NUMERICAL SOFTWARE

Vít Dolejší

Charles University Prague
Faculty of Mathematics and Physics
Czech Republic
dolejsi@karlin.mff.cuni.cz

Prague December 12, 2025

Preface

These lecture notes more or less cover the lecture Numerical software given by the author at the master program at the Charles University, Prague, the Faculty of Mathematics and Physics. They should serve as a survey of the lecture without a mathematically rigorous derivation and without explaining all details. Most ideas are explained by some examples. The important part of the lecture notes are Exercises included at the end of each section.

The aim of the lecture:

- this lecture fill a gap in lectures given at our faculty
- implementation of numerical methods is also a non-trivial task
- we need efficiency, accuracy and robustness
- important aspect is an adaptation
- it is advantageous to use software libraries (subroutines written in Fortran, C++, etc.)
- we learn
  - to understand the basic principles of numerical software
  - to use public software for basic tasks
  - to employ public software for your own project
- a practical introduction to more advanced numerical methods
- role of exercises: students have to solve several Exercises and main tasks

Contents

Part I

Numerical Software 1 (Winter semester)

Chapter 1

Introduction

1.1 Numerical simulation of real-world problems

Process of numerical simulation of real-world problems can be split into several steps:

- physical problem – a real problem which we want to simulate numerically,
- mathematical model – a mathematical description of the physical problem with the set of (differential) equations including boundary (and initial) conditions,
- approximate problem – finite dimensional approximation of the mathematical problem with the aid of a suitable numerical method,
- computer implementation – practical computer realization of the numerical method.

1.1.1 Physical problem

Let $\mathscr{P}_{\text{phys}}$ formally denote the considered physical (chemical, biological, etc.) problem, we seek its solution u_{phys} formally given formally by

$$\mathscr{P}_{\text{phys}}(d_{\text{phys}}; u_{\text{phys}}) = 0, \tag{1.1}$$

where d_{phys} is the set of data of the problem. Let us note that (1.1) described a real problem only formally, $\mathscr{P}_{\text{phys}}$ is not any standard mapping.

In some situation, we know u_{phys} approximately due to physical experiments. The experimental results suffer from errors of measures. Moreover, experimental measurement are expensive and in some cases impossible (e.g., in medicine).

1.1.2 Mathematical model

The problem (1.1) can be mathematically described by a (mathematical or also abstract) model in the form

$$\mathscr{P}(d; u) = 0, \tag{1.2}$$

where $u \in X$ is the unknown exact solution of problem (1.2), $d \in Z$ represents the data of the problem (boundary and initial conditions, source terms, etc.), $\mathscr{P} : Z \times X \to Y$ is a given mapping and X, Y and Z are normed vector spaces. Usually, space X, Y and Z have an infinite dimension. The difference $u - u_{\text{phys}}$ is called the model error. The problem $\mathscr{P}$ is, e.g., a system of differential equation, an integral equation, a system of algebraic equations, etc.

In order to set the model, we have to balance to aspects

(A1) the model $\mathscr{P}$ should approximate $\mathscr{P}_{\text{phys}}$ accurately, i.e., the model error have to be small,

(A2) the model $\mathscr{P}$ should be simple such that we are able to analyse and solve it.

1.1.3 Approximate problem

Usually, the requirement (A1) leads to a model, whose exact solution is impossible, hence we have to solve it approximately. Therefore, we define the approximate (or discrete) problem by

$$\mathscr{P}_h(d_h; u_h) = 0, \tag{1.3}$$

where $u_h \in X_h$ is the approximate solution, $d_h \in Z_h$ is the discrete analogue of d, $\mathscr{P}_h : Z_h \times X_h \to Y_h$ is a given mapping representing a numerical method and X_h, Y_h and Z_h are normed vector spaces having finite dimension. The symbol h formally denotes all parameters of the discretization and if $h \to 0$ than the number of degrees of freedom (DOF) $(= \dim X_h)$ goes to the infinity.

Remark 1.1. *In the finite volume/element method, the symbol h denotes the maximal size of the mesh elements. Generally, h corresponds to all possible parameters of the approximation of the model problem (1.2), e.g., size of the elements of partitions, degrees of polynomial approximation, etc.*

The relation (1.3) represents a numerical method for the approximate solution of the mathematical model (1.2). Let us note that $\mathscr{P}_h$ means a finite sequence of mathematical operations resulting the approximate solution $u_h \in X_h$. The difference $u - u_h$ is called the discretization error, for complicated problems, the discretization error can have several contributions, e.g., an quadrature error, an algebraic error, etc.

Example 1.2. *Let $\mathscr{P}$ be a nonlinear differential equation considered in $\Omega := (0, 1)$*

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(a(u)\frac{\mathrm{d}u}{\mathrm{d}x}\right) = f, \qquad u(0) = u(1) = 0 \tag{1.4}$$

where $a(u) > 0$, $u \in \mathbb{R}$ and $f : \Omega \to \mathbb{R}$ are given. In order to introduce the discrete problem (1.3), we can proceed for example, in the following way:

1. *The equation (1.4) is discretized by the finite element method, where Ω is split onto finite mutually disjoint elements $K \in \mathscr{T}_h$,[1] then the space X_h is the space of piecewise polynomial functions over $K \in \mathscr{T}_h$. Then the approximate solution u_h is sought in the form $u_h = \sum_{i=1}^{N_h} u^i \phi_i$, where u^i, $i = 1, \dots, N_h$ are the unknown coefficients, ϕ_i, $i = 1, \dots, N_h$ are basis functions of X_h and $N_h = \dim X_h$.*

2. *The unknown coefficients u^i, $i = 1, \dots, N_h$ are given by the system of the nonlinear algebraic equations, whose entries are evaluated be the numerical quadrature.*

3. *The system of the nonlinear algebraic equations is solved by the Newton method.*

4. *The linear algebraic system arising in the Newton method are solved by the iterative GMRES solver.*

5. *Rounding errors caused by the inexact arithmetic, see Chapter 3.*

Hence, $\mathscr{P}_h$ represents all steps 1. – 4. In many papers the term discretization error means only the error followed from step 1, the error followed from step 2. is called the quadrature error and the errors followed from steps 3. and 4. are called the algebraic errors.

We expect that readers are familiar with basic numerical methods, e.g.,

type of model problem	numerical method
integrals of a real function	numerical quadrature (Newton-Cotes, Gauss)
linear algebraic systems	direct methods, iterative methods (Jacobi, CG, GMRES)
nonlinear algebraic systems	iterative Newton method
ordinary differential equations	Runge-Kutta, multi-step methods
partial differential equations	finite difference method
	finite element method
	finite volume method

Obviously, it is reasonable to expect that $u_h \to u$ as $h \to 0$, i.e., the numerical method (1.3) converges (for more precise definition of the convergence see Definition 1.18). In order to ensure the convergence of the numerical method (1.3), we require that the family of spaces $\{X_h\}_{h\in(0,h_0)}$ has approximation properties of X, i.e.,

$$\forall v \in X \; \exists \{v_h\}_{h\in(0,h_0)}, \; v_h \in X_h : \; \|v_h - v\| \to 0 \text{ for } h \to 0, \tag{1.5}$$

where the symbol h_0 formally denotes some maximal parameter of the discretization. Similarly, we require that the family of spaces $\{Y_h\}_{h\in(0,h_0)}$ and $\{Z_h\}_{h\in(0,h_0)}$ have approximation properties of Y and Z, respectively.

[1] We call the partition $\mathscr{T}_h$ the mesh and $K \in \mathscr{T}_h$ the mesh elements.

1.1.4 Computer implementation

The discrete problem (1.3) is finite-dimensional which means that the approximate solution u_h is sought in X_h, $\dim X_h < \infty$. In practice, problem (1.3) is solved by a computer. Computation in the range of the real numbers $\mathbb{R}$ on any computer using any software (fortran, C, C++, ..., Matlab, Maple, Mathematica, ..., Excel, ...) suffer from the rounding errors since each software has to use a finite precision arithmetic, see Section 3. E.g., the number π has an infinite series and the memory of any computer is limited. In some cases, the rounding errors are negligible but we have to take them into account generally.

Therefore, the solution of (1.3) in a finite precision arithmetic gives the solution $u_h^* \in X_h$, given formally by

$$\mathscr{P}_h^*(d_h^*; u_h^*) = 0, \tag{1.6}$$

where $\mathscr{P}_h^*$ and d_h^* are the analogous of $\mathscr{P}_h$ and d_h in the finite precision arithmetic, respectively. The difference $u_h - u_h^*$ is called the rounding error.

The solution u_h^* is the only one solution which is available in practice. Sometimes, we can compare u_h^* with experimental data.

1.1.5 Types of errors

The difference between the desired (unknown) solution u_{phys} and the only available solution u_h^* is called the total error. Using the terms introduced in previous Sections, we have

$$\underbrace{u_{\text{phys}} - u_h^*}_{\text{total error}} = \underbrace{u_{\text{phys}} - u}_{\text{model error}} + \underbrace{\underbrace{u - u_h}_{\text{discretization error}} + \underbrace{u_h - u_h^*}_{\text{rounding error}}}_{\text{computational error}}.$$

Within this lecture notes we deal mostly with the computational error.

Usually, the errors are considered as the norms of the appropriate difference, moreover, we distinguish the absolute and relative errors. Hence, the absolute computational error is given by

$$E_{\text{abs}}^{\text{comp}} = \|u - u_h^*\|$$

and the relative computational error is given (for $u \neq 0$) by

$$E_{\text{rel}}^{\text{comp}} = \frac{\|u - u_h^*\|}{\|u\|},$$

etc. for other types of errors.

1.2 Basic terms

In this section, we introduce general terms of the model problem (1.2) and the approximate problem (1.3).

1.2.1 Model problem

Definition 1.3. *Let us assume that problem (1.2) has a unique solution for all admissible data d. We say that solution u of (1.2) depends continuously on the data d if a "small" perturbation of the data d gives a "small" perturbation of the solution δu. Particularly, let δd denote an admissible perturbation of d and δu the corresponding perturbation of the solution, i.e.,*

$$\mathscr{P}(d + \delta d; u + \delta u) = 0. \tag{1.7}$$

Then

$$\forall \eta > 0 \ \exists K(\eta, d) : \ \|\delta d\| \leq \eta \ \Rightarrow \ \|\delta u\| \leq K(\eta, d)\|\delta d\|.$$

The norms used for the data and for the solution may be different.

Definition 1.4. *We say that problem (1.2) is well-posed (or stable) if it admits a unique solution u which depends continuously on the data d.*

Remark 1.5. *If the problem is not well-posed then we say that the problem is ill-posed (or unstable). Sometimes, instead of the stability of the problem we speak about the sensitivity of the problem.*

Remark 1.6. *If the problem is well-posed, then it stands a good chance of solution on a computer using a stable numerical method, see Definition 1.13. If the model problem is not well-posed, it needs to be re-formulated for numerical treatment. Typically this involves including additional assumptions, such as smoothness of solution. This process is known as regularization.*

Example 1.7. *A typical example of an ill-posed problem is finding the number of real roots of polynomial, e.g.,*

$$p(x) = x^4 - x^2(2a - a) + a(a - 1), \quad a \in \mathbb{R} \text{ is a parameter.} \tag{1.8}$$

For $a \geq 1$ we have 4 real roots, for $a \in ([0, 1)$ we have two real roots and for $a < 0$ no real root. Therefore, an inaccuracy in the parameter a (caused, e.g., by a physical measurement) can lead to a qualitatively different solution.

In order to measure qualitatively the dependence of the solution on the data we define the following.

Definition 1.8. *Let us consider problem (1.2). We define the relative condition number by*

$$K(d) = \sup_{\delta d \in D} \frac{\frac{\|\delta u\|}{\|u\|}}{\frac{\|\delta d\|}{\|d\|}}, \tag{1.9}$$

where D is a neighbourhood of the origin and denotes the set of admissible perturbations of data d for which the perturbed problem (1.7) makes sense. If $u = 0$ or $d = 0$ then we have to use the absolute condition number

$$K_{\text{abs}}(d) = \sup_{\delta d \in D} \frac{\|\delta u\|}{\|\delta d\|}. \tag{1.10}$$

If $K(d)$ is "small" we say that problem (1.2) is well-conditioned otherwise we say that problem (1.2) is ill-conditioned. The meaning of "big" and "small" depends on the considered problem.

Remark 1.9. *The condition number is independent of a numerical method.*

Example 1.10. *Let us consider a linear algebraic system $Ax = b$, where $A \in R^{n\times n}$ is a square regular matrix, $b \in R^n$ is the given vector and $x \in R^n$ is unknown. Let δb be a perturbation of b and δx the corresponding perturbation of the solution. Then*

$$Ax = b, \quad A(x + \delta x) = b + \delta b \;\Rightarrow\; A\,\delta x = \delta b \;\Rightarrow\; \delta x = A^{-1}\delta b. \tag{1.11}$$

Let $\|A\|$ be an induced norm of A (see [Wat02]) then $\|Ay\| \le \|A\|\,\|y\|$ and we obtain

$$K(d) = \sup_{\delta b \in D} \frac{\|\delta x\|}{\|x\|}\frac{\|b\|}{\|\delta b\|} = \sup_{\delta b \in D} \frac{\|A^{-1}\delta b\|}{\|x\|}\frac{\|Ax\|}{\|\delta b\|} \le \sup_{\delta b \in D} \frac{\|A^{-1}\|\|\delta b\|}{\|x\|}\frac{\|A\|\|x\|}{\|\delta b\|} = \|A\|\,\|A^{-1}\|.$$

Example 1.11. *Let us consider*

$$A = \begin{pmatrix} 1000 & 999 \\ 999 & 998 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \tilde{b} = \begin{pmatrix} 1.001 \\ 0.999 \end{pmatrix}.$$

We can verify that the solutions x and $\tilde{x}$ of the problems $Ax = b$ and $A\tilde{x} = \tilde{b}$ are

$$x = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \qquad \tilde{x} = \begin{pmatrix} -0.997 \\ 0.999 \end{pmatrix},$$

respectively. Therefore, the perturbation of the right-hand side $\delta b = b - \tilde{b} \approx 10^{-3}$ (e.g., by a physical measurement) causes the perturbation of the solution $\delta x = x - \tilde{x} \approx 2$ (= 200%). We can verify that the condition number is approximately 10^6.

1.2.2 Approximate problem

Obviously, we expect that $u_h \to u$ as $h \to 0$. For that, it is necessary that $d_h \to d$ and that $\mathscr{P}_h$ "approximates" $\mathscr{P}$, as $h \to 0$ in the following manner.

Definition 1.12. *Let $d \in Z$ be an admissible datum of $\mathscr{P}_h$, $h \in (0, h_0)$, we say that the approximate problem (or the numerical method) (1.3) is consistent if*

$$\mathscr{P}_h(d; u) = \mathscr{P}_h(d; u) - \mathscr{P}(d; u) \to 0 \text{ for } h \to 0,$$

where u is the exact solution of problem (1.2) corresponding to the datum d.

Definition 1.4 defines the stability of the model problem (1.2). Similarly, we can introduce the stability of the approximate problem.

Definition 1.13. *We say that the approximate problem (1.3) (or the numerical method) is well-posed (or stable) if it has a unique solution u_h for the given datum d_h and that u_h depends continuously on the data, i.e.,*

$$\forall \eta > 0 \ \exists K_h(\eta, d_h): \ \|\delta d_h\| \le \eta \ \Rightarrow \ \|\delta u_h\| \le K_h(\eta, d_h)\|\delta d_h\|.$$

Otherwise, we say that the approximate problem is ill-posed (or unstable).

Remark 1.14. *The property "stability of the numerical method" is fundamental for a practical use the numerical method. The stability ensures that the rounding errors (caused by the finite precision arithmetic) do not cause a failure of the algorithm, see Example 1.15.*

Example 1.15. *Let us consider the following initial value problem: we seek a function $y(t): \mathbb{R}^+ \to \mathbb{R}$ such that*

$$y' = -100y + 100t + 101, \qquad y(0) = 1. \tag{1.12}$$

We can easily verify that $y(t) = 1 + t$. We solve (1.12) by the Euler method, i.e., let $h > 0$ be the time step, $t_k = k\,h$, $k = 0, 1, \ldots$ the nodes of the partition, then the Euler method gives the approximations $y_k \approx y(t_k)$, $k = 0, 1, \ldots$ by the formula

$$\begin{aligned} y_0 &= 1, \\ y_{k+1} &= y_k + h\,(-100y_k + 100t_k + 101), \quad k = 0, 1, \ldots. \end{aligned} \tag{1.13}$$

Let us put $h = 0.1$, we can derive that

$$\begin{aligned} y_1 &= 1.0 + 0.1 \cdot (-100 \cdot 1.0 + 100 \cdot 0.0 + 101) = 1.1 \\ y_2 &= 1.1 + 0.1 \cdot (-100 \cdot 1.1 + 100 \cdot 0.1 + 101) = 1.2 \\ y_3 &= 1.2 + 0.1 \cdot (-100 \cdot 1.2 + 100 \cdot 0.2 + 101) = 1.3 \\ y_4 &= 1.3 + 0.1 \cdot (-100 \cdot 1.3 + 100 \cdot 0.3 + 101) = 1.4 \\ &\vdots \end{aligned}$$

We conclude that $y_k = y(t_k)$, $k = 0, 1, \ldots$, i.e. the discretization error is equal to zero. However, the simple fortran subroutine

```
      y0 = 1.D+00
      h = 0.1
      k = 0
      write(*,  '(i5, 3es14.6)' ) 0, h, 0., y0

10    continue
      y1 = y0 + h*(-100* y0 + 100 * h*k + 101)
```

```
    k = k + 1

    write(*,  '(i5, 3es14.6)' ) k, h, h*k, y1
    y0 = y1

    if(h*k < 2.) goto 10
```

gives the output

```
    0  1.000000E-01  0.000000E+00  1.000000E+00
    1  1.000000E-01  1.000000E-01  1.100000E+00
    2  1.000000E-01  2.000000E-01  1.200000E+00
    3  1.000000E-01  3.000000E-01  1.299999E+00
    4  1.000000E-01  4.000000E-01  1.400007E+00
    5  1.000000E-01  5.000000E-01  1.499938E+00
    6  1.000000E-01  6.000000E-01  1.600556E+00
    7  1.000000E-01  7.000000E-01  1.694994E+00
    8  1.000000E-01  8.000000E-01  1.845049E+00
    9  1.000000E-01  9.000000E-01  1.494555E+00
   10  1.000000E-01  1.000000E+00  5.649004E+00
   11  1.000000E-01  1.100000E+00 -3.074104E+01
   12  1.000000E-01  1.200000E+00  2.977694E+02
   13  1.000000E-01  1.300000E+00 -2.657824E+03
   14  1.000000E-01  1.400000E+00  2.394352E+04
   15  1.000000E-01  1.500000E+00 -2.154676E+05
   16  1.000000E-01  1.600000E+00  1.939234E+06
   17  1.000000E-01  1.700000E+00 -1.745308E+07
   18  1.000000E-01  1.800000E+00  1.570777E+08
   19  1.000000E-01  1.900000E+00 -1.413700E+09
   20  1.000000E-01  2.000000E+00  1.272330E+10
```

i.e., the Euler method is unstable since the approximate solution $\{y_k\}_{k\in\mathbb{N}}$ *(last column) (as well as the computational error) diverges. It is caused by the instability of the Euler method (for the time step* $h = 0.1$*) and the rounding errors. More details is given in Chapter 10.*

Example 1.16. *Let us consider the initial value problem from Example 1.15 as well as the Euler method with* $h = 0.1$*. Let us perturb the initial condition, namely* $y(0) = 1.01$*. We can found (by the exact arithmetic) that*

$$y_1 = 1.01, \quad y_2 = 2.01, \quad y_3 = -5.99, \quad y_4 = 67.0, \quad y_5 = 589.0, \quad y_6 = 5316, \quad \dots$$

Therefore, a small perturbation of the data leads a big change of the approximate solution, we speak about the propagation of the data error.

Similarly as in Definition 1.8 we define the condition number of the numerical method, namely

Definition 1.17. *For each problem* (1.3), $h \in (0, h)$, *we define the quantities*

$$K_h(d_h) = \sup_{\delta d_h \in D_h} \frac{\frac{\|\delta u_h\|}{\|u_h\|}}{\frac{\|\delta d_h\|}{\|d_h\|}}, \quad K_{\mathrm{abs},h}(d_h) = \sup_{\delta d_h \in D_h} \frac{\|\delta u_h\|}{\|\delta d_h\|}. \tag{1.14}$$

where D_h is a neighbourhood of the origin and denotes the set of admissible perturbations of data d_h for which the perturbed approximate. Moreover, define the relative asymptotic condition number and absolute asymptotic condition number of the numerical method (1.3) *by*

$$K^{\mathrm{num}}(d) = \lim_{h \to 0} \sup_{h' \in (0,h)} K_{h'}(d_{h'}) \quad \text{and} \quad K^{\mathrm{num}}_{\mathrm{abs}}(d) = \lim_{h \to 0} \sup_{h' \in (0,h)} K_{\mathrm{abs},h'}(d_{h'}). \tag{1.15}$$

If $K(d)$ is "small" we say that numerical method (1.3) *is well-conditioned of stable otherwise we say that problem* (1.2) *is ill-conditioned or unstable. The meaning of "big" and "small" depends on the considered problem.*

A necessary requirement for a practical use of any numerical method is its convergence, i.e,

Definition 1.18. *The numerical method* (1.3) *is convergent if*

$$\forall \varepsilon > 0 \, \exists \delta(h_0, \varepsilon) > 0 : \ \forall h \in (0, h_0) \, \forall \|\delta d_h\| \leq \delta(h_0, \varepsilon) \quad \Rightarrow \quad \|u(d) - u_h(d + \delta d)\| \leq \varepsilon,$$

where d is the admissible datum for the model problem (1.2), *$u(d)$ is the corresponding solution, δd is the admissible perturbation of the datum and $u_h(d + \delta d)$ is the solution of the problem* (1.3) *with the datum $d + \delta d$.*

Theorem 1.19 (Equivalence theorem (Lax-Richtmyer))**.** *For a consistent numerical method, stability is equivalent to convergence.*

Proof. See [QSS00, Section 2.2.1]. □

Finally, we define several qualitative terms of the numerical method (1.3) which quantify only in comparison to another method.

Definition 1.20.

- *accuracy – how large is the computational error,*
- *efficiency – how many mathematical operations (or how long computational time) are necessary for obtaining the approximate solution u_h,*
- *robustness – the independence of the previous properties for the data.*

Therefore, if we say that "numerical method No. 1 is more accurate but less efficient than method No. 2" it means that the method No. 1 gives lower computational error than method No. 2 but it requires longer computational time. Usually, the accuracy and efficiency are opposite, it is necessary to balance them carefully.

1.2.3 Implementation of numerical method and mathematical software

Implementing a numerical method on a computer we should take into account:

- the output of the computer u_h^* should be an approximation of u_{phys}, the process of the modelling is a "chain"

$$\mathscr{P}_{\text{phys}} \quad \Longrightarrow \quad \mathscr{P} \quad \Longrightarrow \quad \mathscr{P}_h \quad \Longrightarrow \quad \mathscr{P}_h^*.$$

  A chain is only as strong as its weakest link

- However, each "link" influence the neighbouring one
- accuracy, efficiency, robustness: always a compromise, we can not have anything ideal
- numerical software deals mostly with $\mathscr{P}_h$ and $\mathscr{P}_h^*$

1.3 Glossary of Verification and Validation Terms

We present several (more advancing terms) dealing with a mathematical solution of real-world problems.

- Model is a representation of a physical system or process intended to enhance our ability to understand, predict, or control its behaviour.
  - Abstract model is our mathematical model, e.g., partial differential equations including initial and boundary condition.
  - Computational model is an numerical approximation together with an algoritmization and a computer implementation of the mathematical model.
- Modelling is the process of construction or modification of a model.
- Simulation is the exercise or use of a model. (That is, a model is used in a simulation).
- Calibration is the process of adjusting numerical or physical modelling parameters in the computational model for the purpose of improving agreement with experimental data.
- Prediction is the use of a model to foretell the state of a physical system under conditions for which the model has not been validated.
- Robustness defines the ability of the numerical method to provide a solution despite variabilities in the initial solution and control parameters.

- Verification is the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model.
  - Verification of a code – looking for bugs, incorrect implementations
    * checking of basic relationships expected, i.e. mass conservation
    * simulation of "highly accurate" verification cases, i.e., case with (quasi-) analytical solutions
    * study of mesh adaptations,
    * all the options of the code should be examined.

    Verification should not performed with experimental data.
  - Verification of a calculation – involves error estimation, consistency with theoretical results, experimental order of convergence, etc.
- Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model. Several levels:
  - Unit Problems involve simple geometry and one relevant physical feature. An example is the measurement of a turbulent boundary layer over a flat plate. The experiment data set contains detailed data collected with high accuracy. The boundary conditions and initial conditions are accurately measured.
  - Benchmark Cases involve fairly simple hardware representing a key feature of the system, contain two separate physical features. An example is a shock / boundary layer interaction. The experiment data set is extensive in scope and uncertainties are low; however, some measurements, such as, initial and boundary conditions, may not have been collected.
  - Subsystem Cases involve geometry of a component of the complete system which may have been simplified. The physics of the complete system may be well represented; but the level of coupling between physical phenomena is typically reduced. An example is a test of a subsonic diffuser for a supersonic inlet. The exact inflow conditions may not be matched. The quality and quantity of the experiment data set may not be as extensive as the benchmark cases.
  - Complete System Cases involve actual hardware and the complete physics. All of the relevant flow features are present. An example is a flow around the whole plane model. Less detailed data is collected since the emphasis is on system evaluation. Uncertainties on initial and boundary conditions may be large.

More details can be found in [Ver].

Homeworks

Exercise 1. *Decide, if the following assertions are true or not.*

1. *The model problem is ill-conditioned, if its solution is very sensitive to the perturbation of the data.*

2. *A use of a more accurate finite precision arithmetic improve the condition number of the model problem.*

3. *The condition number of the model problem depends on the choice of the algorithm for its approximate solution.*

4. *The choice of the numerical method has an influence on the propagation of the data error during the computation.*

Exercise 2. *Write a simple code to verify Example 1.15. Find experimentally the limit value h such that the Euler method is stable for the problem* (1.12) *from Example 1.15.*

Chapter 2

Software for computational mathematics

2.1 Types of numerical software

- mathematical software as `Matlab`, suitable for the study of the behaviour and verification of numerical method, not suitable for practical computations of large problems, simple work but it is inefficient

- advanced software platforms for numerical solution of various problems (e.g., `FEniCS`), simplify the work, efficiency is high but it is difficult to see the "core of the computation", therefore it is difficult to distinguish different aspects of numerical methods (due to black box). Generally, it is possible to gain the corresponding information (source files are often available) but not simple task and the efficiency is significantly decreased.
  Cf. list of codes, e.g., `https://www.cfd-online.com/Links/soft.html` and many others

- software libraries for "standard" numerical methods (e.g., `LAPACK`), source codes in the forms of subroutines which can be link to your own code, mostly freely available, e.g., `Netlib` – `www.netlib.org` Strongly advantageous to use it, see Chapter 7.

2.2 General requirements for numerical software

- accuracy – the error of the result is under the prescribed tolerance (or on the level of the machine accuracy ϵ_{mach})

- efficiency – fast computation and possibly a low amount of the used computer memory. Both requirements are frequently opposite since the storing of pre-computed or temporal variables and arrays can decrease significantly the computational time but requires more memory.

Example 2.1. *We have the functions*

$$f_k(x), \quad k = 1, \ldots, n, \qquad \varphi_i(x), \quad i = 1, \ldots, dof.$$

The aim is to compute

$$\int_\Omega f_k(x)\varphi_i(x)\,\mathrm{d}x \approx \sum_{j=1}^{M} w_j f_k(x_j)\varphi_i(x_j) \qquad k = 1, \ldots, n,\ i = 1, \ldots, dof.$$

We have

$w_j,\ j = 1, \ldots, M$	`weights(1:M)`
$f_k(x_j),\ j = 1, \ldots, M,\ k = 1, \ldots, n,$	`func(1:M, 1:n)`
$\varphi_i(x_j),\ j = 1, \ldots, M,\ i = 1, \ldots, dof,$	`phi(1:dof, 1:M)`

The following part of the code

```
allocate(temp(1:M))
do k=1,n
   temp(1:M) = func(1:M,k) * weights(1:M)
   do i=1, dof
      vector(k,i) = dot_product(temp(1:M), phi(i, 1:M) )
   end do
end do
deallocate(temp)
```

is faster (but need more memory – array `temp`*) than the following one*

```
do k=1,n
   do i=1, dof
      vector(k,i) = dot_product(func(1:M,k) * weights(1:M), phi(i, 1:M) )
   end do
end do
```

However, the amount of the additional memory is not essential in this case.

- the preference is to save the computational time than the computer memory
- suitable algoritmization, minimization of the number of mathematical operations
- the (de)allocation of arrays, `if` conditions, control of `do` cycles requires also a non-negligible time

- robustness – code works for a wide range of input data. Robustness depends on the used method, but the code should avoid a failure of the computation, situation like $\sqrt{-1}$. The user can make a mistake and run the code with wrong data. However, too frequent verification of the data and results increase the computational time.

