulation. Error control and a posteriori estimates*, Elsevier, New York, 2004.

[8] O. Nevanlinna, *A note on Picard–Lindelöf iteration*, Numerical Methods for Ordinary Differential Equations, Lecture Notes in Mathematics, Vol. 138, 1989, Springer, Berlin.

[9] O. Nevanlinna, *Power bounded prolongations and Picard–Lindelöf iteration*, Numerische Mathematik, 1990, Vol. 58, Num. 1, p. 479 – 501.

[10] O. Nevanlinna, *Linear acceleration of Picard–Lindelöf iteration*, Numerische Mathematik, 2005, Vol. 57, Num. 1, p. 147 – 156.

[11] O. Nevanlinna, *Remarks on Picard–Lindelöf iteration Part I*, BIT Numerical Mathematics, 1989, Vol. 29, Num. 2, p. 328 – 346.

[12] O. Nevanlinna, *Remarks on Picard–Lindelöf iteration Part I*, BIT Numerical Mathematics, 1989, Vol. 29, Num. 3, p. 535 – 562.

[13] A. Ostrowski, *Les estimations des erreurs a posteriori dans les procédés itératifs*, C.R. Acad. Sci, Paris Sér. A - B, 275(1972). p. A275-A278.

[14] S. Repin, *A Posteriori error estimates for partial differential equations*, Walter de Gruyter, Berlin, NewYork, 2008.

[15] A. Samarski, A. Gulin. *Numerical methods.* Moscow 'Nauka', 1989.