- transportability – codes are "computer independent", codes have to give the same (or at least very similar) results using different computers. The differences are usually caused by

  - different operating systems (`Windows`, `Unix`, `Linux`, ...)
  - different systems on computers "32bits", "64bits"
  - different translators, `gfortran`, `ifort` (Intel), ..., there exists some norms but not always unique and not always are kept.
  - different versions of the translators `gfortran-4.8`, `gfortran-5.1`, etc.

  Typical situation: your made a perfectly working code on your laptop and when the files are moved to the computer of your costumer, the code does not work.
  Advice: use standard structures, no special commands which could seem to be favourable, avoid commands of the operator system (e.g., calling vizualization software), test different computers, ... Let us note that 100% transportability is "an illusion" (namely for the computation in float-point system).

- readability of the code – use easily understandable commands, split particular works in a hierarchy of subroutines, frequent comments, sometimes can decrease the efficiency. Avoid a non-stadard formulae, e.g., the part of the code

```
do i=1,10
  do j=1,10
     a(i,j) = (i/j) * (j/i)
  end do
end do
```

  defines a unit matrix but the construction is awful.

  Graphical structure is important, e.g.,

```
do k=1,n
   do i=1, dof
      vector(k,i) = dot_product(func(1:M,k) * weights(1:M), phi(i, 1:M) )
   end do
end do
```

  is much better than

```
do k=1,n
do i=1, dof
vector(k,i) = dot_product(func(1:M,k) * weights(1:M), phi(i, 1:M) )
end do
end do
```

or even

```
                    do k=1,n
   do i=1, dof
             vector(k,i) = dot_product(func(1:M,k) * weights(1:M), phi(i, 1:M) )
                                              end do
end do
```

Editors usually highlights the commands, e.g., `gedit` editor

```
do k=1,n
   do i=1, dof
      vector(k,i) = dot_product(func(1:M,k) * weights(1:M), phi(i, 1:M) )
   end do
end do
```

- documentation of the code – important but very often omitted, software tools, e.g., `doxygen` which creates a documentation from the comments in the code, very useful (the documentation is created simultaneously) but not always sufficient.

2.3 Procedure for the creation of a software

1. creation of the algorithm: proposal of a numerical methods and hierarchy of its performance
2. structure of the implementation: data structures, main code, hierarchy of subroutines
3. coding and debugging
4. testing of the code
   (a) test on problems with an exactly known solution
   (b) test on problems with an expected solution
   (c) test on problems without any knowledge of the solution
5. documentation
6. maintenance of the software

Few comments:

- procedure can be split in several task
- testing of particular tasks is important, list of a testing examples has to be developed simultaneously
- preliminary variants of the code, adding of verification of partial results, later could be removed
- be prepare that the first variant is not suitable, flexibility for later modifications is recommended
- program languages offers options for debugging a code, e.g., the range of arrays are always checked, code runs longer, recommended to use
- program languages offers optimization, options `-O2`, `-O3`, `-O4` ... Accelerate the computation but sometimes gives a different behaviour, the reason is usually a bug in the code (e.g., a variable is not initialized, a default value is used, it can differ.

Chapter 3

Machine arithmetic

3.1 Machine representation of numbers

3.1.1 Definition of the system of numbers

Any machine operation is affected by rounding error, since only a finite subset of real numbers can be represented by a computer. By $\mathbb{F}$ we denote this subset, which we call the floating-point number system. Obviously $\mathbb{F} \subset \mathbb{R}$. Each $\mathbb{F}$ can be characterized by four integer parameters:

- β – the base,
- t – the length of the mantissa (the accuracy),
- $L < U$ – the range of the exponent.

Hence, we write

$$\mathbb{F} = \mathbb{F}(\beta, t, L, U). \tag{3.1}$$

If $x \in \mathbb{F}$ is a number from $\mathbb{F}$ then it can be express by

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \cdots + \frac{d_{t-1}}{\beta^{t-1}} \right) \beta^e, \tag{3.2}$$

where $0 \leq d_i \leq \beta - 1,\ i = 0, \ldots, t-1$ and $L \leq e \leq U$, e and $d_i,\ i = 0, \ldots, t-1$ are integers. The t-plet $(d_0 d_1 \ldots d_{t-1})$ is called the mantissa (or also significant) and the number e is the exponent. Hence, there is the representation

$$x \in \mathbb{F} \quad \longleftrightarrow \{d_0, d_1, \ldots, d_{t-1}; e\}. \tag{3.3}$$

Since the basic computer information is the bit having values 0 (=off) or 1 (=on), it is suitable use $\beta = 2$. Then $d_i = 0$ or 1, $i = 0, \ldots, t-1$. The parameters (β, t, L, U) should satisfy some standards, namely the standard of the Institute of Electrical and Electronics Engineers (IEEE) and International Electrotechnical Commission (IEC) from 1985, see Table 3.1.

type	β	t	L	U	$\#\mathbb{F}$	ϵ_{mach}
half	2	11	-14	15	6.11E+05	4.88E-04
single	2	24	-126	127	4.26E+09	5.96E-08
double	2	53	-1022	1023	1.84E+19	1.11E-16
extended	2	64	-16 382	16 383	6.04E+23	5.42E-20
quadruple	2	113	-16 382	16 383	3.40E+38	9.6.E-35

Table 3.1: Standards of IEEE and IEC for floating point arithmetic

3.1.2 Normalized system $\mathbb{F}$

The number representation is not unique since, e.g,

$$\{1010; 0\} \longleftrightarrow 1.010 \times 10^0 = 0.101 \times 10^1 \longleftrightarrow \{0101; 1\} \tag{3.4}$$

Then, it is suitable to use the normalization of $\mathbb{F}$.

Definition 3.1. *We say that the system $\mathbb{F}$ is normalized if we prescribe the additional requirement*

$$d_0 \neq 0. \tag{3.5}$$

We speak about the normal numbers and the normalization of $\mathbb{F}$.

The normalization gives the following properties.

- the numbers in the system $\mathbb{F}$ are represented uniquely (except 0)
- the maximal possible accuracy is used
- for $\beta = 2$, we have always $d_0 = 1$ and then it is not necessary to store it (the "hidden" or "implicit" bit).

3.1.3 Renormalization of the system $\mathbb{F}$

The numbers in $\mathbb{F}$ are not distributed equidistantly (only relatively equidistantly).

Example 3.2. *Let $\beta = 2$, $t = 3$, $L = -1$ and $U = 1$. Then the numbers from the normalized $\mathbb{F} = \mathbb{F}(\beta, t, L, U)$ are plotted here:*

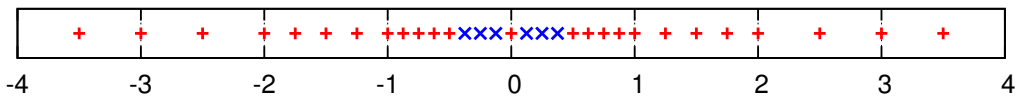
This example shows that there is a gap around zero due to the normalization assumption (3.5). Therefore, we introduce the renormalization of the system $\mathbb{F}$ replacing (3.5) by other requirement.

Definition 3.3. *We say that the system $\mathbb{F}$ is renormalized if we prescribe the requirement*

$$d_0 \neq 0 \text{ only if } e \neq L. \tag{3.6}$$

We speak about renormalization of $\mathbb{F}$ and the numbers with $d_0 = 0$ and $e = L$ are called sub-normal numbers.

The renormalization fill the gap around the zero:



On the other hand, we loose some accuracy.

3.1.4 Under- and over-flow levels

The system $\mathbb{F}$ is finite, the number of numbers in the normalized system $\mathbb{F}$ is equal to

$$\#\mathbb{F} = \underbrace{2}_{\text{sign}\pm}\Big(\underbrace{\beta-1}_{\text{first digit}}\Big)\Big(\underbrace{\beta}_{\text{other digits}}\Big)^{t-1}\Big(\underbrace{U-L+1}_{\text{exponents}}\Big) + \underbrace{1}_{\text{zero}}. \tag{3.7}$$

See Table 3.1 for the concrete values.

We define the maximal and the minimal positive numbers of the normalized system $\mathbb{F}$ by

$$\text{OFL} := \max_{x\in\mathbb{F}} |x| = (1-\beta^{-t})\beta^{U+1}, \tag{3.8}$$

$$\text{UFL} := \min_{x\in\mathbb{F}} |x| = \beta^{L}, \tag{3.9}$$

where OFL means the over-flow level and UFL means the under-flow level.

The renormalized system $\mathbb{F}$ has (instead of (3.9)) the value of UFL given by

$$\text{UFL}^\star = \beta^{L-t+1}. \tag{3.10}$$

3.1.5 Rounding

Generally, $x \notin \mathbb{F}$ for $x \in \mathbb{R}$. Hence, we have to introduce a mapping

$$\hat{}: \mathbb{R} \to \mathbb{F}. \tag{3.11}$$

We require, e.g.,

- $a \in \mathbb{F} \Rightarrow \hat{a} = a$
- $a \leq b \Rightarrow \hat{a} \leq \hat{b}$

In practice, two main approaches are used:

- chopping: a number $a \in \mathbb{R}$ is expanded in the series with the base β, i.e.,

$$a = \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \cdots + \frac{d_{t-1}}{\beta^{t-1}} + \frac{d_t}{\beta^t} + \ldots \right) \beta^e,$$

  and only first t digits of the mantissa are used, i.e.,

$$\hat{a} := \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \cdots + \frac{d_{t-1}}{\beta^{t-1}} \right) \beta^e$$

- round to nearest, where we put

$$\hat{a} = \arg\min_{x \in \mathbb{F}} |a - x|.$$

  If there exists two numbers $x \in \mathbb{F}$ satisfying this definition, we use the number having the last digit even. For example, for $\beta = 10$ and $t = 2$ we put

a		$\hat{a}$	a		$\hat{a}$
1.649	$\to$	1.6	1.749	$\to$	1.7
1.650	$\to$	1.6	1.750	$\to$	1.8
1.651	$\to$	1.7	1.751	$\to$	1.8

  This approach is more expensive but more accurate, standard of IEEE.

The difference $x - \hat{x}$ is called the rounding error.

3.1.6 Machine accuracy

Definition 3.4. *Let $\hat{}$ be the operator of the rounding (3.11). We define the positive real number ϵ_{mach} by*

$$\epsilon_{\text{mach}} := \max_{x \in \mathbb{R} \cap [UFL, OFL]} \left| \frac{\hat{x} - x}{x} \right|. \tag{3.12}$$

The number ϵ_{mach} is called the machine accuracy or simply the accuracy and it represents the maximal possible relative rounding error

We can verify that for the normalized (but not renormalized) system $\mathbb{F}$, we have

$$\epsilon_{\text{mach}} = \begin{cases} \beta^{1-t} & \text{for the chop,} \\ \frac{1}{2}\beta^{1-t} & \text{for the round to nearest.} \end{cases} \tag{3.13}$$

The number of correct (decimal) digits in $\hat{x}$ is equal roughly to $-\log_{10} \epsilon_{\text{mach}}$, i.e. about 7 or 8 in the single and about 15 or 16 in the double precision. In the other words, ϵ_{mach} is the smallest possible number such that $1 + \epsilon_{\text{mach}} > 1$ in the finite precision arithmetic, see Table 3.1.

Remark 3.5. *Definition 3.4 implies that if* $x \in \mathbb{R}$ *such that* $UFL \leq |x| \leq OFL$ *then there exists* $\delta \in \mathbb{R}$, $|\delta| \leq \epsilon_{\text{mach}}$ *such that* $\hat{x} = x(1+\delta)$.

Remark 3.6. *Let us recall that UFL is given by the exponent (namely lower bound L) and* ϵ_{mach} *be the length of the mantissa (the parameter t). Usually, we have*

$$0 < UFL < \epsilon_{\text{mach}} < OFL$$

3.1.7 Infs and NaNs

Usually, the system $\mathbb{F}$ is enriched by two special values:

- NaN – Not a Number, which is a result of certain "invalid" operations, such as $0/0$ or $\sqrt{-1}$. In general, NaNs will be propagated i.e. most operations involving a NaN will result in a NaN, although functions that would give some defined result for any given floating point value will do so for NaN as well, e.g. $NaN^0 = 1$.
- Inf – Infinity (usually with signs $+Inf$ and $-Inf$), which approximate the infinity in a limits, e.g., $1./0 = (+Inf)$, $\tan(\pi/2) = (+Inf)$, etc. IEEE standard requires infinities to be handled in a reasonable way, such as $(+Inf)+(+7) = (+Inf)$, $(+Inf)\times(-2) = (-Inf)$ and $(+Inf)\times 0 = NaN$.

Moreover, let $a =$ OFL then $a+1 = (+Inf)$ and $-a-1 = (-Inf)$. On the other hand, let $a =$ UFL then $a/2 = 0$ usually.

Do Exercise 3

3.2 Storage requirements of the system $\mathbb{F}$

The basic computer information is the bit. Each number from $\mathbb{F}$ requires several bits of the computer memory for its storing based on the used type, namely

- half requires 16 bits: 1 sign bit, 10 bits for mantissa, 5 bits for exponent
- single requires 32 bits: 1 sign bit, 23 bits for mantissa, 8 bits for exponent
- double requires 64 bits: 1 sign bit, 52 bits for mantissa, 11 bits for exponent
- extended requires 80 bits: 1 sign bit, 64 bits for mantissa, 15 bits for exponent [1]
- quadruple requires 128 bits: 1 sign bit, 112 bits for mantissa, 15 bits for exponent

[1] In contrast to the single and double-precision formats, this format does not utilize an implicit/hidden bit

Remark 3.7. *The single precision uses 8 bits for the exponents, i.e, we have* $2^8 = 256$ *possibilities. The number of possible exponents is equal to* $U - L + 1 = 254$*, i.e., two possible exponents are blank. The exponent* 00000000 *(in the binary representation) is used for the subnormal numbers and the exponent* 11111111 *is used for the special values Inf and NaN.*

3.3 Mathematical operations in the system $\mathbb{F}$

The system $\mathbb{F}$ was introduce to approximate the real numbers $\mathbb{R}$. Further, we need to deal with the usual mathematical operations (e.g., adding, subtracting, multiplication, division) within the system $\mathbb{F}$. We speak about the finite precision arithmetic.

3.3.1 Basic aspects of the finite precision arithmetic

Let $*$ denote a mathematical operation on the real numbers $\mathbb{R}$, i.e., $x*y \in \mathbb{R}$ for any $x, y \in \mathbb{R}$. E.g., $* \in \{+, -, \times, :\}$. If $x, y \in \mathbb{F}$ then $x*y \notin \mathbb{F}$ generally. In order to work within the system $\mathbb{F}$, we have to introduce some embedding $\widehat{\ }: \mathbb{R} \to \mathbb{F}$, e.g., the chop or the round to the nearest. Then, to the mathematical operation $* : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, we define its analogue $\hat{*} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ by

$$x \hat{*} y = \widehat{x * y} \tag{3.14}$$

In virtue of Remark 3.5, we have $\widehat{x} = x(1+\rho)$, where $|\rho| \le \epsilon_{\text{mach}}$. Analogously,

$$x \hat{*} y = (x * y)(1+\rho), \quad |\rho| \in \epsilon_{\text{mach}}. \tag{3.15}$$

Example 3.8. *Let* $x, y, x \in \mathbb{R}$*. We assume that* $|x + y + z| \le OFL$*, for simplicity. We want to compute* $x + y + z$*. In the finite precision arithmetic, we can evaluate only* $x \widehat{+} y \widehat{+} z$*. We investigate the corresponding rounding error. Then, using (3.15), we have*

$$\begin{aligned}(x \widehat{+} y) \widehat{+} z &= (x+y)(1+\rho_1) \widehat{+} z = [(x+y)(1+\rho_1) + z](1+\rho_2) \\ &= x + y + z + (x+y)(\rho_1 + \rho_2 + \rho_1\rho_2) + z\rho_2,\end{aligned}$$

where $|\rho_1| \le \epsilon_{\text{mach}}$ *and* $|\rho_2| \le \epsilon_{\text{mach}}$*. Using the different order of adding, we have*

$$\begin{aligned}x \widehat{+} (y \widehat{+} z) &= x \widehat{+} (y+z)(1+\rho_3) = [x + (y+z)(1+\rho_3)](1+\rho_4) \\ &= x + y + z + x\rho_4 + (y+z)(\rho_3 + \rho_4 + \rho_3\rho_4),\end{aligned}$$

where $|\rho_3| \le \epsilon_{\text{mach}}$ *and* $|\rho_4| \le \epsilon_{\text{mach}}$*. From the above relations we deduce that the adding in the finite precision arithmetic is not associative Similarly, we obtain the same conclusion for the multiplication.*

Do Exercise 4

Remark 3.9. *The adding (and similarly the multiplication) in the finite precision arithmetic is usually commutative, we can write*

$$x \widehat{+} y = \widehat{x + y} = \widehat{y + x} = y \widehat{+} x.$$

3.3.2 Practical implementation of the adding and subtracting

Let $x, y \in \mathbb{F}$ be given in the form (see (3.3))

$$x = \{a_0 a_1 \dots a_{t-1}; e_1\} \qquad \left(\Leftrightarrow x = (\sum_{i=0}^{t-1} \frac{a_i}{\beta^i})\beta^{e_1} \right), \tag{3.16}$$
$$y = \{b_0 b_1 \dots b_{t-1}; e_2\} \qquad \left(\Leftrightarrow y = (\sum_{i=0}^{t-1} \frac{b_i}{\beta^i})\beta^{e_2} \right)$$

and we want to compute $x + y$ in the finite precision arithmetic.

The basic idea of the adding of two numbers in the exponential form is the following. We rewrite the numbers into the form having the same exponent and adding the mantissas, we obtain the final result. In practical implementations, both numbers are first transformed to the higher precision (e.g., single to double, double to extended), then both mantissas are added and the results is transformed back to the original precision.

For example, the numbers x, y in (3.16) are transformed to the higher precision, formally written as

$$\begin{aligned} x &= \{a_0 a_1 \dots a_{t-1} 000 \dots 0; e_1\}, \\ y &= \{b_0 b_1 \dots b_{t-1} 000 \dots 0; e_2\}. \end{aligned} \tag{3.17}$$

Example 3.10. *Let, e.g., $e_1 = e_2 + 2$, we rewrite the numbers (3.17) to the form with the same exponent*

$$\begin{aligned} x &= \{a_0 a_1 a_2 \dots a_{t-1} 000 \dots 0; e_1\} \\ y &= \{00 b_0 b_1 \dots b_{t-3} b_{t-2} b_{t-1} 0 \dots 0; e_2 + 2\} = \{00 b_0 b_1 \dots b_{t-3} b_{t-2} b_{t-1} 0 \dots 0; e_1\}. \end{aligned} \tag{3.18}$$

Then we add the mantissas and obtain formally

$$x \widehat{+} y = \{a_0 a_1 (a_2 + b_0)(a_3 + b_1) \dots (a_{t-1} + b_{t-3}) b_{t-2} b_{t-1} 0 \dots 0; e_1\}. \tag{3.19}$$

Of course, (3.19) is true only when $a_{i+2} + b_i < \beta$, $i = 0, \dots, t-3$. Finally, the transformation of the results to the original precision gives

$$x \widehat{+} y =: z = \{a_0 a_1 (a_2 + b_0)(a_3 + b_1) \dots (a_{t-1} + b_{t-3}); e_1\}. \tag{3.20}$$

Example 3.10 gives that the result is in fact independent of b_{t-2} and b_{t-1}. Generally, if the exponents of both numbers differs very much than we lose the accuracy of the mathematical operation. The loss of the accuracy is increasing for the increasing difference between both exponents.

Example 3.11. *Let as consider the infinite row $\sum_{n=1}^{\infty} \frac{1}{n}$. Obviously, this row diverges (the sum is infinity). However, the evaluation of this sum by a simple subroutine*

```
   real :: sum
   integer :: n

   sum = 0.
   n = 0
10 n = n + 1
   sum = sum + 1./n
   print*, sum
   goto 10
```

leads to a finite limit number (approximately 15.40 in the single precision and something more in the double precision).

This follows from the fact that

$$\exists n_0 \in \mathbb{N}: \ \frac{1}{n_0} \leq \epsilon_{\text{mach}} \sum_{n=1}^{n_0-1} \frac{1}{n}.$$

Therefore, the terms $1/n_0$, $1/(n_0+1)$, etc. does not bring any increase of the sum.

Do Exercise 5

3.3.3 Practical implementation of the multiplication and division

Practical implementation of the multiplication and division is similar to the implementation of the adding and subtracting described in Example 3.10. The numbers are transformed to the higher precision type, the operation is performed and the result is transformed back to the original type.

Let $x, y \in \mathbb{F}$ be given in the form (3.16), then their product in the finite precision arithmetic $x \hat{\times} y$ has the mantissa with the length equal approximately to $2t$. Therefore, any operation leads to a loss of the accuracy.

3.3.4 Cancellation

The subtraction of two similar numbers leads to a large loss of the accuracy. This effect is called the cancellation and it is illustrated in the following example.

Example 3.12. *Let*

$$x = 123.456478, \quad y = 123.432191 \qquad \Longrightarrow \qquad x - y = 0.0024267 = 2.4267 \times 10^{-2}.$$

We consider $\mathbb{F}$ with $\beta = 10$ and $t = 6$. The representation of the numbers x and y in $\mathbb{F}$ reads

$$x = 1.23456 \times 10^2, \quad y = 1.23432 \times 10^2$$

and their difference in $\mathbb{F}$ *is*

$$x\widehat{-}y = 2.40000 \times 10^{-2},$$

hence the result has only two decimal digits. Therefore, the relative rounding error of this computation is

$$\frac{(x\widehat{-}y) - (x-y)}{x-y} = \frac{2.4 \times 10^{-2} - 2.4267 \times 10^{-2}}{2.4267 \times 10^{-2}} = 0.011003$$

i.e., more than 10^{-2} *(using* $t = 6$*).*

The total loss of the accuracy is the following case, let $\varepsilon < \epsilon_{\text{mach}}$ then

$$(1+\varepsilon) - (1-\varepsilon) = 2\varepsilon, \quad \text{but} \quad (1\widehat{+}\varepsilon)\widehat{-}(1\widehat{-}\varepsilon) = 1\widehat{-}1 = 0,$$

i.e., the error is 100%.

Example 3.13. *Let us consider the quadratic equation*

$$ax^2 + bx + c = 0. \tag{3.21}$$

The roots are given either by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.22}$$

or by

$$x_{1,2} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}. \tag{3.23}$$

Let $a = 0.05010$, $b = -98.78$ *and* $c = 5.015$. *The exact roots of* (3.21) *are*

$$x_1 = 1971.605916, \quad x_2 = 0.05077068387.$$

Let us consider the system $\mathbb{F}$ *with* $\beta = 10$ *and* $t = 4$. *Then the roots evaluated in the finite precision arithmetic by formula* (3.22) *are*

$$x_1 = 1972, \quad x_2 = 0.0998$$

and by formula (3.23) *are*

$$x_1 = 1003, \quad x_2 = 0.05077.$$

Therefore, x_2 *given by* (3.22) *and* x_1 *given by* (3.23) *are completely wrong. The reason is the cancellation since* $\sqrt{b^2 - 4ac} = 98.77 \approx b$ *in the finite precision arithmetic.*

Do Exercise 6-10

3.3.5 Computer performance

In computing, floating point operations per second (FLOPS, flops) is a measure of computer performance, useful in fields of scientific computations that require floating-point calculations. For such cases it is a more accurate measure than measuring instructions per second.

The use unit of the computer performance:

Name	Unit	Value	
kiloFLOPS	kFLOPS	10^3	
megaFLOPS	MFLOPS	10^6	
gigaFLOPS	GFLOPS	10^9	
teraFLOPS	TFLOPS	10^{12}	
petaFLOPS	PFLOPS	10^{15}	
exaFLOPS	EFLOPS	10^{18}	actual limit of computers
zettaFLOPS	ZFLOPS	10^{21}	
yottaFLOPS	YFLOPS	10^{24}	
brontoFLOPS	BFLOPS	10^{27}	

The term *flop* is sometimes used also for the mathematical operations in $\mathbb{F}$.

3.3.6 Integer numbers

- Computation with integer numbers does not suffer from rounding errors.
- There are maximal and minimal integers: $I_{\min} < 0 < I_{\max}$.
- A pitfall: if $m, n \in \mathbb{Z}$ such that $m, n < I_{\max}$ and $m+n > I_{\max}$ then in finite precision arithmetic $m + n < 0$!!

Homeworks

Exercise 3. *Find experimentally the approximate values of OFL, UFL and ϵ_{mach}. Use an arbitrary compute, operating system and programming language and write a simple code seeking these value. Compare the obtained value with the theoretical ones given by (3.8), (3.9) and (3.13). Try different types (at least single and double).*

Exercise 4. *Show, similarly as in Example 3.8, that the adding in the finite precision arithmetic is not distributive.*

Exercise 5. *Verify Example 3.11. Modify a code in such a way that it results a different limit value (hint: modify the order of the summing).*

Exercise 6. *Try and explain the behaviour of the following codes*

```
   eps = 1.
10 eps = eps/2.
   write(*,'(es18.10)') eps
   eps1 = eps + 1
   if(eps1 > 1.) goto 10
```

and

```
   eps = 1.
10 eps = eps/2.
   write(*,'(es18.10)') eps
   if(eps > 0.) goto 10
```

Explain the differences?

Exercise 7. *The number* $e = 2.7182817459106445\ldots$ *can be defined as* $e = \lim_{n\to\infty}(1 + 1/n)^n$. *This suggests an algorithm for calculating* e: *choose* n *large and evaluate* $e^* = (1 + 1/n)^n$. *The results are:*

n	e^*	# *correct digits*
1.0000E+04	*2.718145926825*	*3*
1.0000E+05	*2.718268237192*	*4*
1.0000E+06	*2.718280469096*	*5*
1.0000E+07	*2.718281694133*	*7*
1.0000E+08	*2.718281798339*	*7*
1.0000E+09	*2.718282051996*	*6*
1.0000E+10	*2.718282052691*	*6*
1.0000E+11	*2.718282052980*	*6*
1.0000E+12	*2.718523495870*	*3*
1.0000E+13	*2.716110033705*	*2*
1.0000E+14	*2.716110033666*	*2*
1.0000E+15	*3.035035206235*	*0*
1.0000E+16	*1.000000000000*	*0*
1.0000E+17	*1.000000000000*	*0*
1.0000E+18	*1.000000000000*	*0*

Explain this effect, i.e, why the approximation e^* *of the Euler number* e *is first increasing for increasing* n *and then it decrease until complete information is lost.*

Exercise 8. *Cf. Example 3.13. Write a code for the solution of the quadratic equation (3.21) which is robust with respect the overflow, underflow and the cancellation. Test the following data:*

- a =*6*, $b = 5$, $c = -4$
- a =*6E+30*, $b = 5$*E+30*, $c = -4$*E+30*

- $a = 1$, $b = -1E + 6$, $c = -1$

Exercise 9. *The Taylor series for the error function is*

$$erf(x) = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{k!(2k+1)}.$$

This series converges for all $x \in \mathbb{R}$. Code it and try $x = 0.5$, $x = 1.0$, $x = 5$ and $x = 10$. Explain the results.

Exercise 10. *Numerical differentiating of a function f is based on the formula:*

$$f'(\bar{x}) \approx \frac{f(\bar{x} + h) - f(\bar{x})}{h} =: Df(\bar{x}; h).$$

- *Determine the dependence of discretization and rounding errors on h.*
- *For which h the formula is the most accurate (in finite precision arithmetic).*
- *Write a simple code for $f(x) = x^2$ at $\bar{x} = 1.5$ and test several values h.*
- *Try to find an algorithm, which gives the optimal size of h.*

Chapter 4

Linux

Linux is a family of free and open-source software operating systems built around the Linux kernel. Typically, Linux is packaged in a form known as a Linux distribution (or distro for short) for both desktop and server use. For more detailed description see, e.g., `https://en.wikipedia.org/wiki/Linux`.

It is Unix-base operating system, there are several distributions: SuSe, Ubuntu, etc. Ubuntu is actually very good choice.

Linux perfectly suits for scientific computing, namely for the solution of large demanding problems.

4.1 Installation of Linux

- a direct installation on the computed
- installation as a sub-system on Windows is also possible, many manuals and directions on web, see also the lecture web.

4.1.1 Instalation using Virtual Box Machine

The detailed description: link
`https://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/NS_source/Linux/index-install.html`

4.2 Basic commands of Linux

Commands from the `command lines`

- `name/codes/integrals/data/` – structures of directories `/`
- `pwd` – return actual directory

- `mkdir` – create a (sub-)directory
- `rmdir` – remover a (sub-)directory
- `cd` – changes the directory
- `ls` – list the files, option `ls -l`
- `rm` – remove a file (or directory), option `rm -r` - DANGEROUS !
- `cp file1 file2` – copy a file
- `mv file1 file2` – rename a file
- `mv file1 dir2` – move a file to directory
- `~/` – home directory
- `touch` – create a file
- `less` – list the file
- `more` – list the file
- `man` – list the help (manual) for the given word

4.3 Installation of libraries in Linux

For the purposes of this lecture, we need

- a Fortran 90 translater, the good choice is `gfortran`
- suitable text editor (e.g., `gedit`, `emacs`)
- software for visualization `gnuplot`

Installation of packages, e.g., of `gedit`, try

```
sudo apt-get install gedit
```

similarly for gnuplot

```
sudo apt-get install gnuplot
```

Your Linux password is required.

4.4 gfortran on Linux

Try

```
sudo apt-get install gfortran
```

or directly from `https://gcc.gnu.org/wiki/GFortranBinaries#GNU.2BAC8-Linux`

Chapter 5

Fortran

For more detailed sources see the lecture webpage.

5.1 Why Fortran?

Title: "fortran" = **for**mula **tran**slation. General Assessment[1]

- It is often perceived that Fortran is an outdated programming language, which is used only by dinosaurs, who got stuck in the past and cannot learn much more efficient languages such as C++.
- If you program like a dinosaur, your code will be outdated no matter what language you use.
- Fortran is designed for number-crunching. It is not for problems with sophisticated data structures or logic. If you follow rules, Fortran provides a very fast and easy to write code.
- Do not use archaic features in Fortran. Fortran is about speed: fast codes is our goal. More details in Section 5.5.

See, e.g., `https://www.ibiblio.org/pub/languages/fortran/ch1-2.html`

[1] https://docplayer.net/59589138-Programming-with-fortran.html

5.2 Special Fortran90 commands

- vectors & matrices operatins, **component-wise**
  - `A(1:N) = B(1:N) + C(1:N)`
  - `A(1:N) = B(1:N) * C(1:N)`
  - `A(1:M, 1:N) = B(1:M, 1:N) * C(1:M, 1:N)`
  - `A(1:N) = B(3, 1:N)`
  - block operations: `A(1:10, 1:10) = B(101:110, 201:210)`
  - `B(1:M,1:N) = transpose( A(1:N, 1:M) )`
  - combinations: `A(1:N) = B(1:N) * C(1:N) + M(1:N, 5)`
  - `x = sum( A(1:M, 1:N))` , `x = sum( abs( A(1:M, 1:N)))` ,
  - maximal value of an array `max_x = maxval( B(1:M, 1:K) )`
- `dot_product`, scalar product of two vectors, `a = dot_product(x(1:N), y(1:N) )`
- `matmul`, product of matrices, `A(1:M, 1:N) = matmul( B(1:M, 1:K), C(1:K, 1:N) )`

5.3 Example

Basic code for the multiplication of two matrices.

```
program multiplication

  integer, parameter :: size= 2000

  real, dimension(:,:), allocatable :: A,B,C
  real :: t1, t2, tt, mflops
  integer :: n, loops, i,j,l, case

  allocate( A(size, size), B(size, size), C(size,size) )

  n = size

  do i = 1, n
     do j = 1, n
        A(i,j) = i+j
        B(i,j) = i-j
     end do
  end do
```

```
  call cpu_time( t1 )

  call BlockMulti(n, A, B, C)

  call cpu_time( t2 )


  tt = (t2 - t1 )

  if(tt > 0) then
     mflops = 2*real(n)**3/tt/1.e6

     write(*,'(a10, i8, 2(a10, es14.6))' )  &
           " size = ", n, "time = ", tt, "  mflops = ", mflops

  else
     print*,'TOO short time'
  endif


  deallocate(A, B, C)

end program Multiplication


subroutine BlockMulti(n, A, B, C )
  integer, intent(in) ::  n
  !real :: A(lda,*), B(ldb,*), C(ldc,*)
  real, dimension(1:n, 1:n), intent(inout) :: A, B, C

  integer :: i, j

  do i = 1, n
     do j = 1, n
        C(i,j) = 0
        do k = 1, n
           C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end do
     end do
  end do

end subroutine BlockMulti
```

5.4 File Makefile

`Makefile` can be employed for the compilation and linking of bigger projects. An example:

```
TARGETS = fft.o dr_fft.o
TARGETS1 = fft1.o dr_fft1.o
TARGETS2 = fft2.o dr_fft2.o

## optimize for computations
FFLAGS=  -fPIC -fdefault-real-8 -fopenmp -O2 -w -ffpe-trap=invalid,zero,overflow

# for debugging of the code
FFLAGS= -fPIC -fdefault-real-8 -g -fbacktrace -fbounds-check -w -Wall -finit-real=nan
-finit-integer=-999999 -fno-align-commons -ffpe-trap=invalid,zero,overflow,denormal

FXX=gfortran

all: dr_fft dr_fft1 dr_fft2

dr_fft: $(TARGETS)
 $(FXX) $(FFLAGS) -o dr_fft $^
dr_fft1: $(TARGETS1)
 $(FXX) $(FFLAGS) -o dr_fft1 $^
dr_fft2: $(TARGETS2)
 $(FXX) $(FFLAGS) -o dr_fft2 $^

clean:
 rm -f *.o

%.o:%.f90
 $(FXX) $(FFLAGS) -c $?

%.o:%.f
 $(FXX) $(FFLAGS) -c $?
```

The compilation is executed by the command `make`. It automatically compile all updated source files.

Few options:

- `-fPIC -fdefault-real-8` – all real variables are in double precisions, some translators of fortran use `-r8`
- `-O2` – level of optimization

- `-W` – turn on warnings
- `-fPIC` – generate position-independent code (PIC) suitable for use in a shared library, reccomended for `LAPCK`
- `-fbounds-check` – check the ranges of arrays
- `-fbacktrace` – if error is met, the sequence of callings is written

5.5 Comparison of fortran and C language

Source: Notes on FORTRAN Programming, chapter 1-2, link

Fortran still dominates in the numerical computing world, but it seems to lose ground. The following points may help you make up your mind. (Partly adapted from the Fortran FAQ)

1) FORTRAN tends to meet some of the needs of scientists better. Most notably, it has built in support for:

   - Variable-dimension array arguments in subroutines. A feature required for writing general purpose routines without explicitly specifying the array dimensions passed to them. Standard C lacks this important feature (some compilers like gcc have it as non-standard extension) and the workarounds are very cumbersome (See Appendix C).

     This feature by itself is sufficient to prefer Fortran over C in numerical computing.
   - A rich set of useful generic-precision intrinsic functions. Such functions can be highly optimized (written in assembly language with optimized cache utilization), and they make programs standard at a higher level (and more portable).
   - Builtin complex arithmetic (arithmetic involving complex numbers represented as having real and imaginary components).
   - Array index-ranges may start and end at an arbitrary integer, the C convention of [0,N-1] is usually inconvenient.
   - Better I/O routines, e.g. the implied do facility gives flexibility that C's standard library can't match. The Fortran compiler directly handles the more complex syntax involved, and as such syntax can't be easily reduced to argument passing form, C can't implement it efficiently.
   - A compiler-supported infix exponentiation operator which is generic with respect to both precision and type, AND which is generally handled very efficiently, including the commonly occurring special case floating-point**small-integer.

- Fortran 90 supports an array notation that allows operations on array sections, and using vector indices.

  The new intrinsic functions allow very sophisticated array manipulations.

  The new array features are suitable for parallel processing.

- Fortran 90 supports automatic selection of numeric data types having a specified precision and range, and makes Fortran programs even more portable.

- Fortran extensions for parallel programming are standardized by the High Performance Fortran (HPF) consortium.

  Fortran 90 supports useful features of C (column independent code, pointers, dynamic memory allocation, etc) and C++ (operator overloading, primitive objects).

2) The design of FORTRAN allows maximal speed of execution:

   - FORTRAN 77 lacks explicit pointers, which is one reason that it is more amenable to automatic code optimization. This is very important for high-performance computing.

     Fortran 90 allows explicit pointers restricted to point only to variables declared with the "target" attribute, thus facilitating automatic optimizations.

   - Fortran was designed to permit static storage allocation, saving the time spent on creating and destroying activation records on the stack every procedure call/return.

     Recursive procedures are impossible with static allocation, but can be simulated efficiently when needed (very rare).

   - Fortran implementations may pass all variables by reference, the fastest method.

   - Fortran disallows aliasing of arguments in procedure-call statements (CALL statements and FUNCTION references), all passed argument lists must have distinct entries.

     Fortran disallows also aliasing between COMMON (global) variables and dummy arguments.

     These restrictions allows better compiler optimizations.

3) There is a vast body of existing FORTRAN code (much of which is publicly available and of high quality). Numerical codes are particularly difficult to port, scientific establishments usually do not have large otherwise idle programming staffs, etc. so massive recoding into any new language is typically resisted quite strongly.

4) FORTRAN 77 tends to be easier for non-experts to learn than C, because its 'mental model of the computer' is much simpler.

   For example, in FORTRAN 77 the programmer can generally avoid learning about pointers and memory addresses, while these are essential in C. More generally, in

FORTRAN 77 the difference between (C notation) `x`, `&x`, and often even `*x` is basically hidden, while in C it's exposed. Consequently, FORTRAN 77 is a much simpler language for people who are not experts at computer internals.

Because of this relative simplicity, for simple programming tasks which fall within its domain, (say writing a simple least-squares fitting routine), FORTRAN 77 generally requires much less computer science knowledge of the programmer than C does, and is thus much easier to use.

Fortran 90 changes the picture somewhat, the new language is very rich and complex, but you don't have to use or even know about all this complexity.

5) The C standard requires only a basic double-precision mathematical library, and this is often what you get. The FORTRAN standard, on the other hand, requires single & double precision math, many vendors add quad-precision (long double, REAL*16) and provide serious math support.

   Single-precision calculations may be faster than double-precision calculation even on machines where the individual machine instructions takes about the same time because single-precision data is smaller and so there are less 'memory cache misses'.

   Quad-precision (long double) calculations are sometimes necessary to minimize round-off errors.

   If you have only double-precision mathematical routines, the basic mathematical primitives will take up unnecessary CPU time when used in single-precision calculations and will be inexact if used with 'long double'.

6) FORTRAN is designed to make numerical computation easy, robust and well-defined:

   - The order of evaluation of arithmetical expressions is defined precisely, and can be controlled with parentheses.
   - The implicit type declaration feature saves time/typing (however it makes your program vulnerable to annoying and hard to detect bugs).
   - Case insensitivity eliminates bugs due to 'miscased' identifiers.
   - The lack of reserved words in the language gives the programmer complete freedom to choose identifiers.
   - The one statement per line principle (of course continuation lines are allowed with a special syntax) makes programs more robust.
   - Added blanks (space characters) are insignificant (except in character constants) this also contributes to the robustness of FORTRAN programs.
   - Linking with the mathematical library doesn't require any compiler option (in C you to have to use "-lm").

7) Last but not least, FORTRAN compilers usually emit much better diagnostic messages.

In summary, we can say that the difference between Fortran and C, is the difference between a language designed for numerical computations, and a language designed for other purposes (system programming).

```
+-------------------------------------------+
|                                           |
|     SUMMARY OF FORTRAN ADVANTAGES         |
|     ============================          |
|     a) Scientifically oriented            |
|     b) Better optimized code              |
|     c) A lot of existing code             |
|     d) Easier to learn                    |
|     e) More efficient mathematics         |
|     f) Easier to use and more robust      |
|     g) Better diagnostics                 |
|                                           |
+-------------------------------------------+
```

Chapter 6

Efficient programming

6.1 Some tips for programming

- `fortran` does not require an implicit declaration of variables, variable starting with `i,j,k,l,m,n` are automatically integers, `z` complex (?), the other are real (this default setting can be switch off by `implicit none`)
- use reasonable names for variables, usually `i,j,k,l,m,n` for integers,
- variable `res` is much better name for the residual than `abc`
- some mathematical formulas can not be directly coded (operation by operation) due to overflow, underflow, cancellation, e.g., `exp(-x)` can vanish for not so small `x`
- if you compute `x/y`, be sure that `y`$\neq 0$
- operations with integers are faster, e.g. difference `x**2.` and `x**2`
- be careful, difference `1/2` and `1./2`, if `n` is integer, then `n=1./2` can give `n=0`
- avoid too "long relations", if necessary split them into a few shorter ones
- minimize the number of mathematical operations, use temporary variables or arrays, e.g.

  ```
  ff = sin(a*sqrt(x) ) /  a*sqrt(x)
  ```

  is slower than

  ```
  ax = a*sqrt(x)
  ff = sin(ax ) / ax
  ```

- `sqrt(x)` is much slower than multiplication, avoid it in the loops, e.g.

```
do i=1, N
  ss = dot_product(a(:), r(:))
  if( sqrt (ss) <= tol ) exit loop
end do
```

is slower than

```
tol2 = tol*tol
do i=1, N
  ss = dot_product(a(:), r(:))
  if( ss <= tol2 ) exit loop
end do
```

- command `sqrt(x)` is much faster (and accurate) than `x**(0.5)`, even `sqrt(sqrt(x))` is usually faster than `x**(0.25)`
- matrices are "column-ordered" (Fortran) or "row-ordered" (C), so there is the difference between

```
do i=1,N
   do j=1,N
       a(i,j) = ....
   end do
end do
```

and

```
do j=1,N
   do i=1,N
       a(i,j) = ....
   end do
end do
```

The number of operations is the same but different speed, see below.

- The following commands return the "same" value, the first one is faster.

```
s = dot_product(a(:), b(:))
```

or

```
s = sum(a(:) * b(:) )
```

- There is also overhead associated with function and subroutine calls that, when nested in the innermost loops, can become significant. Avoid such subroutine calls by inlining or restructuring code.

- Overheads of cycles require also some time, e.g. For `n>m`,

```
do i = 1, n
    do j = 1, m
        ! ...
    end do
end do
```

  is less efficient than

```
do j = 1, m
    do i = 1, n
        ! ...
    end do
end do
```

- Another example

```
do i=1,N
   a(i) = ....
end do
```

  is (can be much) slower than

```
M = mod(N,5)
do i=1,M
   a(i) = ....
end do

M1 = M + 1
do i=M,N,5
   a(i) = ....
   a(i+1) = ....
   a(i+2) = ....
   a(i+3) = ....
   a(i+4) = ....
end do
```

  for large `M`.

6.2 The use of the cache memory

In order to code efficiently the numerical methods, it is necessary to know (a little) about the computers. A very simplified model of a computer is the following.

$$\boxed{\text{main memory}} \longleftrightarrow \boxed{\text{cache memory}} \longleftrightarrow \boxed{\text{registers}} \Longleftarrow \boxed{\text{processor}}$$

- *main (computer) memory* (also operation memory) contains all data of the code
- *cache memory* stores data so future requests for that data can be served **faster**; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere.
- *registers* are a part of memory which can be treated by processor
- *processor* performs the mathematical operations with data in register.

It means that the data stored in the main memory must be moved first to cache and then to register before it can be operated on. The transfer from the main memory to cache is much slower than the transfer from cache to registers and it is also slower than the rate at which the computer can perform arithmetic.

The amount of the cache memory and registers is limited (otherwise the price of the computers would be very high). So during the computation, the data are transferred from the computer memory to the cache memory and back many times. The transfer is faster when more data are moved at once. So if we have a part of the code

```
real :: a, b, c
a = 10.
b = 15.
c = a * b
```

then the command `c = a * b` is performed in such a way that not only the number `a` and `b` are transferred to the cache but also the parts of the memory near these variables. This effect can be used in the programming of numerical methods.

6.2.1 Standard matrix-matrix multiplication

Let us consider an examples where we have two matrices $\mathbb{A}, \mathbb{B} \in \mathbb{R}^{N\times N}$ and we need to compute its multiplication and the results stored in matrix $\mathbb{C} \in \mathbb{R}^{N\times N}$, i.e.,

$$\mathbb{A} = \{a_{i,j}\}_{i,j,=1}^{N},\ \mathbb{B} = \{b_{i,j}\}_{i,j,=1}^{N}, \quad \mathbb{C} = \{c_{i,j}\}_{i,j,=1}^{N},\ c_{i,j} = \sum_{k=1}^{N} a_{i,k}b_{k,j},\ i,j = 1,\ldots,N. \tag{6.1}$$

The number of operations is

$$N^2 \text{ times}(N \text{ multiplications } + (N-1) \text{ summations }) \approx 2N^3. \tag{6.2}$$

If N is high it is not possible to store all matrices elements in the cache memory. So they have to be transferred to the cache memory in several steps.

A simple part of a code for the matrix multiplication is the following.

```
do i = 1, n
 do j = 1, n
    C(i,j) = 0
    do k = 1, n
       C(i,j) = C(i,j) + A(i,k)*B(k,j)    ! multiplication line
    end do
 end do
end do
```

The transfers to and from the cache memory is done in `multiplication line`. So also other entries of $\mathbb{A}$ and $\mathbb{B}$ are moved to cache (and not used in this step). The number of transfers is too high.

Let us assume that $2N$ values can be stored in the cache memory. In order to compute $c_{i,j}$ we have to transfer $2N$ values (N row-entries of $\mathbb{A}$ and N column-entries of $\mathbb{B}$) to cache and then perform $2N$ flops (mathematical operations). For the evaluation of $c_{i,j+1}$, N values of the i-th row of matrix $\mathbb{A}$ can be kept in cache and next N values (N column-entries of $\mathbb{B}$) have to be transfer to cache. Then we perform $2N$ flops. Hence, we have the ratio between memory manipulation and computing operations

$$\frac{\#\,\text{mathematical operations (flops)}}{\#\,\text{transfers to cache}} = \frac{2N}{N} = 2.$$

6.2.2 Block matrix operations

Let $N = Mn$ where M and n are integers. Then we have

$$\mathbb{A} = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} & a_{1,n+1} & \dots & a_{1,2n} & \dots & a_{1,N-n+1} & \dots & a_{1,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & \dots & a_{n,n} & a_{n,n+1} & \dots & a_{n,2n} & \dots & a_{n,N-n+1} & \dots & a_{n,N} \\ a_{n+1,1} & \dots & a_{n+1,n} & a_{n+1,n+1} & \dots & a_{n+1,2n} & \dots & a_{n+1,N-n+1} & \dots & a_{n+1,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{2n,1} & \dots & a_{2n,n} & a_{2n,n+1} & \dots & a_{2n,2n} & \dots & a_{2n,N-n+1} & \dots & a_{2n,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

which can be re-written in the block structure

$$\mathbb{A} = \begin{pmatrix} \mathbb{A}_{1,1} & \mathbb{A}_{1,2} & \dots & A_{1,M} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbb{A}_{M,1} & \mathbb{A}_{M,2} & \dots & A_{M,M} \end{pmatrix},$$

where $\mathbb{A}_{kl}$ are $n \times n$ matrices given by

$$\mathbb{A}_{kl} = \{a_{(k-1)n+i,(l-1)n+j}\}_{i,j=1}^{n}, \quad k,l = 1,\ldots,M.$$

Using similar notation for $\mathbb{B}$ and $\mathbb{C}$ we have

$$\mathbb{C}_{ij} = \sum_{k=1}^{M} \mathbb{A}_{ik}\mathbb{B}_{kj}, \quad i,j = 1,\ldots,M. \tag{6.3}$$

When we modify the previous code for the block structure in such a way that the size of matrix block corresponds to the amount of memory transferred *at once* to the cache, we can save a lot of communications (and therefore the computational time).

In both cases, the number of mathematical operations is the same.

However, if we can stores 3 blocks in cache then we transfer $3n^2$ entries (blocks $\mathbb{C}_{ij}$, $\mathbb{A}_{ik}$ and $\mathbb{B}_{kj}$). In order to perform the computation ($\mathbb{C}_{ij} = \mathbb{C}_{ij} + \mathbb{A}_{ik}\mathbb{B}_{kj}$) we carry out $2n^3$ operations (flops). Hence, we have the ratio between memory manipulation and computing operations

$$\frac{\#\,\text{mathematical operations (flops)}}{\#\,\text{transfers to cache}} = \frac{2n^3}{3n^2} = \frac{2}{3}n,$$

which is increasing for increasing n (increasing size of cache memory). The larger the block are, the less significant the data transfers become.

Starting from Fortran90, there is a function `matmul` which employs the cache memory in an optimal way. The previous part of the code reads

```
C(1:N, 1:N) = matmul(A(1:N, 1:N), B(1:N, 1:N) )
```

or simply

```
C(:, :) = matmul(A(:, :), B(:, :) )
```

or only

```
C = matmul(A, B )
```

However, in both latter cases there can not be any verification of the sizes of arrays, so it is not too much recommended.

6.2.3 Programming of numerical methods

6.2.4 Verification of codes

6.2.5 Norms of programming languages, transportability

Homeworks

Exercise 11. *Let us consider matrices* $\mathbb{A}$ *and* $\mathbb{B}$ *such that*

$$a_{i,j} = i + j, \quad b_{i,j} = i - j, \quad i,j = 1,\ldots,N. \tag{6.4}$$

This is a standard test case since the trace of the resulting matrix $\mathbb{C} = \mathbb{AB}$ *is vanishing. Using the code* `multi.f90` *test three variants of the matrix-matrix multiplications:*

- *simple multiplications* (6.1),
- *block multiplications* (6.3) *with different* n*, the default size is* $n = 40$,
- *multiplications using function* `matmul`.

Use different sizes of N*. This codes measures the used computational time in seconds and gives the speed of computations in Mflops* $= 2N^3/time/1E + 06$.

Chapter 7

LAPACK

LAPACK is a library of Fortran (Fortran 90) with subroutines for solving the most commonly occurring problems in numerical linear algebra. It is freely-available software, and is copyrighted.

LAPACK is available on netlib and can be obtained via the World Wide Web and anonymous ftp.

```
http://www.netlib.org/lapack/
```

The distribution tar file contains the Fortran source for LAPACK and the testing programs. It also contains the Fortran77 reference implementation of the Basic Linear Algebra Subprograms (the Level 1, 2, and 3 BLAS) needed by LAPACK. However this code is intended for use only if there is no other implementation of the BLAS already available on your machine; the efficiency of LAPACK depends very much on the efficiency of the BLAS.

History of versions

```
 VERSION 1.0   :  February 29, 1992
VERSION 1.0a  :  June 30, 1992
VERSION 1.0b  :  October 31, 1992
VERSION 1.1   :  March 31, 1993
VERSION 2.0   :  September 30, 1994
VERSION 3.0   :  June 30, 1999
VERSION 3.0 + update :  October 31, 1999
VERSION 3.0 + update :  May 31, 2000
VERSION 3.1   : November 2006
VERSION 3.1.1 : February 2007
VERSION 3.2   : November 2008
VERSION 3.2.1 : April 2009
VERSION 3.2.2 : June 2010
VERSION 3.3.0 : November 2010
```

7.1 Instalation

Download the library, e.g., the file `lapack-3.3.0.tgz` and unpack it by

```
tar xfz lapack-3.3.0.tgz
```

the directiry with archive `lapack-3.3.0` appears.

First you need to create file `make.inc`, the simplest way is

```
cd lapack-3.3.0
cp make.inc.example make.inc
```

and edit (if necessary) the resulting file.

The you create the `BLAS` library by

```
cd BLAS/SRC
make
cd ../../
```

the library `blas_LINUX.a` appears in the directory `lapack-3.3.0`. Then translate LAPACK library by

```
cd SRC
make
cd ../
```

the library `lapack_LINUX.a` appears in the directory `lapack-3.3.0`, i.e., command `ls -l *.a` results, e.g.,

```
-rw-r--r-- 1 dolejsi dolejsi  1358568 kvě  5 07:54 blas_LINUX.a
-rw-r--r-- 1 dolejsi dolejsi 18074296 kvě  5 07:56 lapack_LINUX.a
```

Alternatively, you can use command `make` directly in the directory `lapack-3.3.0` then tests of LAPACK subroutines are carried out (this is more time consuming of course).

7.2 Naming Scheme of BLAS and LAPACK subroutines

For detailed description see

`http://www.netlib.org/blas`
`http://www.netlib.org/lapack`

The name of each LAPACK routine is a coded specification of its function. All driver and computational routines have names of the form `XYYZZZ`, where for some driver routines the 6th character is blank.

The first letter, `X`, indicates the data type as follows:

```
S  REAL
D  DOUBLE PRECISION
C  COMPLEX
Z  COMPLEX*16 or DOUBLE COMPLEX
```

The next two letters, YY, indicate the type of matrix (or of the most significant matrix). Most of these two-letter codes apply to both real and complex matrices; a few apply specifically to one or the other, as indicated in the following table:

```
BD  bidiagonal
DI  diagonal
GB  general band
GE  general (i.e., unsymmetric, in some cases rectangular)
GG  general matrices, generalized problem (i.e., a pair of general matrices)
GT  general tridiagonal
HB  (complex) Hermitian band
HE  (complex) Hermitian
HG  upper Hessenberg matrix, generalized problem (i.e a Hessenberg and a
   triangular matrix)
HP  (complex) Hermitian, packed storage
HS  upper Hessenberg
OP  (real) orthogonal, packed storage
OR  (real) orthogonal
PB  symmetric or Hermitian positive definite band
PO  symmetric or Hermitian positive definite
PP  symmetric or Hermitian positive definite, packed storage
PT  symmetric or Hermitian positive definite tridiagonal
SB  (real) symmetric band
SP  symmetric, packed storage
ST  (real) symmetric tridiagonal
SY  symmetric
TB  triangular band
TG  triangular matrices, generalized problem (i.e., a pair of triangular matrices)
TP  triangular, packed storage
TR  triangular (or in some cases quasi-triangular)
TZ  trapezoidal
UN  (complex) unitary
UP  (complex) unitary, packed storage
```

The last three letters ZZZ indicate the computation performed, see,

```
http://www.netlib.org/lapack/lug/node26.html#tabdrivelineq
```

For example, SGEBRD is a single precision routine that performs a bidiagonal reduction (BRD) of a real general matrix.

Example 7.1. *Subroutine* daxpy *from BLAS:* $y = y + ax$*, where* $x, y \in \mathbb{R}^N$*,* $a \in \mathbb{R}$*:*

```
        SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)
*     .. Scalar Arguments ..
      DOUBLE PRECISION DA
      INTEGER INCX,INCY,N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION DX(*),DY(*)
*     ..
*
*  Purpose
*  =======
*
*     DAXPY constant times a vector plus a vector.
*     uses unrolled loops for increments equal to one.
*
*  Further Details
*  ===============
*
*     jack dongarra, linpack, 3/11/78.
*     modified 12/3/93, array(1) declarations changed to array(*)
*
*  =====================================================================
*
*     .. Local Scalars ..
      INTEGER I,IX,IY,M,MP1
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC MOD
*     ..
      IF (N.LE.0) RETURN
      IF (DA.EQ.0.0d0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) GO TO 20
*
*        code for unequal increments or equal increments
*          not equal to 1
*
      IX = 1
      IY = 1
      IF (INCX.LT.0) IX = (-N+1)*INCX + 1
```

```
      IF (INCY.LT.0) IY = (-N+1)*INCY + 1
      DO 10 I = 1,N
          DY(IY) = DY(IY) + DA*DX(IX)
          IX = IX + INCX
          IY = IY + INCY
   10 CONTINUE
      RETURN
*
*        code for both increments equal to 1
*
*
*        clean-up loop
*
   20 M = MOD(N,4)
      IF (M.EQ.0) GO TO 40
      DO 30 I = 1,M
          DY(I) = DY(I) + DA*DX(I)
   30 CONTINUE
      IF (N.LT.4) RETURN
   40 MP1 = M + 1
      DO 50 I = MP1,N,4
          DY(I) = DY(I) + DA*DX(I)
          DY(I+1) = DY(I+1) + DA*DX(I+1)
          DY(I+2) = DY(I+2) + DA*DX(I+2)
          DY(I+3) = DY(I+3) + DA*DX(I+3)
   50 CONTINUE
      RETURN
      END
```

Let us note that the loop with label 50 is more efficient than

```
       DO 50 I = MP1,N
          DY(I) = DY(I) + DA*DX(I)
   50 CONTINUE
```

7.3 Link of LAPACK with your own code

Example of my code (in file `lap_sub.f90`) using LAPACK subroutines `dgetri`, `dgetrf`:

```
   subroutine  MblockInverse(n, A)
    integer, intent(in) :: n
```

```
    real, dimension(1:n,1:n), intent(inout) :: A
    external:: dgetri, dgetrf          ! subroutines from LAPACK
    real, dimension(:), allocatable :: ident, work
    integer :: info, iwork

    iwork = 100 * 30
    allocate(ident(1:n), work(1:iwork) )
    ident(:) = 1.

    call DGETRF(n, n, A, n, ident, info )
    if(info /= 0 ) print*,'Problem 1 in MblockInverse in matrix.f90 ', info
    if(info /= 0 ) stop

    call DGETRI(n, A, n, ident, work,  iwork, info )
    if(info /= 0 ) print*,'Problem 2 in MblockInverse in matrix.f90 ', info
    if(info /= 0 ) stop

    deallocate(ident, work)

  end subroutine MblockInverse
```

Example of my `Makefile`:

```
TARGETS= lap_sub.o geom.o integ.o f_mapping.o  main.o

FFLAGS=  -fPIC -fdefault-real-8 -O2 -w

LIBS= lapack-3.3.0/lapack_Linux.a lapack-3.3.0/blas_Linux.a

FXX=gfortran

all: Adgfem

Adgfem: $(TARGETS)
        $(FXX) $(FFLAGS) -o Adgfem $^ $(LIBS)

clean:
        -rm -f Adgfem  *.o *.mod

%.o:%.f90
        $(FXX) $(FFLAGS) -c $?
```

For globally installed library

```
LIBS=-llapack
```

in this case translator seeks files

```
/usr/lib/libblas.a
/usr/lib/liblapack.a
```

Chapter 8

Fundamentals of adaptations

We consider the model problem (1.2) and its numerical approximation (1.3):

$$u \in X: \ \mathscr{P}(d;u) = 0, \qquad u_h \in X_h: \ \mathscr{P}_h(d_h;u_h) = 0.$$

Main goal: solve numerically the given problem such that

- the error (its estimate) is under the given tolerance (accuracy)
- the computational time is as short as possible (efficiency)

Since the requirements of the accuracy and the efficiency are opposite in some sense they should be well balanced. The only way, how to fulfil both requirements, is a suitable choice of the space X_h. In many applications, the optimal choice of X_h is not known a priori. Thus we have to adapt the space X_h based on the previous computations. We speak about about the local adaptation or enhancement of the space X_h.

Example 8.1. *Adaptivity appears in many standard computations:*

- *numerical integration:* $\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^N w_i f(x_i)$, *how to choose nodes and weights?*
- *numerical solution of ODEs:* $y'(t) = f(t, y(t))$, *Euler method* $y_{k+1} = y_k + \tau_k f(t_k, y_k)$, *how to choose* τ_k*?*
- *numerical solution of PDEs:* $\mathcal{D}u = f$, *FEM, adaptive choice of mesh* $\mathscr{T}_h$ *and finite element space* V_h

In order to adapt the space X_h such that the accuracy as well as the efficiency are achieved, we need

- a posteriori error estimates which are able the estimate the computational error locally,
- adaptive strategy which adapt efficiently the space X_h based on a posteriori error estimates.

Remark 8.2. *In this section, we deal with a posteriori error estimates depending of the approximate solution u_h. However, we remind that the approximate solution u_h is not available in practice, we have only u_h^*. Therefore, it would be also desirable to have a posterior error estimates taking into account this aspect.*

8.1 Error and its property

8.1.1 Types of measuring of the error

Our goal is to fulfil the condition

$$e_h := \|u - u_h\| \leq \mathrm{TOL}, \tag{8.1}$$

where u and u_h are the exact and the approximate solutions, $\|\cdot\|$ is a suitable norm on $\cup_{h\in(0,h_0)} X_h \cup X$ and $\mathrm{TOL} > 0$ is the given tolerance.

Or, in many physical applications, the goal is to evaluate a functional $J : \cup_{h\in(0,h_0)} X_h \cup X \to \mathbb{R}$ depending on the (approximate) solution, e.g., lift coefficient in aerodynamics. Then (8.1) has to be replaced by

$$e_h := |J(u) - J(u_h)| \leq \mathrm{TOL}', \tag{8.2}$$

see Chapter 18.

8.1.2 Localisation of the error

We assume that the error e_h is localisable. It means that for each element K of the mesh $\mathscr{T}_h$ we can define the error $e_{h,K}$ over K which satisfies the following: there exists an invertible function $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ such that

$$f(e_h) = \sum_{K\in\mathscr{T}_h} f(e_{h,K}) \tag{8.3}$$

$$\iff \quad e_h = f^{-1}\left(\sum_{K\in\mathscr{T}_h} f(e_{h,K})\right). \tag{8.4}$$

Example 8.3. *The standard norms on the Lebesgue spaces $L^q(\Omega)$ and on the Sobolev spaces $W^{k,q}(\Omega)$, $q \in [1,\infty)$ are localisable since*

$$e_h^q = \|u - u_h\|^q = \sum_{K\in\mathscr{T}_h} \|u - u_h\|_K^q = \sum_{K\in\mathscr{T}_h} e_{h,K}^q. \tag{8.5}$$

Then the value $e_{h,K}$, $K \in \mathscr{T}_h$ is called dicretization error on element K.

8.2 A posteriori error estimates

We recall the discrete problem (1.3) written as

$$\mathscr{P}_h(d_h; u_h) = 0, \tag{8.6}$$

where $u_h \in X_h$ is the approximate solution, d_h represents the discrete variants of the data, $\mathscr{P}_h : Z_h \times X_h \to Y_h$ is a given mapping representing the numerical method and X_h, Y_h and Z_h are finite dimensional normed vector spaces.

The numerical analysis usually provides a priori error estimates, i.e., an inequality

$$e_h \leq c(u)h^\alpha \tag{8.7}$$

where h is the parameter of the discretization (= maximal size of the mesh elements particularly), e_h is the error given by (8.1) or (8.2), u is the exact solution of (1.2), and $c(u)$ is a function dependent on the (unknown) exact solution u but independent of h.

The estimate (8.7) gives information about the rate of the convergence for $h \to 0$. However, it does not provide information about the real size of the error since u (a hence $c(u)$) are unknown and moreover, even in case when u is known, the right-hand side of (8.7) overestimate the error, i.e., $e_h \ll c(u)h^\alpha$.

For practical applications, it is advantageous to derive a posteriori error estimate, i.e. the inequality

$$e_h \leq \eta(u_h), \tag{8.8}$$

where η is the total error estimator depending on u_h (and not on u). Usually, the error estimator η can be express as

$$\eta(u_h) = f^{-1}\left(\sum_{K\in\mathscr{T}_h} f\left(\eta_K(u_h)\right)\right), \tag{8.9}$$

where η_K is the estimator of the local error over the element $K \in \mathscr{T}_h$ and f is the function from (8.3). I.e., in case of Hilbertian norm we have

$$\left(\eta(u_h)\right)^2 = \sum_{K\in\mathscr{T}_h} \left(\eta_K(u_h)\right)^2. \tag{8.10}$$

Two main aims of a posteriori error estimation are to enable

- error control – to achieve the user-specified precision of the computation,
- efficient computations – to avoid exploiting computational resources where it is not reasonable.

In order to provide the previous aims, it is desirable that a posterior error estimates satisfy

(A1) guaranteed error estimate – the estimate (8.8) is valid without any unknown constant, i.e., $\eta(u_h)$ depends only on the input data problem and the approximate solution itself. Therefore, we can check if the prescribed tolerance was achieved.

(A2) local efficiency – the lower bound on the error locally up to a generic constant., i.e.,

$$\eta_K(u_h) \leq C \sum_{K' \in D(K)} e_{h,K'} \qquad \forall K \in \mathscr{T}_h, \tag{8.11}$$

where $\|\cdot\|_{D(K)}$ is a norm over a (small) union of the mesh elements near K.

(A3) asymptotic exactness – the ratio of the actual error and its estimator should approach to one as $h \to 0$,

(A4) robustness – the previous properties 1) - 3) are independent of parameters of the problem. This feature is of great importance in order that the error estimates could be applicable to a wide range of problems.

(A5) low evaluation cost – the evaluation of η_K, $K \in \mathscr{T}_h$ should be fast in the comparison to the solution of the problem itself.

We note that it is difficult to achieve all aspects mentioned above. Although, there exist many works dealing with a posteriori error analysis of many numerical methods and various model problem, the goals (A1) – (A5) are not achieved for general non-linear problem generally.

Hence, in many situation we employ estimates such that

$$e_h \approx \eta(u_h) \qquad \& \qquad e_{h,K} \approx \eta_K(u_h), \quad K \in \mathscr{T}_h \tag{8.12}$$

Thus we have no guaranty of the error (condition (A1) is violated) but we can achieve the efficiency.

8.3 Adaptive strategies

We assume that we solve the problem (1.2) with the aid of a numerical method formally written in the form (1.3). Moreover, we assume that the numerical method as well as its computer implementation are convergent. It means that with the aid of a sufficient adaptation (refinement) of the space X_h, the size of the computational error e_h is under any tolerance TOL where

$$\mathrm{TOL} \gg \epsilon_{\mathrm{mach}} \|u\|. \tag{8.13}$$

We assume that TOL be a given tolerance satisfying (8.13). The ultimate goal is to adapt the space X_h such that

- the computational error is under the given tolerance, i.e,

$$e_h \approx \eta(u_h) \leq \mathrm{TOL}, \tag{8.14}$$

- the number of degree of freedom ($= \dim X_h$) is minimal (or at least small).

A general idea, how to achieve these goals, is a performance of several adaptive cycles, where the elements having too high error estimates are refined. There exists several basic adaptation strategies. For simplicity we assume, that the norm $\|\cdot\|$ satisfies (8.5) for some $q \geq 1$.

- local strategy: Instead of (8.14), we consider a stronger requirement

$$e_{h,K} \approx \eta_K(u_h) \leq \mathrm{TOL} \left(\frac{|K|}{|\Omega|} \right)^{1/q} \quad \forall K \in \mathscr{T}_h, \tag{8.15}$$

  where $|K|$ and $|\Omega|$ denotes measures of K and Ω, respectively. Obviously, if (8.15) is valid for all $K \in \mathscr{T}_h$ then (8.14) is valid. Hence, all elements K for which the inequality (8.15) is violated are refined. We note that relation (8.15) can be modified, e.g.,

$$e_{h,K} \approx \eta_K(u_h) \leq \mathrm{TOL} \left(\frac{1}{\#\mathscr{T}_h} \right)^{1/q} \quad \forall K \in \mathscr{T}_h, \tag{8.16}$$

  where $\#\mathscr{T}_h$ is the number of elements of $\mathscr{T}_h$.

- global strategy: If the condition (8.14) is not valid then then we select elements with the higest η_K for refinement. There are several basic strategies for selecting elements for refinement, e.g.,

  i) we select all $K \in \mathscr{T}_h$ such that

$$\eta_K \geq C' \max_{K' \in \mathscr{T}_h} \eta_{K'},$$

  where $C' \in (0,1)$ is a suitably chosen constant, e.g., $C' = 0.9$.

  ii) we select a fixed percent of elements (e.g., 10%) with the highest value η_K,

Do Exercise 12

8.4 Optimal solution strategy

The ultimate goal is to **solve problem** (1.2) **by the numerical method** (1.3) **in such a way that condition** (8.1) **is satisfied in the shortest possible computational time.** The solution strategy, which fulfils this goal is called the optimal solution strategy. Often, the requirement "the shortest possible computational time" is replaced by "the smallest possible number of degrees of freedom". Obviously, these two requirements are not equivalent.

An achievement of this goal is very complicated, in many situations almost impossible. The problem of developing the optimal solution strategy is complex, it consists of several (very ofter opposite) aspects. In order to demonstrate its complexity, we consider the following example. In the space-time cylinder $\Omega \times (0,T)$, we consider a time-dependent nonlinear partial differential equation, which is discretized by the finite element method in space and the implicit multi-step method in time. Hence, at each time level, we solve a nonlinear algebraic system by the Newton method and the linear system arising at each Newton iteration by the GMRES method with ILU preconditioning.

Hence, the superscript ^{k} denotes the index of the time step, the superscript ^{i} denotes the Newton iteration and the superscript ^{l} denotes the GMRES iteration.

Then, the solution strategy can be written in this form:

1. set $k := 0$, $t_k := 0$,
2. propose the initial mesh $\mathscr{T}_h^0$, time step τ_0 and an approximation of the initial solution u_h^0,
3. if $t_k = T$ then stop the computation
   else preform the time step $t_k \to t_{k+1}$ using the time step τ_k and the mesh $\mathscr{T}_h^k$,
   (a) set $i = 0$, $u_h^{k+1,0} := u_k^k$,
   (b) perform the i^{th} Newton step $u_h^{k+1,i} \to u_h^{k+1,i+1}$ by
      i. set $l = 0$, $u_h^{k+1,i+1,0} := u_h^{k+1,i}$
      ii. perform l^{th} GMRES step with ILU preconditioner starting from $u_h^{k+1,i+1,l}$
      iii. if linear algebraic criterion is satisfied
      then put $u_h^{k+1,i+1} := u_h^{k+1,i+1,l}$ and go to step (c)
      else put $u_h^{k+1,i+1,l+1} := u_h^{k+1,i+1,l}$, $l := l + 1$ go to step ii.
   (c) if non linear algebraic criterion is satisfied
   then put $u_h^{k+1} := u_h^{k+1,i+1}$ and go to step 4.
   else put $u_h^{k+1,i+1} := u_h^{k+1,i}$, $i := i + 1$ go to step (b)
4. estimate computational error
5. if the error estimate is under tolerance
   then propose new time step τ_{k+1}, new mesh $\mathscr{T}_h^{k+1}$, put $t_{k+1} := t_k + \tau_k$, $k := k + 1$ go

to step 2.
else propose better time step τ_k or better mesh $\mathscr{T}_h^k$ and repeat step 2.

Homeworks

Exercise 12. *Propose some examples where each of the adaptive strategy mentioned above makes some troubles, i.e., either the strategy can not achieve the given stopping criterion or many adaptive cycles is required until the stopping is achieved.*

Chapter 9

Numerical integration

The aim is numerically evaluate the (exact) value

$$I(f) = \int_a^b f(x)\,\mathrm{d}x, \tag{9.1}$$

where $f \in L^1(a,b)$ is given. The function f can be given at discrete nodes, then we have to interpolate it (e.g., using a spline).

The value $I(f)$ is evaluated by a quadrature formula (or numerical quadrature or quadrature rule)

$$Q_n(f) := \sum_{i=1}^{n} w_i f(x_i), \qquad R_n := I(f) - Q_n(f), \tag{9.2}$$

where $x_i \in [a,b],\ i = 1,\dots,n$ are the nodes, $w_i \in \mathbb{R},\ i = 1,\dots,n$ are the weights, $Q_n(f)$ is the approximation of $I(f)$ by the numerical quadrature and $R_n \in \mathbb{R}$ is the remainder (or the discretization error).

The weights and the nodes have to be chosen such that the remainder R_n is sufficiently small and the computation computational costs are low. The efficiency is measured by the number of nodes $(= n)$. Usually, we chose the nodes $x_i,\ i = 1,\dots,n$, the function f is approximated by a polynomial function (using values $f(x_i),\ i = 1,\dots,n$) and the resulting polynomial function is integrated analytically. Except the value $Q_n(f)$ we need to approximate the remainder R_n.

Definition 9.1. *We say that the quadrature formula has the order k if $I(q) = Q_n(q)$ for any polynomial function q of degree at most k.*

Remark 9.2. *In practice, the composite formulae are employed. This means that the interval $[a,b]$ is split onto mutually disjoint closed sub-intervals*

$$I_j = [y_{j-1}, y_j],\ j = 1,\dots,N \text{ such that } a = y_0 < y_1 < y_2 < \cdots < y_N = b.$$

Obviously, $I(f) = \sum_{j=1}^{N} \int_{I_j} f(x)\,\mathrm{d}x$. Then the quadrature formula is applied to each interval $I_j,\ j = 1,\dots,N$ separately. If f is sufficiently regular then $R_n = O(h^{p+1})$ where $h \approx y_j - y_{j-1}$ for $j = 1,\dots,N$.

9.1 Newton-Cotes quadrature formulae

The well-known Newton-Cotes quadrature formulae are defined in the following way. The nodes $x_i,\ i = 1, \ldots, n$ are equidistantly distributed in $[a, b]$, i.e.,

$$x_i := a + i\,\frac{b-a}{n-1}, \quad i = 1, \ldots, n \qquad \text{for } n > 1,$$
$$x_1 := \frac{a+b}{2} \qquad \text{for } n = 1.$$

Then the weights $w_i,\ i = 1, \ldots, n$ are chosen such that the resulting quadrature formula is exact (i.e., $R_n = 0$) for the polynomials of the highest possible degree.

This task leads to the system of linear algebraic equations which can be solved analytically. The weights are rational numbers. Obviously, the pairs $(x_i, f(x_i)),\ i = 1, \ldots, n$ uniquely define the polynomial function of degree $n-1$. Thus the Newton-Cotes formulae has the order at least $n-1$. Moreover, it is possible to show that the order of the Newton-Cotes formula is equal to

$$\begin{array}{rl} n & \text{for } n \text{ odd,} \\ n-1 & \text{for } n \text{ even.} \end{array}$$

Finally, we present some properties of the Newton-Cotes formulae.

- The Newton-Cotes formulae are the so-called closed formulae since $x_1 = a$ and $x_n = b$ (for $n > 1$). This is problematic in situation where f has a singularity at $x = a$ or $x = b$, e.g., $\int_0^1 x^{-1/2}\,\mathrm{d}x$.
- For large n ($n \gtrsim 10$), the Newton-Cotes formulae do not work properly, since the corresponding interpolation polynomial function oscillates.
- The efficiency of the Newton-Cotes formulae is not optimal, the order is not the highest possible for the given n.

9.1.1 Error estimation

The remainder R_n is usually estimated by a comparing of the results $Q_n(f)$ and $Q_{n'}(f)$ of two computations. It is possible to apply

- two different quadrature formulae on the same partition,
- one quadrature formulae on two different partitions.

Error estimation using two different quadrature formulae

Let us consider the Newton-Cotes formulae for $n = 1, 2, 3$, namely

$$M(f) := (b-a)\, f\left(\frac{a+b}{2}\right) \qquad \textbf{m}\text{idpoint formula},$$

$$T(f) := \frac{b-a}{2}\,(f(a)+f(b)) \qquad \textbf{t}\text{rapezoid formula},$$

$$S(f) := \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right) \qquad \textbf{S}\text{impson formula}.$$

The midpoint and the trapezoid formulae have the order 1 and the Simpson formula has the order 3.

Let $m := \frac{a+b}{2}$. Let $f \in C^4([a,b])$, the Taylor expansion at m reads

$$\begin{aligned} f(x) =& f(m) + f'(m)(x-m) + \frac{1}{2}f''(m)(x-m)^2 + \frac{1}{6}f'''(m)(x-m)^3 \\ &+ \frac{1}{24}f''''(m)(x-m)^4 + \dots. \end{aligned} \tag{9.3}$$

Integrating of (9.3) over (a,b) gives (the "even" terms disappears)

$$\begin{aligned} I(f) &= f(m)(b-a) + \frac{1}{24}f''(m)(b-a)^3 + \frac{1}{1920}f''''(m)(b-a)^5 + \dots \\ &=: M(f) \quad + \quad E \quad + \quad F \quad + \dots. \\ &= M(f) + E + F + \dots. \end{aligned} \tag{9.4}$$

Moreover, we put $x := a$ and $x := b$ in (9.3) and then we sum both relations, which gives (again the "even" terms disappears)

$$f(a)+f(b) = 2f(m) + \frac{2}{2}f''(m)\frac{(b-a)^2}{4} + \frac{2}{24}f''''(m)\frac{(b-a)^4}{16} + \dots. \tag{9.5}$$

Multiplying (9.5) by $(b-a)/2$ implies

$$\frac{f(a)+f(b)}{2}(b-a) = f(m)(b-a) + \frac{1}{8}f''(m)(b-a)^3 + \frac{1}{384}f''''(m)(b-a)^5 + \dots. \tag{9.6}$$

which can be rewritten (using the notation from (9.4)) as

$$T(f) = M(f) + 3E + 5F + \dots. \tag{9.7}$$

Finally, applying (9.4) again we have

$$I(f) = T(f) - 2E - 4F + \dots. \tag{9.8}$$

From (9.4) and (9.8) we conclude that if $F \ll E$ then the error of the trapezoid formula is two times higher then the error of the midpoint formula. Furthermore, if $F \ll E$ then (9.7) gives

$$E \approx \frac{T(f) - M(f)}{3}. \tag{9.9}$$

Therefore, the dominant part of the error E of the midpoint formula can be approximated by one third of the difference of $T(f)$ and $M(f)$. Hence, we have estimated the error by a difference of two quadrature formulae.

Do Exercise 13

Finally, adding of two thirds of (9.4) and one third of (9.8) gives

$$I(f) = \frac{2}{3}M(f) + \frac{1}{3}T(f) - \frac{2}{3}F + \cdots = S(f) - \frac{2}{3}F + \dots. \tag{9.10}$$

Hence, we derived the Simpson formula by an alternative way.

Remark 9.3. *In (9.10), we combined two first order methods and obtained a third order method. This is a general approach, where two results (employing for the error estimation) are further used for obtaining of a more accurate results.*

Error estimation using the half-size step method

Using the notation from (9.4), we have

$$I(f) = M_h(f) + E_h + F_h + \dots, \tag{9.11}$$

where the subscript h denotes the size of the interval $[a, b]$. Now we split the interval $[a, b]$ on two sub-intervals having the same size, apply the previous procedure separately for each sub-interval and sum the results. Then we have

$$I(f) = M_{h/2}(f) + E_{h/2} + F_{h/2} + \dots. \tag{9.12}$$

Symbol $M_{h/2}(f)$ denotes the approximate value of $I(f)$ computed by composite mid-point rule on both sub-intervals, similarly $E_{h/2}$ and $F_{h/2}$ represent the corresponding error. Since $E_h = O(h^3)$ then $E_{h/2} \approx \frac{1}{4}E_h$ ($(1/2)^3 = 1/8$ but we have two sub-intervals and thus $2 \cdot 1/8 = 1/4$). Then we have from (9.11) and (9.12) the estimate

$$E_{h/2} \approx \frac{1}{3}(M_{h/2}(f) - M_h(f)). \tag{9.13}$$

Generally, for a method of order p, the error estimate of the composite quadrature Q_h by the half-step size method reads

$$e_{h/2} \approx \frac{Q_{h/2}(f) - Q_h(f)}{2^{p+1} - 1}. \tag{9.14}$$

9.2 Gauss formulae

The well-known Gauss quadrature formulae are defined in the following way. The nodes $x_i \in [a,b],\ i = 1,\dots,n$ and the weights $w_i,\ i = 1,\dots,n$ are chosen in such a way that the resulting quadrature formula is exact (i.e., $R_n = 0$) for the polynomials of the highest possible degree.

Since $x_i,\ w_i,\ i = 1,\dots,n$ represent $2n$ degrees of freedom, we expect that the Gauss formulae are exact for polynomials of degree $2n-1$ (=order of accuracy). The derivation of the Gauss formulae leads to nonlinear algebraic systems. The weights and the nodes are generally irrational numbers.

Example 9.4. *The two-points Gauss formula is*

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right). \tag{9.15}$$

Do Exercise 14

Remark 9.5. *The Gauss formulae are usually derived by the Legendre polynomials which are orthogonal with respect $L^2((-1,1))$-scalar product. Thus, in many textbooks, the Gauss formulae are derived for $a = -1$ and $b = 1$. For a general $[a,b]$, the nodes and weights have to be transformed.*

Do Exercise 15

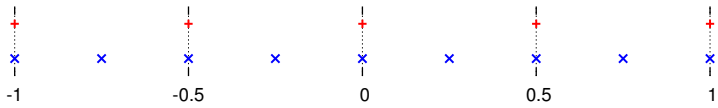
The basic properties of the Gauss formulae are the following.

- The Gauss formulae are the so-called open formulae since $x_1 \neq a$ and $x_n \neq b$. This is advantageous in situation where f has a singularity at $x = a$ or $x = b$, e.g., $\int_0^1 x^{-1/2}\,\mathrm{d}x$.
- The Gauss formulae have the highest possible order for the given n, hence their efficiency is optimal. Hence a high order is achieved by a small number of nodes, we avoid oscillations of a polynomial interpolation.
- The nodes are placed non-equidistantly which may be problematic for a use of the error estimation techniques from Section 9.1.1.

9.2.1 Error estimation

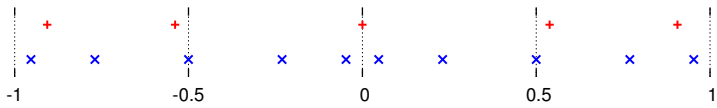
The remainder R_n is usually estimated by a comparing of the results two different computations. However, since the nodes of the Gauss formulae are placed non-equidistantly, the use of the error estimation techniques from Section 9.1.1 is not efficient.

The following figure shows the Newton-Cotes nodes for $n = 5$ and $a = -1,\ b = 1$ (red nodes) and the corresponding nodes used for the error estimation using the half-size step method (blue nodes).



We see that all red nodes are used also as the blue ones, hence we can save some computational costs.

On the other hand, the following figure shows the same situation for the Gauss nodes.



Obviously, the red and the blue nodes do not coincide, hence any computation can not be saved. Hence, we can conclude that the error estimation by the half-size step method is inefficient for the Gauss formulae.

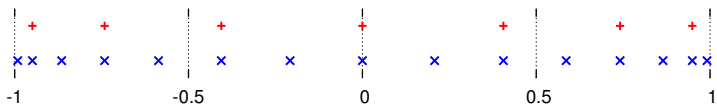
Remark 9.6. *A similar observation can be achieved also for the error estimation based on a comparison of two different quadrature formulae on the same partition.*

Gauss-Kronrod quadrature formulae

One possibility, how to achieve the efficiency of the error estimation are the so-called Kronrod quadrature formulae. Let $n > 1$ be given, and let G_n denote the Gauss quadrature formula. Then we construct the Kronrod quadrature formulae K_{2n+1} having $2n + 1$ nodes in the following way:

- all nodes from G_n are also the nodes of K_{2n+1},
- we add $n + 1$ additional nodes and chose the weights $w_i,\ i = 1, \dots, 2n + 1$ such that the resulting quadrature formula has the maximal possible order.

It is possible show that the formula K_{2n+1} has the order $3n + 1$. (The order of G_{2n+1} is $4n + 1$.) The following figure shows the nodes of both formulae G_7 (red nodes) and K_{15} (blue nodes).



Obviously, all red nodes are used also as the blue ones, hence we can save some computational costs.

The pair of formulae $G_n\, K_{2n+1}$ is used for the error estimation of the Gauss formula G_n is estimated (see [Pat69]) in the form

$$(200|G_n - K_{2n+1}|)^{3/2}\,. \tag{9.16}$$

The use of $G_n\, K_{2n+1}$ is very efficient, it is used in many software. Very popular is the pair $G_7\, K_{15}$.

9.3 Subroutine QUANC8

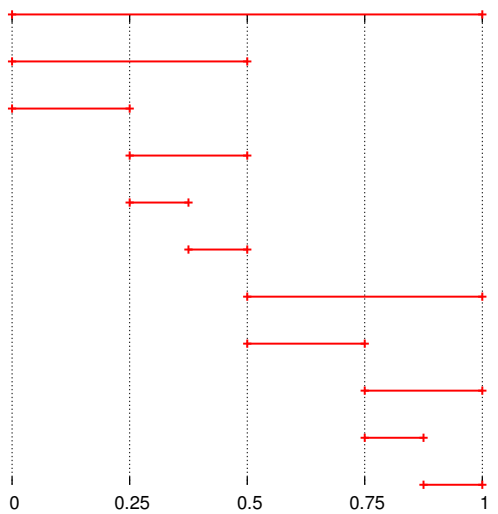
9.3.1 Overview

Subroutine QUANC8 is a code computing numerically integral (9.1) with the aid of the Newton-Cotes formula with $n = 9$ (eight panels) and the local adaptive strategy using the criterion (8.15) with $q = 1$. Therefore, the interval is split several times by the bisection of the interval until the following condition is fulfilled:

$$|R^I(f)| \le \frac{\mathrm{TOL}}{r}, \quad r = \frac{b-a}{|I|}, \tag{9.17}$$

where I denotes formally a sub-interval arising from the bisection, $R^I(f)$ is the estimation of the error and TOL is the given tolerance.

The following figure illustrates a possible splitting of interval $[0, 1]$.



In order to achieve the efficiency, all values $f(x_i)$, which may be used in a next refinement, are stored.

9.3.2 Input/output parameters

The subroutine QUANC8 is called by the command

```
call QUANC8(FUN,A,B,ABSERR,RELERR,RESULT,ERREST,NOFUN,FLAG)
```

where the input parameters are

```
FUN     THE NAME OF THE INTEGRAND FUNCTION SUBPROGRAM FUN(X).
A       THE LOWER LIMIT OF INTEGRATION.
B       THE UPPER LIMIT OF INTEGRATION.(B MAY BE LESS THAN A.)
RELERR  A RELATIVE ERROR TOLERANCE. (SHOULD BE NON-NEGATIVE)
ABSERR  AN ABSOLUTE ERROR TOLERANCE. (SHOULD BE NON-NEGATIVE)
```

and the output parameters are

```
RESULT  AN APPROXIMATION TO THE INTEGRAL HOPEFULLY SATISFYING THE
        LEAST STRINGENT OF THE TWO ERROR TOLERANCES.
ERREST  AN ESTIMATE OF THE MAGNITUDE OF THE ACTUAL ERROR.
NOFUN   THE NUMBER OF FUNCTION VALUES USED IN CALCULATION OF RESULT.
FLAG    A RELIABILITY INDICATOR.  IF FLAG IS ZERO, THEN RESULT
        PROBABLY SATISFIES THE ERROR TOLERANCE.  IF FLAG IS
        XXX.YYY , THEN  XXX = THE NUMBER OF INTERVALS WHICH HAVE
        NOT CONVERGED AND  0.YYY = THE FRACTION OF THE INTERVAL
        LEFT TO DO WHEN THE LIMIT ON  NOFUN  WAS APPROACHED.
```

Each interval can be split be the bisection at most `LEVMAX`-times (default value is 30). If the condition (9.17) is not achieve then `FLAG = FLAG + 1`. Moreover, QUANC8 tries to use at most `NOMAX` nodes (default value is 5000). Hence, it predicts the used number of nodes, namely it checks the condition `NOFUN <= NOFIN`, where the value `NOFIN` is given by

```
NOFIN = NOMAX - 8*(LEVMAX-LEVOUT+2**(LEVOUT+1))
```

If `NOFUN > NOFIN` then we put `LEVMAX = LEVOUT` (default value is 6) and

```
FLAG = FLAG + (B-X0)/(B-A)
```

(`X0` is the node of the trouble where the condition `NOFUN > NOFIN` arises).

The nodes x_i and the values $f(x_i)$ are stored in arrays `XSAVE(1:8, 1:30)` and `FSAVE(1:8, 1:30)`, respectively. Moreover,

- `QPREV` – the value of the integral over the given interval,
- `QNOW` – the value of the integral over the given interval using half-size step,
- `QDIFF = QNOW - QPREV`,
- `ESTERR = ABS(QDIFF) / 1023.0D0` – error estimate,
- `COR11 = COR11 + QDIFF / 1023.0D0` – correction to the order 11.

9.3.3 Installation and use of the QUANC8 subroutine

- Archive can be downloaded from
  `http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/QUANC8.tar.gz`
- after unpacking of the file (in Linux by `tar xfz QUANC8.tar.gz`), an archive with the following files appears:
  - `makefile` – makefile for translation
  - `QUANC8.FOR` – the subroutine QUANC8
  - `SAMPLE.FOR` – the main program calling QUANC8 and containing the definition of the input parameters
- the code can be translated by the command `make` (if it is supported) which use the file `makefile`

```
SAMPLE   : SAMPLE.o QUANC8.o
        f77 -o SAMPLE SAMPLE.o QUANC8.o
SAMPLE.o : SAMPLE.FOR
        f77 -c  SAMPLE.FOR
QUANC8.o : QUANC8.FOR
        f77 -c  QUANC8.FOR
```

or by a direct use of previous commands, namely

```
f77 -c SAMPLE.FOR
f77 -c QUANC8.FOR
f77 -o SAMPLE SAMPLE.o QUANC8.o
```

The symbol `f77` denotes the name of the translator, it can be replaced by any available fortran translator (`g77`, `gfortran`, ...). If the translation is successful, the executable file `SAMPLE` arises.

- the setting of the input parameters has to be done by hand in the file `SAMPLE.FOR` (the translation has to be repeated thereafter):

```
C    SAMPLE PROGRAM FOR QUANC8

      REAL FUNCTION FUN (X)
      REAL X
      FUN = SQRT (X)
      RETURN
      END
```

```
      EXTERNAL FUN
      REAL A,B,ABSERR,RELERR,RESULT,ERREST,FLAG
      INTEGER NOFUN
      A =  0.0
      B =  1.0

      RELERR = 1.0E-07
      ABSERR = 1.0E-07
      CALL QUANC8(FUN,A,B,ABSERR,RELERR,RESULT,ERREST,NOFUN,FLAG)
      WRITE(6,1) RESULT,ERREST
      IF (FLAG.NE.0.0) WRITE(6,2)FLAG
    1 FORMAT(8H RESULT=, F15.10, 10H   ERREST=, E10.2)
    2 FORMAT(44H WARNING! RESULT MAY BE UNRELIABLE.  FLAG = ,F6.2)
      STOP
      END
```

- the code is run by `./SAMPLE`, the output looks like

```
RESULT=   0.3303169310   ERREST=  0.20E-07
WARNING! RESULT MAY BE UNRELIABLE.  FLAG =   4.00
```

9.4 Subroutine Q1DA

9.4.1 Overview

Subroutine Q1DA is a code computing numerically integral (9.1) with the aid of the Gauss-Kronrod formula $G_7\, K_{15}$ and the global adaptive strategy using the criterion (8.14). The computation starts with a random splitting of $[a, b]$ and integrating over both sub-intervals. If the criterion (8.14) is violated, the sub-intervals with the highest error is split onto two halves and the procedure is repeated.

9.4.2 Input/output parameters

The subroutine Q1DA is called by the command

```
call Q1DA(A,B,EPS,R,E,KF,IFLAG)
```

where the input/output parameters are

```
C       A
C       B     (INPUT) THE ENDPOINTS OF THE INTEGRATION INTERVAL
C       EPS   (INPUT) THE ACCURACY TO WHICH YOU WANT THE INTEGRAL
C                COMPUTED.  IF YOU WANT 2 DIGITS OF ACCURACY SET
```

```
C                EPS=.01, FOR 3 DIGITS SET EPS=.001, ETC.
C                EPS MUST BE POSITIVE.
C       R     (OUTPUT) Q1DA'S BEST ESTIMATE OF YOUR INTEGRAL
C       E     (OUTPUT) AN ESTIMATE OF ABS(INTEGRAL-R)
C       KF    (OUTPUT) THE COST OF THE INTEGRATION, MEASURED IN
C                      NUMBER OF EVALUATIONS OF YOUR INTEGRAND.
C                      KF WILL ALWAYS BE AT LEAST 30.
C       IFLAG (OUTPUT) TERMINATION FLAG...POSSIBLE VALUES ARE
C               0   NORMAL COMPLETION, E SATISFIES
C                        E<EPS  AND  E<EPS*ABS(R)
C               1   NORMAL COMPLETION, E SATISFIES
C                        E<EPS, BUT E>EPS*ABS(R)
C               2   NORMAL COMPLETION, E SATISFIES
C                        E<EPS*ABS(R), BUT E>EPS
C               3   NORMAL COMPLETION BUT EPS WAS TOO SMALL TO
C                     SATISFY ABSOLUTE OR RELATIVE ERROR REQUEST.
C
C               4   ABORTED CALCULATION BECAUSE OF SERIOUS ROUNDING
C                     ERROR.  PROBABLY E AND R ARE CONSISTENT.
C               5   ABORTED CALCULATION BECAUSE OF INSUFFICIENT STORAGE.
C                     R AND E ARE CONSISTENT.
C               6   ABORTED CALCULATION BECAUSE OF SERIOUS DIFFICULTIES
C                     MEETING YOUR ERROR REQUEST.
C               7   ABORTED CALCULATION BECAUSE EPS WAS SET <= 0.0
C
C            NOTE...IF IFLAG=3, 4, 5 OR 6 CONSIDER USING Q1DAX INSTEAD.
```

The integrand is given as an external function `F`, see file `EXAMPLE.FOR`:

```
 C Typical problem setup for Q1DA
C

         A = 0.0
         B = 1.0
C Set interval endpoints to [0,1]
         EPS = 0.001
C Set accuracy request for three digits
         CALL Q1DA (A,B,EPS,R,E,KF,IFLAG)
         WRITE(*,*)'Q1DA RESULTS: A, B, EPS, R, E, KF, IFLAG'
         WRITE(*,'(3F7.4,2E16.8,2I4)') A,B,EPS,R,E,KF,IFLAG
         WRITE(*,*)
         END
C
```

```
          FUNCTION F(X)
C Define integrand F
c          F = SIN(2.*X)-SQRT(X)
          F = SQRT (X)
          RETURN
          END
```

Finally, we mention several remarks concerning Q1DA code:

- in order to avoid some troubles caused by singularities at the endpoints, Q1DA use a transformation of the integrand

$$\int_a^b f(x)\,\mathrm{d}x = \int_a^b f(g(y))g'(y)\,\mathrm{d}y, \qquad g(y) = b-(b-a)u^2(2u+3), \quad u = \frac{y-b}{b-a}.$$

  Here $g(a) = a$, $g(b) = b$ and $g'(a) = g'(b) = 0$.

- The maximal number of panels is set to `NMAX = 50`. If this number is achieved and the given stopping criterion is not satisfied, the code allows to remove from the memory the sub-interval with the smallest error and then employ this memory for a next refinement.

9.4.3 Installation and use of the Q1DA subroutine

- Archive can be downloaded from
  `http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/Q1DA.tar.gz`

- after unpacking of the file (in Linux by `tar xfz Q1DA.tar.gz`), and archive with the following files appears:

  - `makefile` – makefile for translation
  - `Q1DA.FOR` – the subroutine Q1DA
  - `EXAMPLE.FOR` – the main program calling Q1DA and containing the definition of the input parameters
  - `BLAS.FOR` – Basic Linear Algebra Subroutines, some subroutines from the BLAS library
  - `MACHCON.FOR` – evaluation of the machine dependent parameters (UFL, OFL, ϵ_{mach},...)
  - `UNI.FOR` – generation of a random number for the first splitting
  - `XERROR.FOR` – error messages

- the code can be translated by the command `make` (if it is supported) which use the file `makefile`

```
EXAMPLE   : EXAMPLE.o Q1DA.o BLAS.o MACHCON.o UNI.o XERROR.o
        f77 -o EXAMPLE EXAMPLE.o Q1DA.o BLAS.o MACHCON.o UNI.o XERROR.o
EXAMPLE.o : EXAMPLE.FOR
        f77 -c  EXAMPLE.FOR
Q1DA.o : Q1DA.FOR
        f77 -c  Q1DA.FOR
BLAS.o : BLAS.FOR
        f77 -c BLAS.FOR
MACHCON.o : MACHCON.FOR
        f77 -c MACHCON.FOR
UNI.o : UNI.FOR
        f77 -c UNI.FOR
XERROR.o : XERROR.FOR
        f77 -c XERROR.FOR
```

or by a direct use of previous commands, namely

```
f77 -c  EXAMPLE.FOR
f77 -c  Q1DA.FOR
f77 -c BLAS.FOR
f77 -c MACHCON.FOR
f77 -c UNI.FOR
f77 -c XERROR.FOR
f77 -o EXAMPLE EXAMPLE.o Q1DA.o BLAS.o MACHCON.o UNI.o XERROR.o
```

The symbol `f77` denotes the name of the translator, it can be replaced by any available fortran translator (`g77`, `gfortran`, ...). If the translation is successful, the executable file `EXAMPLE` arises.

- the setting of the input parameters has to be done by hand in the file `EXAMPLE.FOR` (the translation has to be repeated thereafter), see above.
- the code is run by `./EXAMPLE`, the output looks like

```
 Q1DA RESULTS: A, B, EPS, R, E, KF, IFLAG
 0.0000 1.0000 0.0010  0.66666663E+00  0.39666666E-05  30   0
```

9.4.4 Several remarks

A numerical computation of integrals may fail, i.e., a code returns

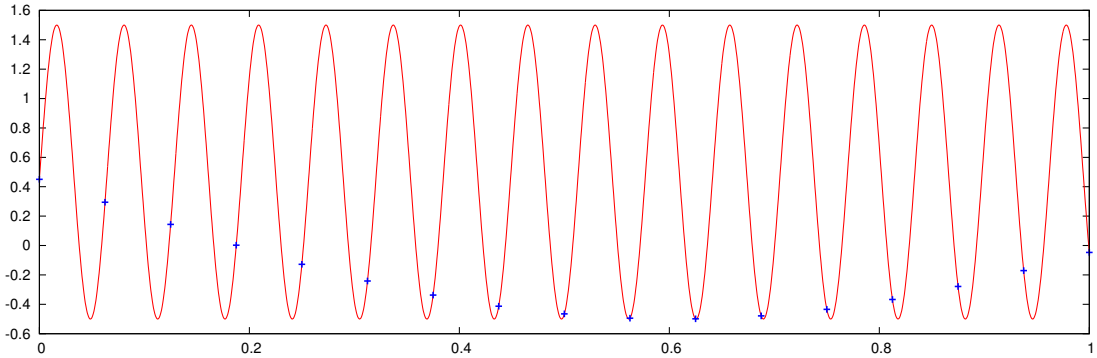
- bad approximate value of the integral,
- bad estimate of the error,

- correct values of the integral and the error estimators, but the indicator (`FLAG`) indicates non-reliability of the result.

This is caused by several reasons, some of them are listed bellow.

Equidistantly distributed nodes and periodically oscillating integrands

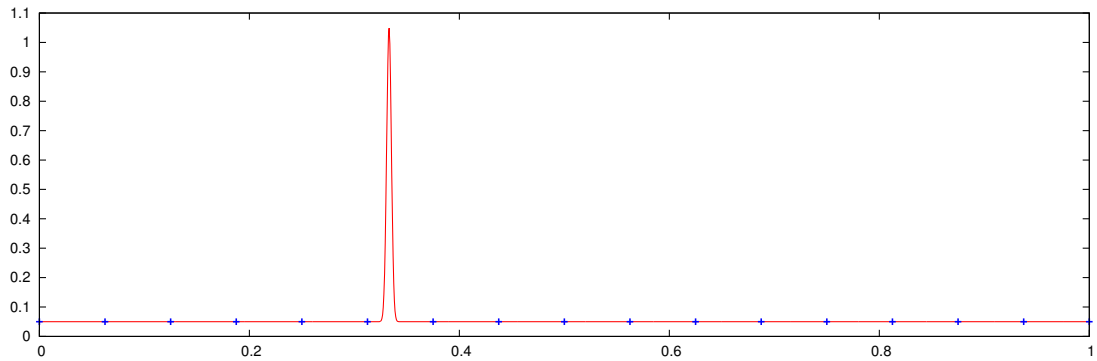
Any code has information about the integrand only at nodes of the quadrature formula. The following figure shows a periodically oscillating function (red line) Using equidistantly distributed nodes (blue crosses), we see completely different function.



It is obvious that for this case, the estimation of the error by the half-size step method gives a vanishing estimate even for a coarse partition although the result is bad.

Integrand having a relative small support

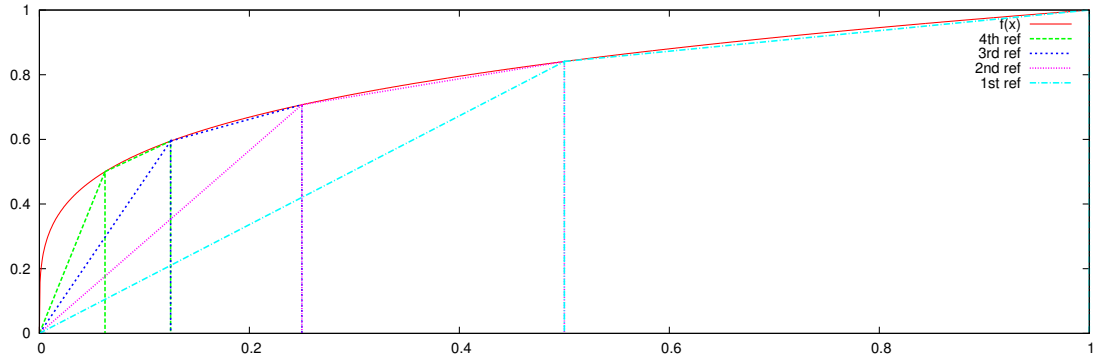
Again, any code has information about the integrand only at nodes of the quadrature formula. The following figure shows a function having a relative small support (= essential information is localised for a small interval) (red line). Then coarse partitions (blue crosses) are not able to capture this support.



It is obvious that for this case, the estimation of the error by two different quadrature (both having a coarse partition) gives (almost) zero although the result is bad.

Local adaptation for integrands with a singularity

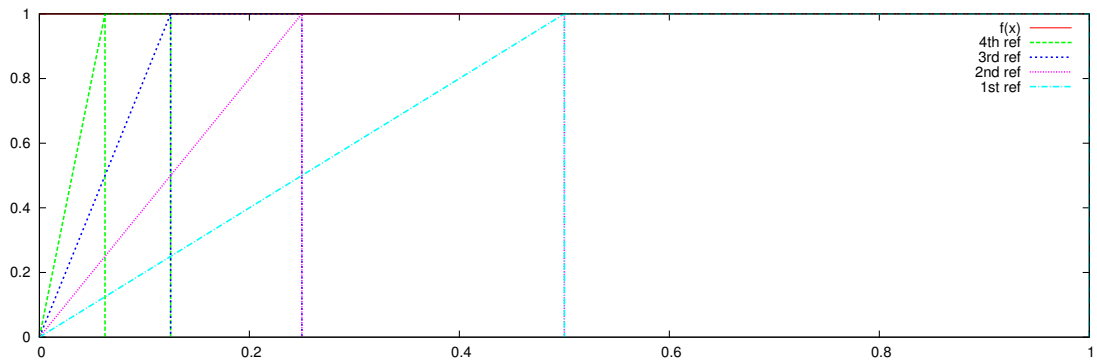
Let us consider an integrand $f(x)$ with a singularity such that $f'(a) \to \infty$. For simplicity, we consider the trapezoid formulae and estimate the error by the half-size step method. The following figure shows a sequence of refinement of the sub-intervals closes to the singularity.



The size of the error estimator corresponds to the areas of the obtuse triangles K_ℓ having two vertices on the graph of $f(x)$ and the third vertex is the origin. In order to fulfil the criterion (8.15) of the local adaptation, it is necessary that the areas of these triangles is decreasing sufficiently fast, namely

$$\frac{|K_{\ell+1}|}{|K_\ell|} < 2.$$

In some situation, this condition is not valid. Let us consider a hypothetical example $f(0) = 0$ and $f(x) = 1\ \forall X \in (0, 1]$. The corresponding situation is pictured bellow.



Obviously, the areas of triangles (= values of the error estimators) satisfy $|K_{\ell+1}| / |K_\ell| = 2$ hence the local adaptive criterion (8.15) can not be achieved.

In practice the condition $|K_{\ell+1}| / |K_\ell| < 2$ is satisfied but if $|K_{\ell+1}| / |K_\ell| \approx 2$ then (8.15) is not achieve within the prescribed maximal number of adaptation. Therefore, the indicator (`FLAG`) may indicate non-convergence but the error as well as its estimate may by sufficiently small.

Integrands with a singularity

Let $f : (a, b) \to \mathbb{R}$ have a singularity at $x = a$. Then the numerical integration

$$\int_a^b f(x)\,\mathrm{d}x \tag{9.18}$$

may cause troubles by closed quadratures. Possible solutions:

(i) there exists a finite limit $\lim_{x\to a^+} f(x) = A$. Then we define

$$\bar{f}(x) := \begin{cases} A & \text{for } x = a, \\ f(x) & \text{for } x \in (a, b], \end{cases} \tag{9.19}$$

and instead of (9.18) integrate $\int_a^b \bar{f}(x)\,\mathrm{d}x$. It make a good sense since the change $\bar{f} \to f$ is on the set of measure equal to 0. In principle, we can replace the value at $x = a$ by any real number but if the integrand is not smooth then the used quadrature rule has only the lowest order of accuracy.

(ii) We can replace (9.18) by

$$\int_\epsilon^b f(x)\,\mathrm{d}x, \qquad \epsilon > 0 \tag{9.20}$$

which makes good sense. Then we should (numerically) investigate if the limit

$$\lim_{\epsilon\to a^+} \int_\epsilon^b f(x)\,\mathrm{d}x \tag{9.21}$$

is finite. Particularly, we have to carry out several experiments for decreasing ϵ and try to set the limit value.

Do Main task 1

Homeworks

Exercise 13. *Find an example, where the estimate (9.9) fails, i.e., when the right-hand side of (9.9) is zero whereas the error $I(f) - M(f)$ is non-zero.*

Exercise 14. *Verify the order of the two-point Gauss quadrature formula (9.15).*

Exercise 15. *Transform the two-point Gauss quadrature formula (9.15) to interval $[0, 1]$.*

Exercise 16. *Write a simple code which computes the given integral using*

- **composite midpoint** *rule,*
- **trapezoid** *rule,*
- **Simpson** *rule.*

Show by numerical examples the following items:

1. *the order of the corresponding quadrature is p, i.e., $Q(f)$ is exact for polynomials f of degree p,*
2. *the order of the corresponding composite quadrature is p, i.e., $Q(f) = O(h^{p+1})$,*
3. *test and explain, why $Q(f) = O(h^{p+1})$ is not true for $\int_0^1 \sqrt{x}\,\mathrm{d}x$?*
4. *estimate the errors of the midpoint formula by the relation*

$$E \approx \frac{1}{3}(M(f) - T(f))$$

for smooth and non-smooth functions.

Main task 1. *With the aid of codes QUANC8 and Q1DA, compute the following integrals:*

(I1) $\displaystyle\int_0^1 e^{x^2}\,\mathrm{d}x$

(I2) $\displaystyle\int_0^1 e^{-x^{-2}}\,\mathrm{d}x$

(I3) $\displaystyle\int_0^2 \sin(10x)\,\mathrm{d}x$

(I4) $\displaystyle\int_0^2 \sin(100x)\,\mathrm{d}x$

(I5) $\displaystyle\int_1^{100} \ln x\,\mathrm{d}x$

(I6) $\displaystyle\int_0^1 \sqrt{x}\ln x\,\mathrm{d}x$

(I7) $\displaystyle\int_1^5 \frac{(x-1)^{1/5}}{x^2+1}\,\mathrm{d}x$

(I8) $\displaystyle\int_0^2 \frac{\sin x}{x}\,\mathrm{d}x$

(I9) $\displaystyle\int_0^2 \frac{\tan x}{x}\,\mathrm{d}x$

(I10) $\displaystyle\int_{-1}^1 \frac{1}{1+100x^2}\,\mathrm{d}x$

(I11) $\displaystyle\int_0^1 \frac{1}{\sqrt{x}}\,\mathrm{d}x$

(I12) $\displaystyle\int_{-200\,000}^{200\,000} x^2 \exp(-\frac{1}{2}x^2)\,\mathrm{d}x$

(I13) $\displaystyle\int_0^1 f(x)\,\mathrm{d}x,\ \textit{where}\ f(x) = \begin{cases} \frac{1}{x+2} & 0 \le x < e-2 \\ 0 & e-2 \le x \le 1 \end{cases}$

The main task is **not to obtain the correct value of the integral and the error** *but* **to understand the behaviour of the codes**. *Some integrals are simple and some of them cause troubles, i.e., the results or the error estimates are bad. For each integral (I1) – (I13), carry out the following steps:*

- *predict the possible troubles from the analytical form of the integrals,*
- *test both codes with several (at least two) tolerances,*
- *based on the resulting value of the integral, error estimate and the indicator (*`FLAG`*) decide, if the result is reliable,*
- *in case of some troubles (mentioned at the beginning of Section 9.4.4), give an explanation,*
- *perform also a reference computation using a software package as Matlab, Maple, Mathematica, etc. (obligatory)*

Chapter 10

Ordinary differential equations

10.1 Problem definition

The aim is to solve numerically the following first order initial value problem represented by an ordinary differential equation: we seek a function $y : [a, b] \to \mathbb{R}^m$, $m \in \mathbb{N}$ such that

$$y' = f(x, y), \tag{10.1a}$$
$$y(a) = \eta, \tag{10.1b}$$

where

$$f = (f_1, \ldots, f_m) : [a, b] \times \mathbb{R}^m \to \mathbb{R}^m, \qquad \eta \in \mathbb{R}^m$$

are the given data. The relation (10.1b) is called the initial condition.

We assume that f is such that there exists a unique global solution $y : [a, b] \to \mathbb{R}^m$. [1]

Remark 10.1. *The problem (10.1) covers also the case of ODE of m^{th} order ($m > 1$):*

$$u^{(m)} = g(x, u, u', u'', \ldots, u^{(m-1)}) \tag{10.2}$$

with initial conditions

$$u(a) = \eta_1, \quad u'(a) = \eta_2, \quad u''(a) = \eta_3, \quad \ldots \quad u^{(m-1)}(a) = \eta_m.$$

Using the substitution

$$y_1 = u, \quad y_2 = u', \quad y_3 = u'', \quad \ldots, \quad y_m = u^{(m-1)},$$

the m^{th} order scalar equation (10.2) reduces to a system of m ODEs of the first order

$$y' = f(x, y(x)) \quad \Leftrightarrow \quad \frac{\mathrm{d}}{\mathrm{d}x} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m-1} \\ y_m \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ \vdots \\ y_m \\ g(x, y_1, y_2, \ldots, y_m) \end{pmatrix}.$$

[1] E.g., we can assume that f is Lipschitz continuous function. Then the proof of the existence follows from the Picard theorem.

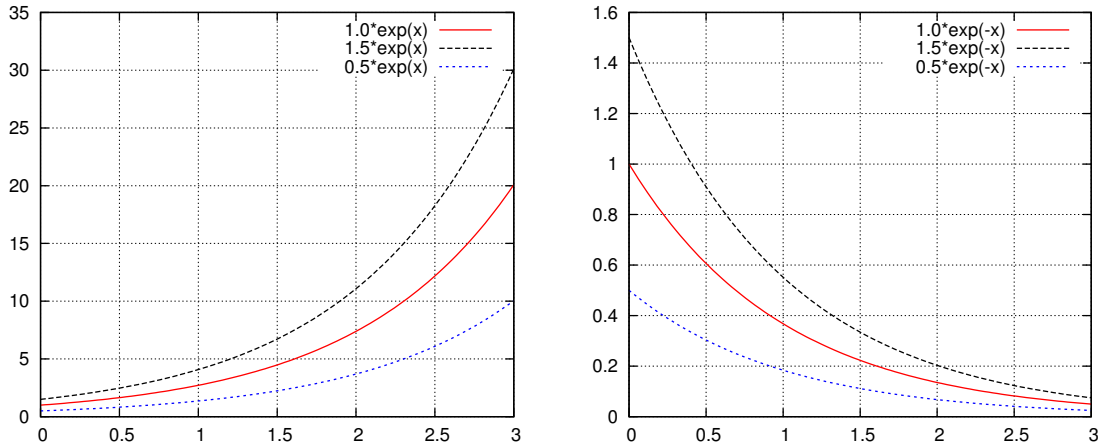


Figure 10.1: Example of the unstable system $y' = y$ (left) and the stable system $y' = -y$ (right).

Therefore, a software developed for system (10.1) *can be used with a success for problems* (10.2).

10.1.1 Stability of the system (10.1)

Let $f = f(x, y) = (f_1, \ldots, f_m) \in \mathbb{R}^m$ be the function from (10.1) depending on $y = (y_1, \ldots, y_m)$. Moreover, let $f \in C^1([a, b] \times \mathbb{R}^m)$. We denote by

$$J_f := \left\{ \frac{\partial f_i}{\partial y_j} \right\}_{i,j=1}^{m} \tag{10.3}$$

the Jacobi matrix corresponding to f.

Definition 10.2. *We say that system* (10.1) *is stable (well conditioned) if all eigenvalues of the Jacobi matrix J_f have negative real part.*
We say that system (10.1) *is unstable (ill conditioned) if all eigenvalues of the Jacobi matrix J_f have positive real part.*

Example 10.3. *The ODE $y' = y$ is unstable whereas the ODE $y' = -y$ is stable. Figure 10.1 shows the solutions of $y' = y$ and $y' = -y$ for different initial conditions. We observe that in the former case the solutions are grow away each other for $x \to \infty$. Other hand, in the latter case the solutions are closer to each other for $x \to \infty$. This indicate that (discretization as well as rounding) errors will magnify (diminish) for unstable (stable) systems.*

Obviously, if some eigenvalues of J_f have negative real parts and the others have positive real parts, the system is neither stable nor unstable. Sometimes we say that some components of system (10.1) are stable and the others unstable.

Example 10.4. *Let us consider a linear system*

$$y'(x) = \mathbb{A}y(x), \qquad y(0) = y_0 \in \mathbb{R}^m, \tag{10.4}$$

where $\mathbb{A}$ *is a matrix* $m \times m$*. Then* $\mathbb{A}$ *is the Jacobi matrix of the right-hand side of* (10.4)*. Let* $\lambda_i,\ i = 1, \dots, m$ *and* $u_i,\ i = 1, \dots, m$ *be the eigenvalues and eigenvectors of* $\mathbb{A}$*, respectively such that the eigenvectors create a basis of* $\mathbb{R}^m$*. Then, there exists* $\alpha_i \in \mathbb{R},\ i = 1, \dots, m$ *such that*

$$y_0 = \sum_{i=1}^{m} \alpha_i u_i.$$

Then the exact solution of (10.4) *is*

$$y(x) = \sum_{i=1}^{m} \alpha_i u_i \exp(\lambda_i x).$$

We observe that
if $Re\,\lambda_i > 0$ *then the corresponding component* $\alpha_i u_i \exp(\lambda_i x)$ *is unstable,*
if $Re\,\lambda_i < 0$ *then the corresponding component* $\alpha_i u_i \exp(\lambda_i x)$ *is stable,*
if $Re\,\lambda_i = 0$ *then the corresponding component* $\alpha_i u_i \exp(\lambda_i x)$ *is neutrally stable.*

Moreover, the stability of system (10.1) can change for $x \in [a, b]$:

Example 10.5. *Let us consider the problem*

$$y' = -2\alpha(x - 1)y. \tag{10.5}$$

The exact solution is $y(x) = c\exp[-\alpha(x-1)^2]$*. Obviously, ODE* (10.5) *is unstable for* $x < 1$ *but stable for* $x > 1$*. Figure 10.2 shows the exact solution of* (10.5) *for several initial conditions.*

10.1.2 Stiff systems

A special class of stable ODE systems are the so-called *stiff systems*. A definition of a stiff system is usually vague. Hardly speaking, a numerical solution of stiff systems by a numerical method with a restricted region of stability (e.g., explicit methods) requires very small time step in comparison to unconditionally stable methods.

Another characterization of stiff system is the following:

Definition 10.6. *We say that system* (10.1) *is stiff if the eigenvalues of* J_f *have negative real parts (= system is stable) with very different magnitudes.*

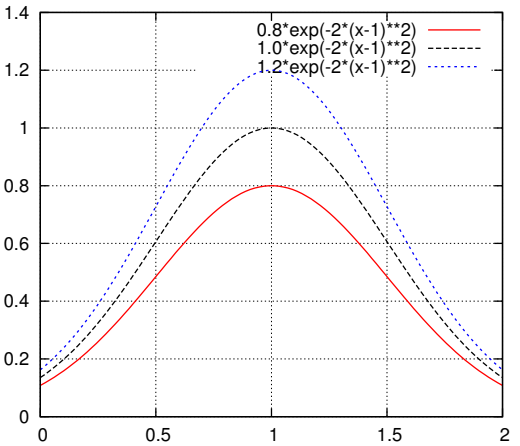


Figure 10.2: Exact solution of (10.5) for $\alpha = 2$ for several initial conditions, the system is unstable for $x < 1$ but stable for $x > 1$.

Example 10.7. *Let us consider the system*

$$u' = 998u + 1998v, \tag{10.6}$$
$$v' = -999u - 1999v. \tag{10.7}$$

Then the Jacobi matrix reads

$$J_f = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix},$$

the eigenvalues of J_f are -1 and -1000. Therefore, in virtue of Definition 10.6, system (10.6) is stiff. Numerical solution of stiff ODE by an explicit method requires very small time step although it is not necessary from the point of view of accuracy, see Section 10.3.3.

Example 10.8. *Let us consider the ODE problem*

$$y' = -\alpha(y - \sin x) + \cos x, \quad y(0) = 1, \qquad \alpha > 0.$$

The exact solution is

$$y(x) = e^{-\alpha x} + \sin x.$$

If $\alpha \gg 1$ then this example is stiff.

Do Exercise 17

10.2 Numerical solution of ODE

The basic idea of a numerical solution of problem (10.1) is the following. Let

$$a = x_0 < x_1 < x_2 < \cdots < x_N = b$$

be a partition of the interval $[a, b]$, x_i, $i = 0, \ldots, N$ are called the nodes. We approximate the value of the solution $y(x)$ at nodes x_i by

$$y(x_i) \approx y_i \in \mathbb{R}^m, \quad i = 0, \ldots, N. \tag{10.8}$$

Usually we put $y_0 = \eta$, where η is given by the initial condition (10.1b).

The unknown values y_i, $i = 0, \ldots, N$ are given usually either by a one-step formula

$$y_{k+1} = F_k(x_{k+1}, x_k; y_{k+1}, y_k), \quad k = 0, \ldots, N-1 \tag{10.9}$$

or by a multi-step formula

$$y_{k+1} = F_k(x_{k+1}, x_k, x_{k-1}, \ldots, x_{k-\ell+1}; y_{k+1}, y_k, y_{k-1}, \ldots, y_{k-\ell+1}), \quad k = p-1, \ldots, N-1, \tag{10.10}$$

where F_K are suitable functions of their arguments. The formula (10.10) represents the ℓ-step formula. If the functions F_k in (10.9) or (10.10) do not depend explicitly on y_{k+1}, we speak about an explicit method, otherwise the method is implicit.

We see that the approximate solution y_{k+1} at the time level x_{k+1} is computed using the approximate solutions on the previous time levels x_l, $l = 0, \ldots, k$. This means that any error (discretization as well as rounding errors) is propagated.

We distinguish two types of errors[2]:

- global error (= accumulated error) is given simply by

$$G_k := y_k - y(x_k), \quad k = 0, \ldots, N. \tag{10.11}$$

  Therefore, the global error G_k represents the difference between the exact and the approximate solution at the node x_k, i.e., the error after k steps.

- local error is given by

$$L_k := y_k - u_{k-1}(x_k), \quad k = 0, \ldots, N, \tag{10.12}$$

  where u_{k-1} is the function satisfying (10.1a) with the condition $u_{k-1}(x_{k-1}) = y_{k-1}$. Therefore, the local error L_k represents the error arising within the k^{th}-step.

[2]Let us note that the rounding errors are usually neglected

Figure 10.3 illustrates the global and local errors. Moreover, it shows that for an unstable system (10.1), we have

$$G_k > \sum_{l=1}^{k} L_l.$$

On the other hand, we can deduce that for a stable system (10.1), we have

$$G_k < \sum_{l=1}^{k} L_l.$$

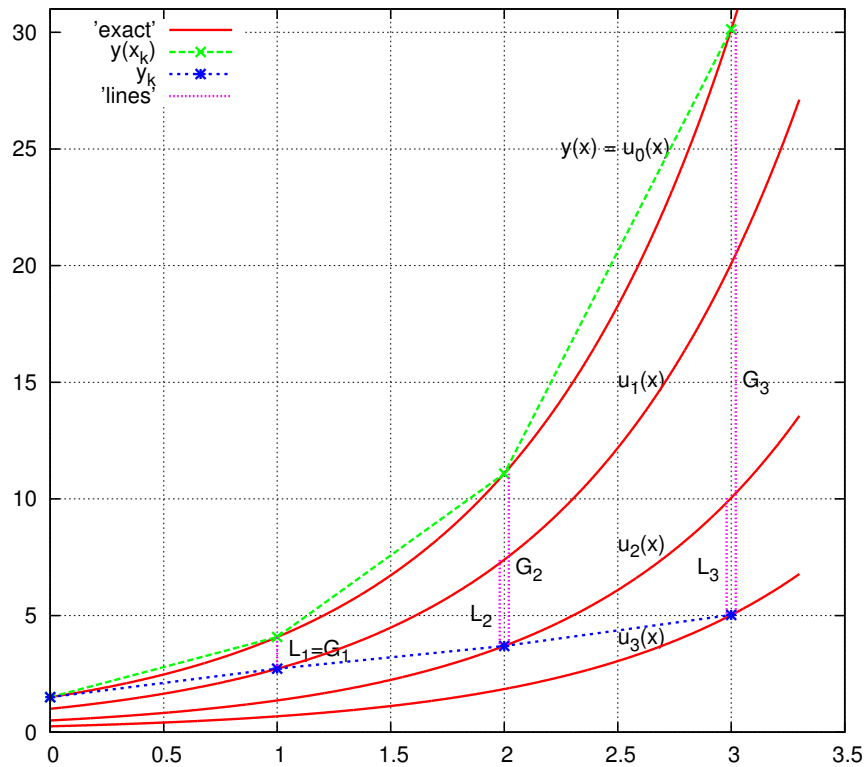


Figure 10.3: An illustration of global and local errors G_k and L_k, respectively.

Definition 10.9. *We say that a numerical method has the order p if $L_k = O(h^{p+1})$.*

Let us note that if a method has the order p then $\sum_{k=1}^{N} L_k \approx \frac{b-a}{h} O(h^{p+1}) = O(h^p)$.

A natural requirement for the numerical solution of ODEs is $|G_k| \leq \text{TOL}$. However, in practice, we are able estimate only the local error L_k. Therefore, the most of software for the numerical solution of ODEs is based on the estimation of the local error.

10.3 Explicit Euler method

The simplest method for the numerical solution of ODEs is the (explicit or forward) Euler method. For simplicity, we consider a scalar equation (10.1) with $m = 1$. Let $a = x_0 < x_1 < x_2 < \cdots < x_N = b$ be a partition of $[a, b]$, we put

$$h_k = x_{k+1} - x_k, \qquad k = 0, \ldots, N-1.$$

Let $y \in C^2([a, b])$ be the exact solution of (10.1) and $k = 0, \ldots, N-1$, then the Taylor expansion gives

$$y(x_k + h_k) = y(x_k) + y'(x_k)h_k + \frac{1}{2}y''(x_k + \tau_k h_k)h_k^2, \quad \tau_k \in [0, 1]. \tag{10.13}$$

Using the equality $y'(x_k) = f(x_k, y(x_k))$ (following from (10.1a)), the approximation $y(x_i) \approx y_i$, $i = 0, \ldots, m$ and neglecting the higher order term $y''(\cdot)h_k^2$, we get from (10.13)

$$y_{k+1} = y_k + h_k f(x_k, y_k), \quad k = 0, \ldots, N-1, \tag{10.14}$$

which is the explicit Euler method. Usually, we put

$$y_0 = \eta, \tag{10.15}$$

where η is given by the initial condition (10.1b). Relations (10.14) and (10.15) define the sequence of approximations y_k, $k = 1, 2, \ldots, N$.

The local error of the Euler method corresponds to the neglected higher order term in (10.13), i.e.,

$$L_k = \frac{1}{2}y''(x_k + \tau_k h_k)h_k^2. \tag{10.16}$$

Therefore, the Euler method has the order 1.

10.3.1 Stability

We analyze the stability of the Euler method, see Definition 1.13. Subtracting (10.13) from (10.14) and using the identity $y'(x_k) = f(x_k, y(x_k))$, we get

$$\underbrace{y_{k+1} - y(x_{k+1})}_{G_{k+1}} = \underbrace{y_k - y(x_k)}_{G_k} + h_k \underbrace{\left(f(x_k, y_k) - f(x_k, y(x_k))\right)}_{\text{propagation of the error}} - \underbrace{\frac{1}{2}y''(x_k + \tau_k h_k)h_k^2}_{\text{local error}}. \tag{10.17}$$

Let $f \in C^1([a, b] \times \mathbb{R})$, then there exists $\xi \in \mathbb{R}$ between y_k and $y(x_k)$ such that

$$f(x_k, y_k) - f(x_k, y(x_k)) = J_f(x_k, \xi)(y_k - y(x_k)), \tag{10.18}$$

where J_f is the Jacobi matrix given by (10.3). Inserting (10.18) into (10.17) and using (10.11) and (10.16), we obtain

$$G_{k+1} = (1 + h_k J_f(x_k, \xi))\, G_k + L_k. \tag{10.19}$$

The term $A := 1 + h_k J_f(x_k, \xi)$ is called the amplification factor.

Let $L_k \approx L$ for any $k = 0, 1, 2, \ldots$. Then we have

$$\begin{aligned}
G_1 &= AG_0 + L,\\
G_2 &= AG_1 + L = A^2 G_0 + L(1 + A),\\
G_3 &= AG_2 + L = A^3 G_0 + L(1 + A + A^2),\\
&\vdots\\
G_{k+1} &= A^{k+1} G_0 + L(1 + A + \cdots + A^k)\\
&= A^{k+1} G_0 + L\frac{A^{k+1} - 1}{A - 1}.
\end{aligned}$$

So, if $|A| > 1$ then G_{k+1} will blow up.

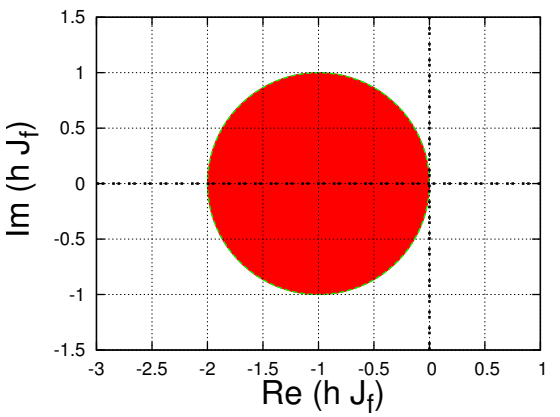
Definition 10.10. *A numerical method is stable if the magnitude of the amplification factor is strictly less than* 1.

This means that for a stable numerical method, the propagation of the errors from previous time step is limited. Therefore, the rounding errors do not destroy the approximate solution. Example 1.15 showed a behaviour of an unstable numerical method. **Demonstrate by the computer**

For the explicit Euler method, we have the stability condition

$$|1 + h_k J_f(x_k, \xi)| < 1,$$

which means that $h_k J_f \in (-2, 0)$ for a scalar equation and/or the spectral radius $\rho(\mathbb{I} + h_k J_f) < 1$ for a system of ODEs. The so-called domain of stability is shown in the following figure, which shows the value of $h_k \lambda_f$ in the complex plane (λ_f formally denotes an eigenvalue of J_f):



Therefore, if the ODE system (10.1) is unstable ($Re\,\lambda_f > 0$) then the explicit Euler method is always unstable. On the other hand, if the ODE system (10.1) is stable ($Re\,\lambda_f < 0$) then the explicit Euler method is stable only if h_k is sufficiently small (namely if $h_k < \frac{2}{|\lambda_f|}$ for the scalar equation). We say the the explicit Euler method is conditionally stable.

Let us note that generally the amplification factor A depends on

- ODE – size of J_f,
- numerical methods (for the explicit Euler equation we have $A = 1 + h_k J_f(x_k, \xi)$),
- the size of h_k.

10.3.2 Error estimate and the choice of the time step

In order to efficiently solve ODEs, the steps $h_k,\ k = 1, \dots, m$ have to be chosen such that

- it is sufficiently small in order to guarantee the stability and the accuracy,
- it is sufficiently large in order to achieve an efficiency.

In order to balance between the accuracy and the efficiency, we (usually) choose h_k such that

$$L_k = \mathrm{TOL}, \qquad k = 1, \dots, N, \tag{10.20}$$

where $\mathrm{TOL} > 0$ is the given tolerance.

For the explicit Euler method the local discretization error is given by (10.16), hence we have

$$\frac{1}{2} y''(x_k + \tau_k h_k) h_k^2 = \mathrm{TOL} \qquad \Leftrightarrow \qquad h_k = \sqrt{\frac{2\mathrm{TOL}}{y''(\cdot)}}. \tag{10.21}$$

The second order derivative can be approximated by a difference

$$y'' \approx \frac{y'_k - y'_{k-1}}{x_k - x_{k-1}} = \frac{f(x_k, y_k) - f(x_{k-1}, y_{k-1})}{x_k - x_{k-1}}. \tag{10.22}$$

In the most software for the numerical solution of ODEs, the time steps h_k, $k = 1, \dots, n$ are chosen according condition (10.20), i.e., the accuracy and the efficiency are balanced. On the other hand, the stability condition is not explicitly checked. In fact it is hidden in the approximation of L_k.

Let us consider the ODE (10.5),

$$y' = -2\alpha(x - 1)y. \tag{10.23}$$

The exact solution is

$$y(x) = c \exp[-\alpha(x - 1)^2]$$

and thus

$$y''(x) \approx x^2 \exp[-\alpha x^2] \quad \text{for } x \gg 1.$$

Therefore,

$$y''(x) \to 0 \quad \text{for } x \to \infty,$$

and, using the explicit Euler method, we have $L_k \ll 1$ for large x and thus relation (10.21) gives $h_k \gg 1$. Then the stability condition may be violated

However, in a practical computation, when that stability of the method is violated, the approximations y_k oscillate and then the approximation of the second order derivative (10.22) gives a large value and consequently h_k will be chosen smaller. See Figure 10.4 showing the approximate solution of (10.23) by the explicit Euler method using the fixed time step $h = 0.02$ and the adaptively chosen time step according (10.21) – (10.22) with $\mathrm{TOL} = 10^{-4}$. **Demonstrate by the computer**

Let us note that the choice of h_k directly according (10.21) – (10.22) is a little naive, in practice more sophisticated techniques are required, see Section 10.7

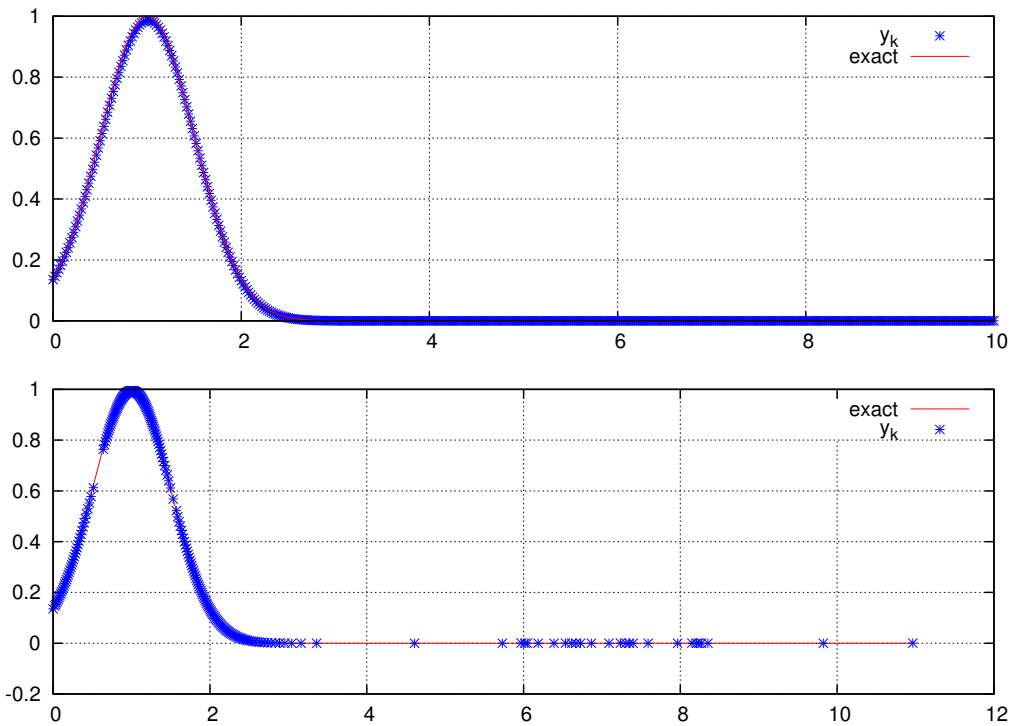


Figure 10.4: Approximate solution of (10.23) by the explicit Euler method using the fixed time step $h = 0.02$ (top) and the adaptively chosen time step according (10.21) – (10.22) with $\mathrm{TOL} = 10^{-4}$ (bottom).

Do Exercise 18

10.3.3 Euler method for stiff problems

Numerical solution of stiff ODE systems (see Section 10.1.2) by conditionally stable methods is usually very inefficient. Let us consider problem (10.6) written in the form

$$\begin{aligned} u' &= 998u + 1998v, \\ v' &= -999u - 1999v \end{aligned} \tag{10.24}$$

with the initial conditions $u(0) = 1$ and $v(0) = 0$. The eigenvalues of J_f corresponding to (10.24) are -1 and -1000, thus the stability condition for the explicit Euler method gives

$$|1 + h_k J_f| < 1 \qquad \Rightarrow \qquad h_k < 2 \cdot 10^{-3}. \tag{10.25}$$

Moreover, the exact solution or (10.24) is

$$\begin{aligned} u(x) =& 2\mathrm{e}^{-x} - \mathrm{e}^{-1000\,x}, \\ v(x) =& -\mathrm{e}^{-x} + \mathrm{e}^{-1000\,x}. \end{aligned}$$

Then

$$\begin{aligned} u''(x) =& 2\mathrm{e}^{-x} - 10^6\mathrm{e}^{-1000\,x}, \\ v''(x) =& -\mathrm{e}^{-x} + 10^6\mathrm{e}^{-1000\,x}. \end{aligned}$$

Hence, from (10.21), we have $h_k \approx \frac{K}{\sqrt{\max(|u''|,\ |v''|)}}$, which gives

$$\begin{aligned} &\text{for } x = 0 \qquad h_k \approx K10^{-3}, \\ &\text{for } x = 1 \qquad h_k \approx K10^{0}, \\ &\text{for } x = 3 \qquad h_k \approx K10^{1}. \end{aligned}$$

Therefore, from the point of view of the accuracy, the time step h_k can be many times larger than the stability condition (10.25) allows.

This is the characteristic property of stiff ODE systems: their numerical solution by a conditionally stable method is inefficient.

10.4 Implicit methods

In order to efficiently solve stiff systems, numerical method with a large domain of stability are required. The simplest are the implicit (backward) Euler method written in the form

$$y_{k+1} = y_k + h_k f(x_{k+1}, y_{k+1}), \quad k = 0, 1, \ldots, N-1. \tag{10.26}$$

Let us note that the right-hand side of (10.26) depends on the unknown approximate solution y_{k+1} hence a nonlinear algebraic system on each time level has to be solved. Usually, the Newton or a Newton-like method is employed. On the other hand, this method is unconditionally stable (see bellow) hence the time step can be chosen large and then the total computational time can be shorter.

10.4.1 Stability and accuracy of the implicit Euler method

We investigate the so-called linear stability of the implicit Euler method (10.26). We consider the linear ODE

$$y' = \lambda y, \qquad \lambda \in \mathbb{R}. \tag{10.27}$$

Let $y \in C^2([a,b])$, then the Taylor expansion at x_{k+1} gives

$$y(x_k) = y(x_{k+1}) - h_k y'(x_{k+1}) + L_k, \tag{10.28}$$

where $L_k = O(h_k^2)$ is the local discretization error, cf. (10.13) and (10.16). Therefore, for (10.27), we have from (10.28),

$$y(x_{k+1}) = y(x_k) + h_k y'(x_{k+1}) - L_k = y(x_k) + h_k \lambda y(x_{k+1}) - L_k. \tag{10.29}$$

Applying the implicit Euler method to (10.27), we obtain

$$y_{k+1} = y_k + \lambda h_k y_{k+1}. \tag{10.30}$$

Subtracting (10.30) from (10.29), we obtain

$$y_{k+1} - y(x_{k+1}) = y_k - y(x_k) + \lambda h_k (y_{k+1} - y(x_{k+1})) + L_k.$$

which gives

$$(1 - \lambda h_k)(y_{k+1} - y(x_{k+1})) = y_k - y(x_k) + L_k,$$

and with the notation (10.11)

$$G_{k+1} = AG_k + AL_k \qquad \text{with } A := \frac{1}{1 - \lambda h_k}, \tag{10.31}$$

where A is the amplification factor, cf. (10.19). We note that (10.30) also implies

$$y_{k+1} = Ay_k, \qquad A := \frac{1}{1 - \lambda h_k},$$

where A is given by (10.31).

In order to guaranteed the stability (the global error is not propagated with increasing factor), we need $|A| < 1$. Obviously, if $\lambda < 0$ (i.e. system (10.27) is stable) then $|A| < 1$ for any $h_k > 0$. We say that implicit Euler method is unconditionally stable.

Finally, we investigate the accuracy of the implicit Euler method. We consider again the ODE (10.27). The implicit Euler method and an expand into a series give

$$y_{k+1} = y_k \frac{1}{1 - \lambda h_k} = y_k \left[1 + \lambda h_k + (\lambda h_k)^2 + (\lambda h_k)^3 + \dots\right]. \tag{10.32}$$

Moreover, the Taylor expansion of the exact solution gives

$$\begin{aligned} y(x_k+h_k) =& y(x_k) + h_k y'(x_k) + \frac{1}{2}h_k^2 y''(x_k) + \frac{1}{6}h_k^3 y'''(x_k) + \dots, \\ =& \mathrm{e}^{\lambda x_k} + \lambda h_k \mathrm{e}^{\lambda x_k} + \frac{1}{2}(\lambda h_k)^2 \mathrm{e}^{\lambda x_k} + \frac{1}{6}(\lambda h_k)^3 \mathrm{e}^{\lambda x_k} + \dots \\ =& \mathrm{e}^{\lambda x_k}\left[1 + \lambda h_k + \frac{1}{2}(\lambda h_k)^2 + \frac{1}{6}(\lambda h_k)^3 + \dots\right] \\ =& y(x_k)\left[1 + \lambda h_k + \frac{1}{2}(\lambda h_k)^2 + \frac{1}{6}(\lambda h_k)^3 + \dots\right]. \end{aligned} \tag{10.33}$$

Let us assume that $y(x_k) = y_k$, then $L_k = y_{k+1} - y(x_{k+1})$. Subtracting (10.32) and (10.33), we obtain

$$L_k = y_{k+1} - y(x_k + h_k) = \frac{1}{2}(\lambda h_k)^2 + \dots = O(h_k^2),$$

which means the implicit Euler method has the order 1.

10.4.2 Crank-Nicolson method

The Crank-Nicolson method (trapezoid method) is formaly "an average" between the explicit and implicit Euler methods, namely

$$y_{k+1} = y_k + \frac{1}{2}h_k\left(f(x_k, y_k) + f(x_{k+1}, y_{k+1})\right), \quad k = 0, 1, \dots, N-1. \tag{10.34}$$

Obviously, the method is implicit.

We investigate the linear stability of the Crank-Nicolson method (10.34). We consider the linear ODE

$$y' = \lambda y, \qquad \lambda \in \mathbb{R}. \tag{10.35}$$

Applying the Crank-Nicolson method, we obtain

$$y_{k+1} = y_k + \frac{\lambda h_k}{2}(y_k + y_{k+1}),$$

which gives

$$y_{k+1} = A y_k, \qquad A := \frac{1 + \frac{\lambda h_k}{2}}{1 - \frac{\lambda h_k}{2}}, \tag{10.36}$$

where A is the amplification factor. Obviously, if $\lambda < 0$ (i.e. system (10.35) is stable) then $|A| < 1$ for any $h_k > 0$. Hence, the Crank-Nicolson method is unconditionally stable. We note that formula like (10.31) can by derived also for the Crank-Nicolson method with the amplification factor given by (10.36).

Finally, we investigate the accuracy of the Crank-Nicolson method. We consider again the ODE (10.35). The Crank-Nicolson method and the expand into a series give

$$\begin{aligned} y_{k+1} = y_k \frac{1+\frac{\lambda h_k}{2}}{1-\frac{\lambda h_k}{2}} =& y_k \left(1+\frac{\lambda h_k}{2}\right)\left[1+\lambda h_k/2+(\lambda h_k/2)^2+(\lambda h_k/2)^3+\dots\right] \\ =& y_k\left[1+\lambda h_k+(\lambda h_k)^2/2+(\lambda h_k)^3/4+\dots\right]. \end{aligned} \tag{10.37}$$

Moreover, from (10.33), we have

$$y(x_k+h_k) = y(x_k)\left[1+\lambda h_k+\frac{1}{2}(\lambda h_k)^2+\frac{1}{6}(\lambda h_k)^3+\dots\right]. \tag{10.38}$$

Let us assume that $y(x_k)=y_k$, then $L_k = y_{k+1}-y(x_{k+1})$ Subtracting (10.37) and(10.38), we obtain

$$L_k = y_{k+1}-y(x_k+h_k) = \frac{1}{12}(\lambda h_k)^3+\dots = O(h_k^3),$$

which means the Crank-Nicolson method is the order 2.

We conclude that the Crank-Nicolson method is unconditionally stable as the implicit Euler method and moreover, it is more accurate.

10.4.3 Implicit method for stiff problems

We consider again the problem (10.24) written as

$$\begin{aligned} u' &= 998u+1998v, & u(0)&=1, \\ v' &= -999u-1999v, & v(0)&=0. \end{aligned} \tag{10.39}$$

We are interested in the solution of u and v at $x=10$. We solve (10.39) by the explicit as well as implicit Euler method.

From (10.25), the stability of the explicit Euler method is guaranteed for $h<10^{-3}$. On the other hand, the implicit Euler method is unconditionally stable but we need to solve a (2×2) linear algebraic problem.

For each method, we use several time step, the results are given in the following tables: **Demonstrate by the computer**

explicit

h	$u_h(10)$	$v_h(10)$
10^{-3}	9.04E-05	-4.52E-05
$2\cdot 10^{-3}$	oscillates	oscillates
10^{-2}	∞	∞

implicit

h	$u_h(10)$	$v_h(10)$
10^{-3}	9.13E-05	-4.56E-05
10^{-2}	9.63E-05	-4.82E-05
10^{-1}	1.60E-04	-7.98E-05
10^{0}	1.95E-04	-9.77E-04

Let us note that the exact solution is $u(10)$ =9.08E-5 and $v(10)$ =-4.04E-5.

We oberve that

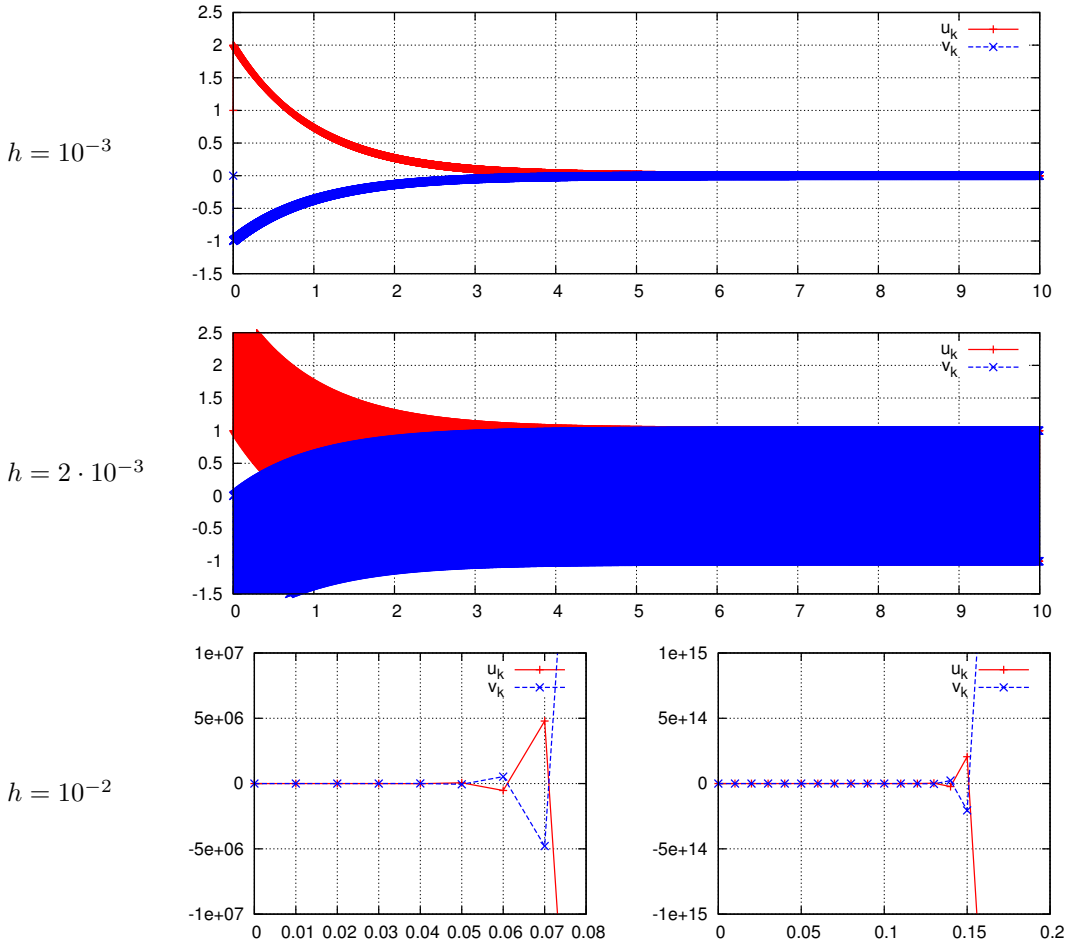


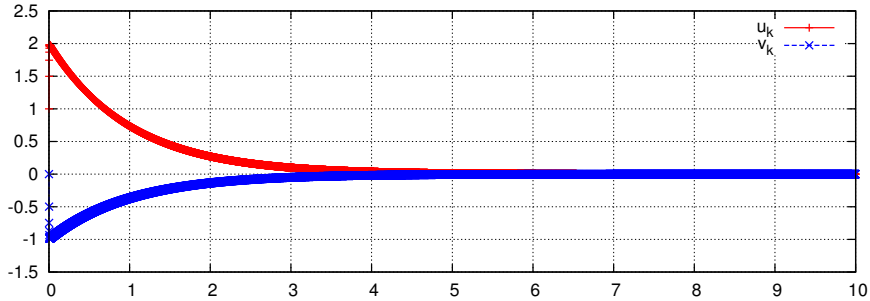
Figure 10.5: Approximate solution of (10.39) by the explicit Euler method.

- Explicit method works only for $h = 10^{-3}$, i.e., only if the stability condition is satisfied. However, when for $h = 10^{-3}$, the approximate solution well corresponds to the exact one.
- Implicit method works for all tested h. For $h = 10^{-3}$, the approximate solution is very similar to the approximate solution obtained with the aid of the implicit method.
- Increasing h for implicit method leads to an increase of the error. However, in situation, when we do not care about the accuracy, a large h can be used.

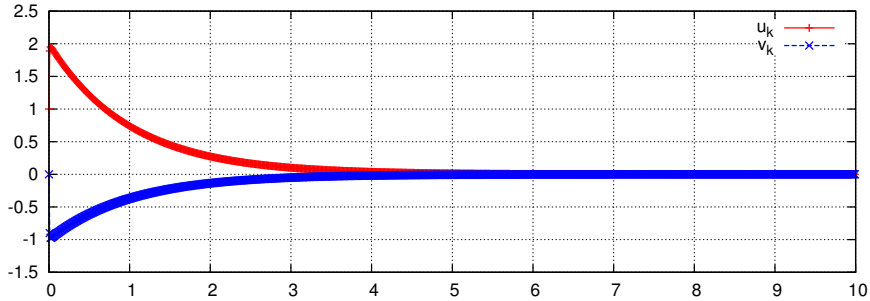
Figures 10.5 and 10.6 show the approximate solution obtained with the aid of the explicit and implicit Euler methods, respectively, for several time steps h.

Do Exercise 19

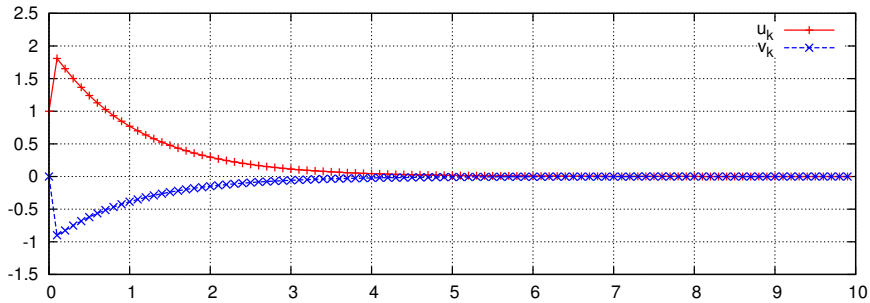
$h = 10^{-3}$



$h = 10^{-2}$



$h = 10^{-1}$



$h = 10^{0}$

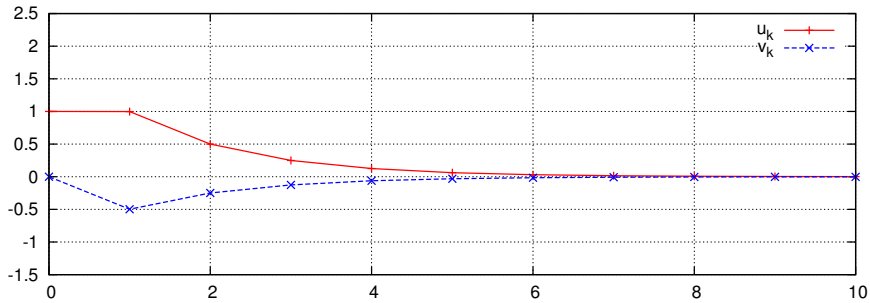


Figure 10.6: Approximate solution of (10.39) by the implicit Euler method.

10.5 Numerical methods used in public software

In public numerical software, two types of numerical methods are mostly used:

- explicit Runge-Kutta methods for the solution of non-stiff problems,
- implicit multi-step formulae for the solution of stiff problems.

10.5.1 Runge-Kutta methods

Let $s \in \mathbb{N}$, then the explicit Runge-Kutta methods can be written in the form

$$y_{k+1} = y_k + h_k \sum_{i=1}^{s} w_i \mathrm{k}_i, \tag{10.40}$$

where

$$\begin{aligned}
\mathrm{k}_1 &= f(x_k, y_k), \\
\mathrm{k}_2 &= f(x_k + \alpha_2 h_k, y_k + \beta_{21}\mathrm{k}_1), \\
&\vdots \\
\mathrm{k}_i &= f(x_k + \alpha_i h_k, y_k + h_k \sum_{j=1}^{i-1} \beta_{ij}\mathrm{k}_j), \quad i = 1, \dots, s,
\end{aligned}$$

and w_i, $i = 1, \dots, s$, α_i, $i = 2, \dots, s$, and β_{ij}, $j = 1, \dots, i-1$, $i = 1, \dots, s$ are real coefficients. Sometimes we speak about s-stage method. Obviously, method (10.40) is explicit, we do not need to solve any algebraic system. On the other hand, at each time step, the function f has to be evaluated s-times.

The Runga-Kutta methods are one step methods, hence they are the so-called self-starting. Therefore, it is possible to adapt the time step in a simple way.

Example 10.11. *a) For $s = 1$, the Runge-Kutta method is equivalent to the explicit Euler method.*
b) The simplest (non-trivial) method is Heun's method

$$\begin{aligned}
y_{k+1} &= y_k + \frac{h_k}{2}(\mathrm{k}_1 + \mathrm{k}_2), \\
\mathrm{k}_1 &= f(t_k, y_k), \\
\mathrm{k}_2 &= f(t_k + h_k, y_k + h_k \mathrm{k}_1),
\end{aligned}$$

which has the order equal to 2, see Definition 10.9.

c) The standard Runge-Kutta method ($s = 4$) reads

$$\begin{aligned} y_{k+1} &= y_k + \frac{h_k}{6}(\mathrm{k}_1 + 2\mathrm{k}_2 + 2\mathrm{k}_3 + \mathrm{k}_4), \\ \mathrm{k}_1 &= f(t_k, y_k), \\ \mathrm{k}_2 &= f(t_k + \frac{1}{2}h_k, y_k + \frac{1}{2}h_k\mathrm{k}_1), \\ \mathrm{k}_3 &= f(t_k + \frac{1}{2}h_k, y_k + \frac{1}{2}h_k\mathrm{k}_2), \\ \mathrm{k}_4 &= f(t_k + h_k, y_k + h_k\mathrm{k}_3), \end{aligned}$$

it has the order equal to 4.

Lemma 10.12. *If $s \leq 4$ then there exists a Runge-Kutta method of the order $p = s$. Moreover, if a Runge-Kutta method has the order $p > 4$ then $s > p$.*

This lemma implies that the Runge-Kutta method with $s = p = 4$ is optimal from the point of view of the accuracy as well as the efficiency.

10.5.2 Multi-step methods

Let $m \in \mathbb{N}$, then a m-step method can be written in the form

$$\sum_{i=0}^{m} \alpha_i y_{k+i} = h \sum_{i=0}^{m} \beta_i f_{k+i}, \qquad f_j = f(x_j, y_j),\ j = 0, 1, \ldots \tag{10.41}$$

where $\alpha_i,\ \beta_i,\ i = 0, \ldots, m$ are the real parameters. If $\beta_m = 0$ then method (10.41) is explicit, otherwise, it is implicit. Obviously, the implicit methods are much more stable (but not always unconditionally stable), i.e., the time step h can be chosen much more larger than for the explicit ones.

The values $y_0, \ldots, y_{m-1}$ are not given by (10.41), they can be evaluated, e.g., by a one-step method. We say that (10.41) is not self-starting.

Example 10.13. *a) The explicit and implicit Euler methods are a special case of (10.41) for $m = 1$.*
b) The one-step implicit method (= Crank-Nicolson) reads

$$y_{k+1} = y_k + \frac{h}{2}\left(f_{k+1} + f_k\right),$$

its order is 2.
c) The two-steps explicit method reads

$$y_{k+1} = y_k + \frac{h}{2}\left(3f_k - f_{k-1}\right),$$

its order is 1.
c) The two-steps implicit method reads

$$y_{k+1} = y_k + \frac{h}{12}\left(5f_{k+1} + 8f_k - f_{k-1}\right),$$

its order is 2.
d) Further examples are the so-called Adams-Bashforth methods, which are explicit and m-step method has the order m, e.g.,

$$y_{k+1} = y_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}).$$

e) Moreover, the so-called Adams-Moulton methods are implicit and the m-step method has the order $m+1$, e.g.,

$$y_{k+1} = y_k + \frac{h}{24}(9f_{k+1} + 19f_k - 5f_{k-1} + f_{k-2}).$$

Implicit multi-step methods

The implicit multi-step methods leads to a nonlinear algebraic system, which has to be solved at each time step. It is possible to use:

- direct solution of the nonlinear algebraic system, e.g., by the Newton or a Newton-like method. Then the Jacobi matrix of f (or its approximation) has to be evaluated. Usually, either an explicit evaluation of the Jacobi matrix by user is required or the Jacobi matrix can be evaluated numerically by differentiation (efficiency is usually low).

- predictor-corrector technique, where a combination of an explicit and an implicit method is used. The explicit method gives an approximation of y_{k+m} denoted by $y^\star_{k+m}$, we put $y^0_{k+m} := y^\star_{k+m}$ and replace (10.41) by

$$\alpha_m y^{\ell+1}_{k+m} + \sum_{i=0}^{m-1} \alpha_i y_{k+i} = h\left(\beta_m f(t_{k+m}, y^\ell_{k+m}) + \sum_{i=0}^{m-1} \beta_i f_{k+i}\right), \quad \ell = 0, 1, \ldots. \tag{10.42}$$

  The formula (10.42) is explicit and the approximations y^0_{k+m}, y^1_{k+m}, y^2_{k+m},... should converge to y_{k-m} given by (10.41). The convergence follows from the pix-point theorem based on some assumptions on f and h. Usually, it is sufficient to carry out 1 or 2 iterations in (10.42).

Let us note that the use of the predictor-corrector technique leads to a larger restriction on the size of the time step than the use of the direct solution of the nonlinear algebraic system.

Backward difference formulae

A special class of the multi-step methods are the backward difference formulae (BDF) given by

$$\sum_{i=0}^{m} \alpha_i y_{k+i} = h f_{k+m}, \qquad f_{k+m} = f(t_{k+m}, y_{k+m}), \tag{10.43}$$

where $\alpha_i,\ i = 0, \ldots, m$ are the real parameters.

Example 10.14. *Examples of BDF methods are:*

$$\begin{aligned} y_{k+1} - y_k &= h f_{k+1}, \\ \frac{3}{2} y_{k+2} - 2 y_{k+1} + \frac{1}{2} y_k &= h f_{k+2}, \\ \frac{11}{6} y_{k+3} - 3 y_{k+2} + \frac{3}{2} y_{k+1} - \frac{1}{3} y_k &= h f_{k+3}. \end{aligned}$$

m-step BDF has the order m, for $m > 6$ the method is unstable.

It is possible to say that the BDF is the class of the most suitable methods for the solution of stiff systems.

10.5.3 Comparison of Runge-Kutta methods and multi-step methods

	Runge-Kutta	explicit multi-step	implicit multi-step
self-starting	yes	no	no
time step adaptation	simple	complicated	complicated
stability	low	low	high
computational costs	low	the lowest	high
suitable for stiff	no	no	yes

10.5.4 Other numerical methods for ODE

- multivalue methods – some generalization of multi-step methods, which allows a simple time step adaptation, an adaptive choice of the order of the method and a "switcher" between explicit and implicit approach.
- numerical differentiation formulae – similar to BDF methods, suitable for stiff problems, used in Matlab (ode15s).
- time discontinuous Galerkin method – very accurate, flexible but expensive.

10.6 Estimates of the local error

The estimate of the local error arising in the numerical solution of ODEs is crucial for an efficient and accurate solution, see Section 10.3.2. We have already mentioned that we are unable to estimate the global error G_k but only the local error L_k. We present three approaches of the estimate of the local error.

10.6.1 Error estimates based on the interpolation function

Let $y_0,\ y_1,\ y_2, \dots, y_k$ be the known approximation of y at $x_0,\ x_1,\ x_2, \dots, x_k$. Let $n \in \mathbb{N}$, $n \le k$ be given. We define a function $z \in C^1([y_{k-n}, y_k])$ such that

$$z(x_{k-i}) = y_{k-i}, \quad i = 0, \dots, n.$$

Then we define the function

$$r(x) := z'(x) - f(x, z(x)), \quad x \in [y_{k-n}, y_k],$$

which represents a residuum of the approximate solutions $y_0,\ y_1,\ y_2, \dots, y_k$.

A natural requirement is $|r(x)| \le \mathrm{TOL}$, where TOL is a given tolerance. This approach is general, however a suitable tolerance TOL have to be chosen empirically.

10.6.2 Technique of half-size step

Theorem 10.15 (W. B. Gragg (1963)). *Let us consider ODEs (10.1) on $[a,b] \times \mathbb{R}^m$, let $f \in C^{N+2}([a,b] \times \mathbb{R}^m)$, $N \in \mathbb{N}$. Let $y(x,h)$ denote the approximate solution at $x \in [a,b]$ obtained by a one-step method of order p ($p \le N$) with the time step $h \in (0, \bar{h})$. Then there exists functions $e_i : [a,b] \to \mathbb{R}^m$, $i = p, \dots, N$ and $E_{N+1} : [a,b] \times (0,\bar{h}) \to \mathbb{R}^m$ such that*

$$y(x) = y(x,h) + h^p e_p(x) + h^{p+1} e_{p+1}(x) + \cdots + h^N e_N(x) + h^{N+1} E_{N+1}(x,h). \tag{10.44}$$

Moreover,

1) $e_k(a) = 0, \quad k = p, p+1, \dots, N,$
2) $e_k(x)$ *is independent of* $h,\ k = p, p+1, \dots, N,$
3) $\exists C(x) : [a,b] \to \mathbb{R} : \ |E_{N+1}(x,h)| \le C(x) \quad \forall x \in [a,b]\ \forall h \in (0,\bar{h}).$

Let us note the this theorem does not take into account the rounding errors.

This theorem implies that the global error satisfies

$$e(x,h) = y(x) - y(x,h) = h^p e_p(x) + O(h^{p+1}).$$

If $h \ll 1$ then the term $O(h^{p+1})$ can be neglected.

The Gragg theorem can be used in the estimated of the error by the half-size step technique. We have

$$\left.\begin{array}{c} y(x)-y(x,h)\approx h^p e_p(x) \\ y(x)-y(x,h/2)\approx (h/2)^p e_p(x) \end{array}\right\} \Longrightarrow y(x)-y(x,h/2)\approx \frac{y(x,h/2)-y(x,h)}{2^p-1}.$$

Therefore, carrying out the computation two times with the step h and $h/2$, we estimate the error by the difference of both approximate solutions multiplied by the factor $(2^p-1)^{-1}$.

10.6.3 Runge-Kutta-Fehlberg methods

Fehlberg proposed the error estimate based on the use of two Runge-Kutta formula

$$y_{k+1} = y_k + h_k \sum_{i=1}^{s} w_i \mathrm{k}_i, \tag{10.45a}$$

$$\hat{y}_{k+1} = y_k + h_k \sum_{i=1}^{s} \hat{w}_i \mathrm{k}_i, \tag{10.45b}$$

$$\mathrm{k}_i = f(x_k + \alpha_i h_k, y_k + h_k \sum_{j=1}^{i-1} \beta_{ij} \mathrm{k}_j), \quad i = 1, \dots, s,$$

where $w_i, \hat{w}_i,\ i = 1, \dots, s,\ \alpha_i,\ i = 2, \dots, s$, and $\beta_{ij},\ j = 1, \dots, i-1,\ i = 1, \dots, s$ are suitable real coefficients. The menthod (10.45a) has the order equal to p and method (10.45b) has the order equal to $p+1$.

In order to estimate the error, we assume that

$$|\hat{y}_{k+1} - u_k(x_{k+1})| \ll |y_{k+1} - u_k(x_{k+1})|$$

where u_k is the function satisfying (10.1a) with the condition $u_k(x_k) = y_k$, see the definition of the locall error (10.12).

Then we estimate the local error by the formula

$$L_{k+1} \approx y_{k+1} - \hat{y}_{k+1} = h_k \sum_{i=1}^{s} (w_i - \hat{w}_i) \mathrm{k}_i.$$

10.7 Adaptive choice of the time step

In order to efficiently solve ODEs, the steps $h_k,\ k = 1, \dots, m$ have to be chosen such that

- it is sufficiently small in order to guarantee the stability and the accuracy,
- it is sufficiently large in order to achieve an efficiency.

10.7.1 Basic idea

Now, we introduce some technique which propose an optimal time step h_k^{opt} which should fulfil both previous requirements. Let EST is an estimate the absolute value of the local error L_k, see Section 10.6. In order to balance between the accuracy and the efficiency, we (usually) choose h_k such that

$$\begin{array}{rlll} \text{either} & \text{EST} \approx \text{TOL}, & k = 1, \dots, N, & \text{E.P.S. (error per step)} \\ \text{or} & \text{EST} \approx h_k \text{TOL}, & k = 1, \dots, N, & \text{E.P.U.S. (error per unit step)} \end{array} \tag{10.46}$$

where $\text{TOL} > 0$ is a suitably chosen tolerance. Let us underline that EST depends on h_k.

Let us consider an numerical method of order p for the solution of (10.1). Then $L_k = O(h^{p+1})$, see Definition 10.9. We assume that

$$\text{EST} = \text{EST}(h_k) = C\, h_k^{p+1},$$

where $C > 0$ is a constant.

Then taking into the account (10.46) (for E.P.S.), the optimal time step h_k^{opt} satisfies

$$\text{TOL} = C\, (h_k^{\text{opt}})^{p+1}.$$

From two last relations we have

$$\frac{\text{TOL}}{\text{EST}} = \left(\frac{h_k^{\text{opt}}}{h_k} \right)^{p+1} \quad \Rightarrow \quad h_k^{\text{opt}} = h_k \left(\frac{\text{TOL}}{\text{EST}} \right)^{\frac{1}{p+1}}. \tag{10.47}$$

For E.P.U.S, the relation reads

$$\frac{h_k \text{TOL}}{\text{EST}} = \left(\frac{h_k^{\text{opt}}}{h_k} \right)^{p+1} \quad \Rightarrow \quad h_k^{\text{opt}} = h_k \left(\frac{h_k \text{TOL}}{\text{EST}} \right)^{\frac{1}{p+1}}. \tag{10.48}$$

10.7.2 Practical implementations

The relation (10.47) gives us the optimal size of the time step h_k. Based on numerical experiments several "improving strategies" were proposed. We present and comment here some of them:

- relation (10.47) is replaced by

$$h_k^{\text{opt}} = \begin{cases} h_k \left(\frac{\text{TOL}}{\text{EST}} \right)^{\frac{1}{p+1}} & \text{if } \text{TOL} \geq \text{EST}, \\ h_k \left(\frac{\text{TOL}}{\text{EST}} \right)^{\frac{1}{p}} & \text{if } \text{TOL} < \text{EST}. \end{cases}$$

  It means that if the estimate is bigger than the tolerance then the optimal step is shorter than the step given by (10.47).

- The size of the optimal step is multiplied by an security factor FAC, i.e., (10.47) is replaced by

$$h_k^{\text{opt}} = \text{FAC} h_k \left(\frac{\text{TOL}}{\text{EST}} \right)^{\frac{1}{p+1}},$$

  where $\text{FAC} \in (0,1)$, e.g., $\text{FAC} = 0.8$ or $\text{FAC} = 0.9$ or $\text{FAC} = (0.25)^{1/(p+1)}$.

- we limit the variations of the time step, e.g., we require that

$$\text{FACMIN} \leq \frac{h_k^{\text{opt}}}{h_K} \leq \text{FACMAX},$$

  where, e.g., $\text{FACMIN} \in (0.1, 0.5)$ and $\text{FACMAX} \in (1.5, 10)$. Moreover, the values FACMIN and FACMAX can be modified during the computation.

10.7.3 Choice of the first time step

In order to perform the first time step, we have to set h_1.

- by the user of the software, if he can provide this information,
- automatically by a prescribed fixed value,
- choice proposed by [H. Watts (1983)]:

$$h_1 = \left(\frac{\text{TOL}}{A} \right)^{1/(p+1)}, \quad A = \left(\frac{1}{\max(|a|, |b|)} \right)^{p+1} + \|f(a, y(a))\|^{p+1}.$$

- a use of a simple (e.g. Euler) method in order to find optimal h_0.

10.7.4 Abstract algorithm for the solution of ODE

1. check of input data,
2. initialization of h_0 and FLAG (indicator of a successfully performed time step),
3. for $k = 1, 2, \ldots,$
   (a) preparation of the new time step h_k (depends on FLAG), saving of data, etc.,
   (b) computation of y_{k+1},
   (c) estimation of the local error EST, setting of FLAG (if $\text{EST} \leq \text{TOL}$ then FLAG is OK), setting h_k^{opt},

(d) if EST $\leq$ TOL then

. successful time step:

. if $x_k + h \geq b$ then end of computation

. else

. $y_k := y_{k+1}$, $x_k := x_k + h_k$, $k := k + 1$,

. propose of h_k (usually $h_k := h_k^{\text{opt}}$)

. endif

else

. unsuccessful time step: generally repeat step k with $h_k := h_k^{\text{opt}}$

endif

10.8 Subroutine RKF45

10.8.1 Overview

Subroutine for the numerical solution of (10.1) with the aid of the Runge-Kutta-Fehlberg method of order 4 and 5. It suits for non-stiff and middle-stiff systems.

It carried out the solution from `T` to `TOUT` and return the value y at `TOUT`. Therefore, if a visualization of y is desired, RKF45 should be called several times, see file `dr_rkf45.f`.

10.8.2 Input/output parameters

The subroutine RKF45 is called by the command

```
call RKF45(F,NEQN,Y,T,TOUT,RELERR,ABSERR,IFLAG,WORK,IWORK)
```

where the input/output parameters are the following

```
FUNC -- SUBROUTINE FUNC(T,Y,YP) TO EVALUATE DERIVATIVES YP(I)=DY(I)/DT
NEQN -- NUMBER OF EQUATIONS TO BE INTEGRATED
Y(*) -- SOLUTION VECTOR AT T
T -- STARTING POINT OF INTEGRATION ON INPUT
     LAST POINT REACHED IN INTEGRATION ON OUTPUT
TOUT -- OUTPUT POINT AT WHICH SOLUTION IS DESIRED
RELERR,ABSERR -- RELATIVE AND ABSOLUTE ERROR TOLERANCES FOR LOCAL
      ERROR TEST. AT EACH STEP THE CODE REQUIRES THAT
           ABS(LOCAL ERROR) .LE. RELERR*ABS(Y) + ABSERR
      FOR EACH COMPONENT OF THE LOCAL ERROR AND SOLUTION VECTORS
IFLAG -- INDICATOR FOR STATUS OF INTEGRATION
         +1, -1 ON INPUT (-1 ONLY ONE TIME STEP)
         +2, -2 SUCCESSFUL COMPUTATION
         3 - 8  SOME TROUBLES
WORK(*) -- ARRAY TO HOLD INFORMATION INTERNAL TO RKF45 WHICH IS
      NECESSARY FOR SUBSEQUENT CALLS. MUST BE DIMENSIONED
      AT LEAST  3+6*NEQN
IWORK(*) -- INTEGER ARRAY USED TO HOLD INFORMATION INTERNAL TO
      RKF45 WHICH IS NECESSARY FOR SUBSEQUENT CALLS. MUST BE
      DIMENSIONED AT LEAST  5
```

10.8.3 Installation and use of RKF45

- Archive can be downloaded from
  `http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/RKF45.tgz`

- after unpacking of the file (in Linux by `tar xfz RKF45.tgz`), an archive with the following files appears:
  - `makefile` – makefile for translation
  - `rkf45.f` – the subroutine RKF45
  - `dr_rkf45.f` – the main program calling RKF45 and containing the definition of the input parameters
- the code can be translated by the command `make` (if it is supported) which use the file `makefile`

```
dr_rkf45   : dr_rkf45.o rkf45.o
        gfortran -o dr_rkf45 dr_rkf45.o rkf45.o
dr_rkf45.o : dr_rkf45.f
        gfortran -c  dr_rkf45.f
rkf45.o : rkf45.f
        gfortran -c  rkf45.f
```

  or by a direct use of previous commands, namely

```
f77 -c dr_rkf45.f
f77 -c rkf45.f
f77 -o dr_rkf45 dr_rkf45.o RKF45.o
```

  The symbol `f77` denotes the name of the translator, it can be replaced by any available fortran translator (`g77`, `gfortran`, ...). If the translation is successful, the executable file `dr_rkf45` arises.

- the setting of the input parameters has to be done by hand in the file `dr_rkf45.f` (the translation has to be repeated thereafter):

```
      PROGRAM DR_RKF45

      PARAMETER (NEQN = 2)
      INTEGER IFLAG,IWORK(5),NSTEP, I
      DOUBLE PRECISION WORK(3+6*NEQN), Y(NEQN),T,TOUT,RELERR,ABSERR,
     *     A, B, STEP

C     NUMBER OF EQUATIONS
c      NEQN = 1

C     INITIALIZATION
      IFLAG = 1
```

```
C     DESIRED ACCURACY
      RELERR = 1.D-08
      ABSERR = 1.D-08

C     INITIAL CONDITION
      Y(1) = 1.0D+00
c      Y(2) = 1.0D+00

C     INTERVAL OF INVESTIGATION
      A = 0.D+00
      B = 5.D+00

C     NUMBER OF STEP ON <A,B>
      NSTEP = 200
      STEP = (B-A) / NSTEP

c     STARTING POINT OF INTEGRATION
      T = A

      if( NEQN == 2) then
         WRITE(*,'(3e14.6,i5)') T,Y(1),Y(2),IFLAG
      else
         WRITE(*,'(2e14.6,i5)') T,Y(1),IFLAG
      endif

      DO I=1,NSTEP
         TOUT = T + STEP
         CALL RKF45(F,NEQN,Y,T,TOUT,RELERR,ABSERR,IFLAG,WORK,IWORK)


         if( NEQN == 2) then
            WRITE(*,'(3e14.6,i5)') T,Y(1),Y(2),IFLAG
         else
            WRITE(*,'(2e14.6,i5)') T,Y(1),IFLAG
         endif

         T = TOUT
      ENDDO
      END

      SUBROUTINE FUNC(NEQN,T,Y,YP)
```

```
      DOUBLE PRECISION  T,Y(NEQN),YP(NEQN)
      DOUBLE PRECISION alpha


      YP(1) = 998.0D+00 * Y(1) + 1998.D+00 * Y(2)
      YP(2) = -999.0D+00 * Y(1) - 1999.D+00 * Y(2)

      RETURN
      END
```

- the code is run by `./dr_rkf45`, the output looks like

```
  0.000000E+00  0.100000E+01  0.691704-309    1
  0.250000E-01  0.195062E+01 -0.975310E+00    2
  0.500000E-01  0.190246E+01 -0.951229E+00    2
  0.750000E-01  0.185549E+01 -0.927743E+00    2
  0.100000E+00  0.180967E+01 -0.904837E+00    2
  0.125000E+00  0.176499E+01 -0.882497E+00    2
  0.150000E+00  0.172142E+01 -0.860708E+00    2
  0.175000E+00  0.167891E+01 -0.839457E+00    2
  0.200000E+00  0.163746E+01 -0.818731E+00    2
  0.225000E+00  0.159703E+01 -0.798516E+00    2
  0.250000E+00  0.155760E+01 -0.778801E+00    2
  0.275000E+00  0.151914E+01 -0.759572E+00    2
  0.300000E+00  0.148164E+01 -0.740818E+00    2
  0.325000E+00  0.144505E+01 -0.722527E+00    2
.
.
.
.
```

The columns are: x_k, $y_{1,k}$, $y_{2,k}$, `IFLAG`

10.9 Subroutine DOPRI5

10.9.1 Overview

Subroutine for the numerical solution of (10.1) with the aid of the Runge-Kutta of order 4 and 5, the so-called Dormand and Prince method. It suits for non-stiff systems.

10.9.2 Input/output parameters

The subroutine DOPRI5 is called by the command

```
call DOPRI5(N,FCN,X,Y,XEND,
   &                  RTOL,ATOL,ITOL,
   &                  SOLOUT,IOUT,
   &                  WORK,LWORK,IWORK,LIWORK,RPAR,IPAR,IDID)
```

where the selected input/output parameters are the following

```
    INPUT PARAMETERS
    ----------------
    N           DIMENSION OF THE SYSTEM
    FCN         NAME (EXTERNAL) OF SUBROUTINE COMPUTING THE
                VALUE OF F(X,Y):
                   SUBROUTINE FCN(N,X,Y,F,RPAR,IPAR)
                   DOUBLE PRECISION X,Y(N),F(N)
                   F(1)=...   ETC.
    X           INITIAL X-VALUE
    Y(N)        INITIAL VALUES FOR Y
    XEND        FINAL X-VALUE (XEND-X MAY BE POSITIVE OR NEGATIVE)
    RTOL,ATOL   RELATIVE AND ABSOLUTE ERROR TOLERANCES. THEY
                CAN BE BOTH SCALARS OR ELSE BOTH VECTORS OF LENGTH N.
    ITOL        SWITCH FOR RTOL AND ATOL:
                  ITOL=0: BOTH RTOL AND ATOL ARE SCALARS.
                    THE CODE KEEPS, ROUGHLY, THE LOCAL ERROR OF
                    Y(I) BELOW RTOL*ABS(Y(I))+ATOL
                  ITOL=1: BOTH RTOL AND ATOL ARE VECTORS.
                    THE CODE KEEPS THE LOCAL ERROR OF Y(I) BELOW
                    RTOL(I)*ABS(Y(I))+ATOL(I).
```

10.9.3 Installation and use of DOPRI5

- Archive can be downloaded from
  `http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/DOPRI5.tgz`

- after unpacking of the file (in Linux by `tar xfz DOPRI5.tgz`), an archive with the following files appears:
  - `makefile` – makefile for translation
  - `dopri5.f` – the subroutine DOPRI5
  - `dr_dopri5.f` – the main program calling DOPRI5 and containing the definition of the input parameters
- the code can be translated by the command `make` (if it is supported) which use the file `makefile`

```
dr_dopri5   : dr_dopri5.o dopri5.o
        gfortran -o dr_dopri5 dr_dopri5.o dopri5.o
dr_dopri5.o : dr_dopri5.f
        gfortran -c  dr_dopri5.f
dopri5.o : dopri5.f
        gfortran -c  dopri5.f
```

  or by a direct use of previous commands, namely

```
f77 -c dr_dopri5.f
f77 -c dopri5.f
f77 -o dr_dopri5 dr_dopri5.o DOPRI5.o
```

  The symbol `f77` denotes the name of the translator, it can be replaced by any available fortran translator (`g77`, `gfortran`, ...). If the translation is successful, the executable file `dr_dopri5` arises.

- the setting of the input parameters has to be done by hand in the file `dr_dopri5.f` (the translation has to be repeated thereafter):

```
C * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C --- DRIVER FOR DOPRI5 ON ARENSTORF ORBIT
C * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        IMPLICIT REAL*8 (A-H,O-Z)
        PARAMETER (NDGL=2,NRDENS=2)
        PARAMETER (LWORK=8*NDGL+5*NRDENS+20,LIWORK=NRDENS+20)
        DIMENSION Y(NDGL),WORK(LWORK),IWORK(LIWORK),RPAR(2)
        EXTERNAL FAREN,SOLOUT
C --- DIMENSION OF THE SYSTEM
        N=NDGL
C --- OUTPUT ROUTINE (AND DENSE OUTPUT) IS USED DURING INTEGRATION
        IOUT=2
```

```
C --- INITIAL VALUES AND ENDPOINT OF INTEGRATION
        RPAR(1)=0.012277471D0
        RPAR(2)=1.D0-RPAR(1)
        X=-1.0D0
        Y(1)=0.4D0
        Y(2)=0.0D0
!        Y(3)=0.0D0
!        Y(4)=-2.00158510637908252240537862224D0
!        XEND=17.06521656015796255889172062 49D0
        XEND = 2.
C --- REQUIRED (RELATIVE AND ABSOLUTE) TOLERANCE
        ITOL=0
        RTOL=1.0D-7
        ATOL=RTOL
C --- DEFAULT VALUES FOR PARAMETERS
        DO 10 I=1,20
        IWORK(I)=0
  10    WORK(I)=0.D0
C --- DENSE OUTPUT IS USED FOR THE TWO POSITION COORDINATES 1 AND 2
        IWORK(5)=NRDENS
        IWORK(21)=1
        IWORK(22)=2
C --- CALL OF THE SUBROUTINE DOPRI5
        CALL DOPRI5(N,FAREN,X,Y,XEND,
     &              RTOL,ATOL,ITOL,
     &              SOLOUT,IOUT,
     &              WORK,LWORK,IWORK,LIWORK,RPAR,IPAR,IDID)
C --- PRINT FINAL SOLUTION
!        WRITE (6,99) Y(1),Y(2)
        WRITE (6,*) Y(1)
 99     FORMAT(1X,'X = XEND     Y =',2E18.10)
C --- PRINT STATISTICS
        WRITE (6,91) RTOL,(IWORK(J),J=17,20)
 91     FORMAT('     tol=',D8.2,'   fcn=',I5,' step=',I4,
     &            ' accpt=',I4,' rejct=',I3)
        STOP
        END
C
C
        SUBROUTINE SOLOUT (NR,XOLD,X,Y,N,CON,ICOMP,ND,RPAR,IPAR,IRTRN)
C --- PRINTS SOLUTION AT EQUIDISTANT OUTPUT-POINTS BY USING "CONTD5"
        IMPLICIT REAL*8 (A-H,O-Z)
        DIMENSION Y(N),CON(5*ND),ICOMP(ND),RPAR(2)
```

```
      COMMON /INTERN/XOUT
      IF (NR.EQ.1) THEN
         WRITE (6,99) X,Y(1),Y(2),NR-1
         XOUT=X+2.0D0
      ELSE
 10      CONTINUE
         IF (X.GE.XOUT) THEN
            WRITE (6,99) XOUT,CONTD5(1,XOUT,CON,ICOMP,ND),
     &                   CONTD5(2,XOUT,CON,ICOMP,ND),NR-1
            XOUT=XOUT+2.0D0
            GOTO 10
         END IF
      END IF
 99   FORMAT(1X,'X =',F6.2,'    Y =',2E18.10,'    NSTEP =',I4)
      RETURN
      END
C
      SUBROUTINE FAREN(N,X,Y,F,RPAR,IPAR)
C --- ARENSTORF ORBIT
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION Y(N),F(N),RPAR(2)
      AMU=RPAR(1)
      AMUP=RPAR(2)
!      F(1)=Y(3)
!      F(2)=Y(4)
!      R1=(Y(1)+AMU)**2+Y(2)**2
!      R1=R1*SQRT(R1)
!      R2=(Y(1)-AMUP)**2+Y(2)**2
!      R2=R2*SQRT(R2)
!      F(3)=Y(1)+2*Y(4)-AMUP*(Y(1)+AMU)/R1-AMU*(Y(1)-AMUP)/R2
!      F(4)=Y(2)-2*Y(3)-AMUP*Y(2)/R1-AMU*Y(2)/R2
      F(1) = 1./sin(sqrt(abs(x)))
      F(2) = 0.
      RETURN
      END
```

- the code is run by `./dr_dopri5`, the output looks like

```
 X = -1.00    Y =  0.4000000000E+00  0.0000000000E+00    NSTEP =   0
 X =  1.00    Y =  0.4638986379E+01  0.0000000000E+00    NSTEP =  83
   5.7140503064330046
```

```
tol=0.10D-06   fcn=  728 step= 121 accpt=  86 rejct= 35
```

Homeworks

Exercise 17. *Let us consider two ODEs:*

$$\begin{cases} y' = y, \\ y(0) = 1, \end{cases} \qquad \begin{cases} y' = \exp x, \\ y(0) = 1. \end{cases}$$

Both equations have the exact solution $y(x) = exp(x)$. When we solved them numerically, which of them gives larger problems? Decide about the stability of these ODEs.

Exercise 18. *Write a simple code for the numerical solution of (10.23) by the explicit Euler method with the adaptively chosen time step h_k according (10.21) – (10.22). Demonstrate that the choice (10.21) – (10.22) practically guarantees the stability condition. Hint: use a code from Appendix. Code is available in*
http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/NS_source/ODE/index.html
file `methods.tgz`*, code* `adapt_time.f90`

Exercise 19. *Write a simple code in order to demonstrate the example from Section 10.4.3. Hint: use a code from Appendix. Code is available in the link above, file* `methods.tgz`*, code* `stiff.f90`

Main task 2. *With the aid of codes RKF45 or DOPRI5, compute the following equations:*

(ODE1) $y' = \dfrac{1}{\sin\sqrt{|x|}}, \quad y(-1) = 0, \quad x \in [-1; 2]$

(ODE2) $y' = -\mathrm{sign}(x)|1 - |x||y^2, \quad y(-2) = \dfrac{2}{3}, \quad x \in [-2; 2]$

(ODE3) $y'' = 100y, \quad y(0) = 1, \; y'(0) = -10 \quad x \in [0; 4]$

(ODE4) $u' = 998u + 1998v, \quad u(0) = 1, \qquad x \in [0; 10]$
$v' = -999u - 1999v, \quad v(0) = 0$

(ODE5) $y' = \dfrac{1}{4}\sqrt{y}, \quad y(0) = 0, \quad x \in [0; 5]$

(ODE6) *prey (králíci) k and predator (lišky) l, if $k < 1$ or $l < 1$ then prey or predator die*
$k' = 2k - \alpha kl, \qquad k(0) = k_0, \qquad x \in [0; ?]$
$l' = -l + \alpha kl, \qquad l(0) = l_0, \quad \alpha > 0,$
(a) $\alpha = 0.01$; $k_0 = 300$; $l_0 = 150$ *periodic solution, find the period*
(b) $\alpha = 0.01$; $k_0 = 15$; $l_0 = 22$ *prey die*

(c) $\alpha = 0.01$; *find* k_0 *and* l_0 *when predator die*

(ODE7) $$y' = \begin{cases} -1 & \text{for } y \geq 0 \\ 1 & \text{for } y < 0 \end{cases} \qquad y(0) = 0,\ x \in [0;1],$$

(ODE8) $$y' = \begin{cases} \frac{1}{1+y} & \text{for } 0 < x \leq 2 \\ 1 & \text{for } x > 2 \end{cases} \qquad y(0) = 1,\ x \in [0;5],$$

Remark 10.16.

(ODE1) and (ODE2) contain singularities,

(ODE3) is in fact a system of 2 equations of the first order, explain the arising troubles,

(ODE4) is stiff,

(ODE5) has a non-unique solution, find the non-trivial one,

(ODE6) is a little practical

(ODE7) has discontinuous f*, exists the solution?*

(ODE8) has again discontinuous f*, it may cause some troubles.*

The main task is **not to obtain the correct solution of the ODEs** *but* **to understand the behaviour of the codes**. *For each (ODE1) – (ODE6), carry out the following steps:*

- *predict the possible troubles from the analytical form of the problem,*
- *test both codes with several (at least two tolerances),*
- *based on the results, the error estimate and the indicator decide, if the result is reliable,*
- *in case of some troubles, give an explanation,*
- *perform also a reference computation using a software package as Matlab, Maple, Mathematica, etc. (obligatory)*

Part II

Numerical Software 2 (Summer semester)

Chapter 11

Finite element methods

There are separated Lecture notes (implementation of finite element method) FEM-implement.pdf at https://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/NS2018.html

Chapter 12

Software for FEM

12.1 FreeFEM++

`FreeFem++` is a software to solve partial differential equations numerically. As its name says, it is a free software (see copyright for full detail) based on the Finite Element Method; it is not a package, it is an integrated product with its own high level programming language. This software runs on all unix OS (with g++ 2.95.2 or later, and X11R6) , on Window95, 98, 2000, NT, XP, and MacOS X.

12.2 Fenics

12.2.1 Python

12.2.2 Instalation

12.2.3 Running of available scripts

Chapter 13

Mesh generation and adaptation

13.1 General settings

Let us consider a PDE in $\Omega \subset \mathbb{R}^d$. Let Ω_h be a polygonal approximation of Ω. The set of mutually disjoint closed elements K (denoted by $\mathscr{T}_h = \{K\}_{K \in \mathscr{T}_h}$) is called the mesh of Ω if

$$\overline{\Omega_h} = \cup_{K \in \mathscr{T}_h} K.$$

The parameter h is given by

$$h = \max_{K \in \mathscr{T}_h} \operatorname{diam}(K)$$

and it represents the parameter of discretization. Then the approximate solution is sought in a space of piecewise polynomial functions on $K \in \mathscr{T}_h$.

The elements K, $K \in \mathscr{T}_h$ are suitable geometrical subjects, triangles or quadrilaterals for $d = 2$ or tetrahedra, pyramids, hexahedra and prismatic elements for $d = 3$, see Figure 13.1.

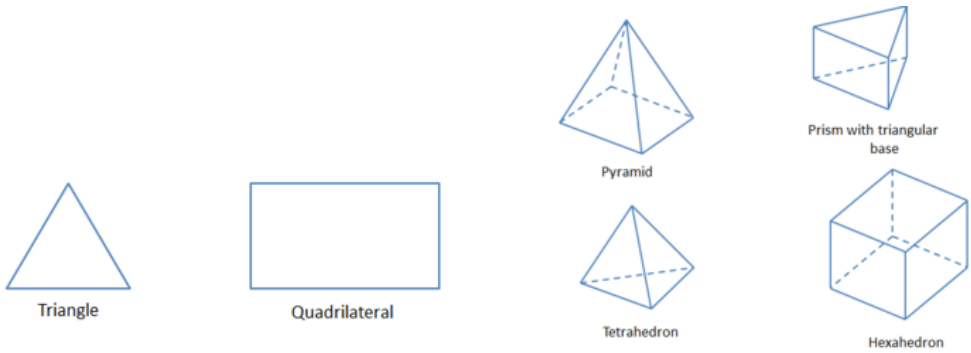


Figure 13.1: Examples of elements for $d = 2$ (left) and $d = 3$ (right).

We distinguish:

- structured grids – having some structures, elements can be indexed by " integer Cartesian coordinates" i, j (for $d = 2$) or i, j, k (for $d = 3$); they are simple for implementation
- unstructured grids – not having structure, better capture complicated domains.

Figure 13.2 shows and example of structured and unstructured meshes for $d = 2$.

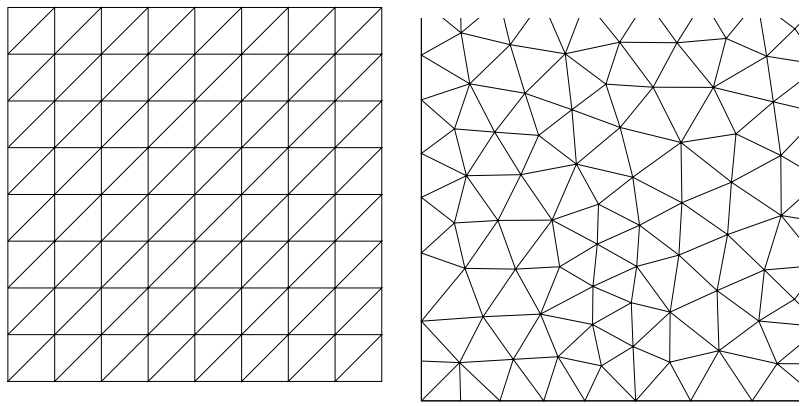


Figure 13.2: Examples of a structured mesh (left) and unstructured one (right).

Depending of the used numerical methods, meshes $\mathscr{T}_h$, $h > 0$ should satisfy some assumptions, e.g.

- maximal angle condition,
- minimal angle condition,
- shape regularity,
- quasi-uniformity,
- local quasi-uniformity,
- conformity (hanging-nodes are prohibited).

In any case, a special attention has to be paid to the approximation of a nonpolygonal boundary, i.e., $\partial\Omega_h \approx \partial\Omega$.

Mesh generation software usually generate shape regular, quasi-uniform and conforming meshes, see Section 13.3. Such grids are used initial solution of the given problem and then they are adapted by mesh adaptation techniques, see Section 13.3.

Mesh generating software produce very often the so-called Delaunay triangulations,i.e., the sum of opposite angles of two neighbouring element is $\leq \pi$. This is equivalent to the

condition that no point of triangulation is inside the circumcircle of any triangle, see Figure 13.3, left.

The Delaunay triangulations is a dual graph to the so-called Voronoi diagram. Voronoi diagram is a way of dividing space into a finite number of regions. Let $P_k,\ K \in I$ be the given set of nodes, we define the sets (the Voronoi cells)

$$R_k := \{x;\ |x - P_k| \leq |x - P_j|\ \forall j \in I,\ j \neq k\}, \quad k \in I,$$

see Figure 13.3, right.

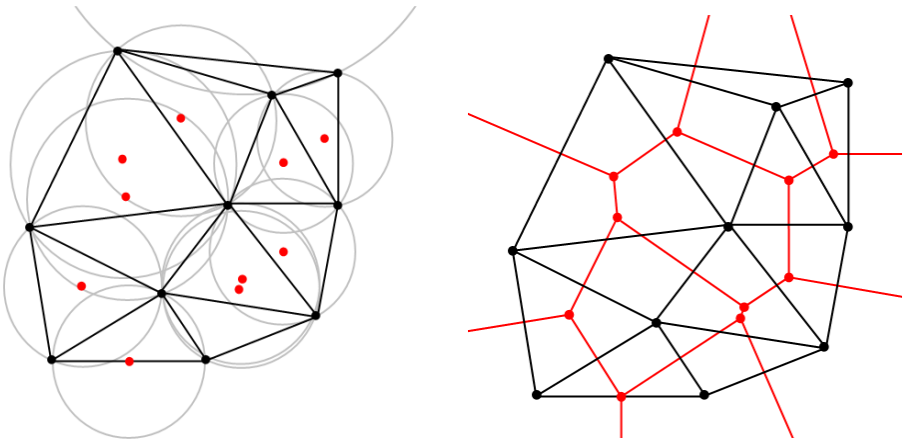


Figure 13.3: The Delaunay triangulation with all the circumcircles and their centers in red (left); Connecting the centers of the circumcircles produces the Voronoi diagram (right).

13.2 Mesh generation

A general strategy of the mesh generation is the following:

1. define the boundary of Ω_h,
2. insert a suitable number of nodes on $\partial\Omega_h$,
3. insert a suitable number of nodes in the interior Ω_h,
4. optimize the mesh.

Optimal mesh means, e.g., that edges of all triangles have the optimal given size h^{opt}, i.e., we define quality parameter of the mesh by

$$Q_{\mathscr{T}_h} = \sum_{\Gamma \in \mathscr{T}_h} (|\Gamma| - h^{\text{opt}})^2,$$

where the sum is taken over all edges of mesh $\mathscr{T}_h$ and $|\Gamma|$ denotes the size of the edge Γ.

The value h^{opt} may not be constant, for example generating a mesh around a circle with center at the origin, we can prescribe $h^{\text{opt}} \sim r$, where r is the polar coordinate of the center of the given edge.

Examples of mesh algorithms:

- overlap generation – we define parallelogram containing Ω, then a structured grid is constructed. Elements outside of the domain are taken away and a correction of the boundary is necessary.
- step advancing algorithm – from the nodes on the boundary an equilateral element is constructed, then the boundary is redefined and the meshing proceed to the new step.

A useful link to mesh generation codes

`http://www.cfd-online.com/Links/soft.html#mesh`

13.3 Mesh adaptation

Mesh adaptation allows to refine (recoarse, align) the given mesh in order to improve the quality of the solution. It is based on

- error estimates indicating the elements suitable for refinement,
- mesh adaptation technique performing the itself refinement.

The usual mesh refinement technique is the red-green refinement method, see Figure 13.4. Moreover, Figure 13.5 shows the possible situation for $d = 3$.

More general approach is the so-called anisotropic mesh adaptation, see

`http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/AMA.pdf`

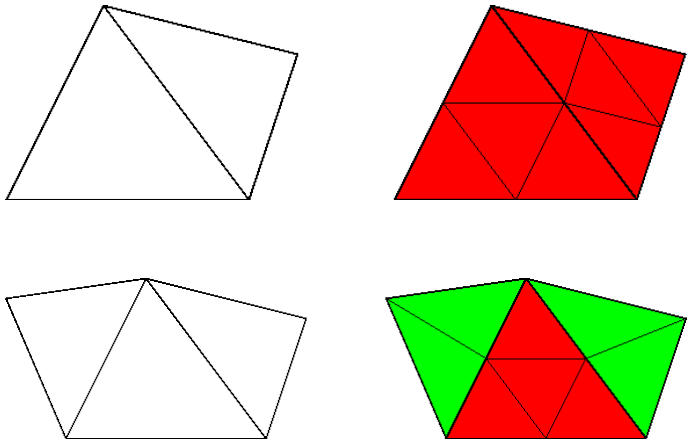


Figure 13.4: A red-refined triangle with a red-refined neighbor (top), A red-refined triangle with green-refined neighbors (right).

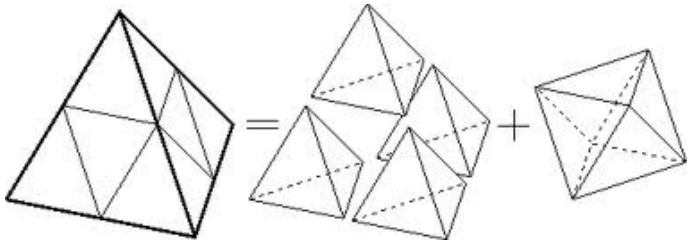


Figure 13.5: A red-refinement for $d = 3$.

13.4 Main tasks: AMA+FEM

Main task 3. *Solve numerically the following PDE:*

$$\begin{aligned} -\Delta u &= 90x_1^8(1-x_2^{20}) + 380x_2^{18}(1-x_1^{10}), \qquad \text{in } \Omega = (0,1)^2, \\ u &= u_D \qquad \text{on } \partial\Omega, \end{aligned} \tag{13.1}$$

where u_D is the exact solution given by

$$u(x_1, x_2) = (1-x_1^{10})(1-x_2^{20}), \quad (x_1, x_2) \in \Omega.$$

Instructions:

1. *Solve problem (13.1) by a suitable numerical method and by an arbitrary code based on your choice. You can used freely available software or you can write a simple own code. Hint: use the code from tutorials:*

   `http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/NS_source/FEM/index.html`

2. *Carry out several adaptation cycles using the code ANGENER, see*

   `http://msekce.karlin.mff.cuni.cz/~dolejsi/angen/angen3.1.htm`

3. *Use a suitable visualization of the adapted grids and the corresponding solutions.*

Main task 4. *We consider the L-shape computational domain* $\Omega := (-1,1) \times (-1,1) \setminus [0,1]^2$. *Solve numerically the following PDE:*

$$\begin{aligned} -\Delta u &= 0, \quad && \text{in } \Omega, \\ u &= u_D && \text{on } \partial\Omega, \end{aligned} \tag{13.2}$$

where u_D *is the exact solution given by*

$$u(r,\phi) = r^{2/3}\sin(2\phi/3)$$

with (r,ϕ) *being the polar coordinates. Owing to the re-entrant corner, this problem features a singularity at the origin such that* $u \in H^{5/3-\epsilon}(\Omega)$, $\epsilon > 0$. *The presence of the singularity does not allow for faster convergence than* $O(h^{2/3})$ *on uniformly refined grids. Instructions:*

1. *Using the code ANGENER, generate a sequence of quasi-uniform grids of* Ω *and set the experimental order of convergence* α *in terms*

   $$\|u - u_h\| \approx ch^{\alpha}. \tag{13.3}$$

   For quasi-uniform grids we have $h \approx C/\sqrt{\#\mathscr{T}_h}$.

2. *Using the code ANGENER in combination with FEM from Main task 3 generate a sequence of adaptively refined grids and set the experimental order of convergence* α *in terms*

   $$\|u - u_h\| \approx c\left(\frac{1}{\sqrt{\#\mathscr{T}_h}}\right)^{\alpha}. \tag{13.4}$$

3. *Use a suitable visualization of the adapted grids and the corresponding solutions.*

Chapter 14

Fast Fourier Transformation

14.1 Problem definition

14.1.1 Fourier transformation

Fourier transformation transforms a function $f(t)$ representing a detected signal in time t to its image $\hat{f}(\omega)$ representing the amplitude of signal with the frequency ω.

Let $f : \mathbb{R} \to \mathbb{C}$ be a function, its *Fourier image* is

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t) \exp[-i\omega t] \mathrm{d}t, \quad \omega \in \mathbb{R}, \tag{14.1}$$

where $i = \sqrt{-1}$ is the imaginary unit. The inverse transformation is given by

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) \exp[i\omega t] \mathrm{d}\omega, \quad t \in \mathbb{R}. \tag{14.2}$$

It is valid $\widehat{\hat{f}} = f$. (Scaling factors can differ in the literature.)

Many applications, e.g., signal processing, (sound, light), solution of some PDE, etc.

14.1.2 Fourier series

If $f : \mathbb{R} \to \mathbb{C}$ is a 2π-periodic function then

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \exp[i\, k\, t],$$

where

$$c_k = \frac{1}{2\pi} \int_0^{2\pi} f(t) \exp[-i\, k\, t] \mathrm{d}t, \qquad k = -\infty, \ldots, \infty.$$

14.1.3 Discrete Fourier transformation

In practice, input data are available only in the discrete nodes. Moreover, using computers, we can deal only with the values in discrete nodes.

Let $N \in \mathbb{N}$ and $\boldsymbol{a} := \{a_k\}_{k=0}^{N-1}$ be given vector, its *discrete Fourier image* is the vector $\boldsymbol{b} := \{b_j\}_{j=0}^{N-1}$ such that

$$b_j = \sum_{k=0}^{N-1} a_k \exp[2\pi\, i\, j\, k/N], \qquad j = 0, \ldots, N-1. \tag{14.3}$$

It is valid that (inverse discrete Fourier transformation)

$$a_k = \frac{1}{N} \sum_{j=0}^{N-1} b_j \exp[-2\pi\, i\, j\, k/N], \qquad k = 0, \ldots, N-1. \tag{14.4}$$

For the given $\boldsymbol{a} \in \mathbb{C}^N$, the evaluation of $\boldsymbol{b} \in \mathbb{C}^N$ requires $O(N^2)$ operation (matrix-vector multiplication) – too expensive. The matrix has the form

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & \ldots & 1 \\
1 & \exp\left[\frac{2\pi i}{N}\right] & \exp\left[\frac{4\pi i}{N}\right] & \exp\left[\frac{6\pi i}{N}\right] & \exp\left[\frac{8\pi i}{N}\right] & \ldots & \exp\left[\frac{2\pi i(N-1)}{N}\right] \\
1 & \exp\left[\frac{4\pi i}{N}\right] & \exp\left[\frac{8\pi i}{N}\right] & \exp\left[\frac{12\pi i}{N}\right] & \exp\left[\frac{16\pi i}{N}\right] & \ldots & \exp\left[\frac{4\pi i(N-1)}{N}\right] \\
1 & \exp\left[\frac{6\pi i}{N}\right] & \exp\left[\frac{12\pi i}{N}\right] & \exp\left[\frac{18\pi i}{N}\right] & \exp\left[\frac{24\pi i}{N}\right] & \ldots & \exp\left[\frac{6\pi i(N-1)}{N}\right] \\
1 & \exp\left[\frac{8\pi i}{N}\right] & \exp\left[\frac{16\pi i}{N}\right] & \exp\left[\frac{24\pi i}{N}\right] & \exp\left[\frac{32\pi i}{N}\right] & \ldots & \exp\left[\frac{6\pi i(N-1)}{N}\right] \\
\vdots & & & & & & \\
1 & \exp\left[\frac{2\pi i(N-1)}{N}\right] & \exp\left[\frac{4\pi i(N-1)}{N}\right] & \exp\left[\frac{6\pi i(N-1)}{N}\right] & \ldots & \ldots & \exp\left[\frac{2\pi i(N-1)(N-1)}{N}\right]
\end{pmatrix}$$

All matrix elements are complex unities: $|\exp[2\pi\, i\, j\, k/N]| = 1 \;\forall j, k = 0, \ldots, N-1$. They acquire only N different values

$$\exp[2\pi i \frac{0}{N}],\ \exp[2\pi i \frac{1}{N}],\ \exp[2\pi i \frac{2}{N}],\ \exp[2\pi i \frac{3}{N}],\ \ldots \exp[2\pi i \frac{N-1}{N}].$$

This property can be employed for the reduction of the number of operations.

14.1.4 Fast (discrete) Fourier transformation (FFT)

FFT reduces the computational costs for (14.3) and (14.4) to $O(N \log_2 N)$ -operations.

	N	1E+01	1E+03	1E+06	1E+09
DFT	N^2	1E+02	1E+06	1E+12	1E+18
FFT	$N \log_2 N$	3E+01	1E+04	2E+07	3E+10

Let N be even. Then

$$b_j = \sum_{k=0}^{N-1} a_k \exp[2\pi\, i\, j\, k/N] \tag{14.5}$$
$$= \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j\, (2k)}{N}\right] + \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j\, (2k+1)}{N}\right],\ j = 0, \ldots, N-1.$$

Thus, we can write the relations for odd and even terms separately. Moreover, we have the relations for the first and the second halves of the b_j.

$$b_j = \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right] + \exp\left[\frac{2\pi\, i\, j}{N}\right] \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right],\ j = 0, \ldots, N/2-1, \tag{14.6}$$

$$b_{N/2+j} = \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right] - \exp\left[\frac{2\pi\, i\, j}{N}\right] \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right],\ j = 0, \ldots, N/2-1.$$

More details for the second relation of (14.6): Let $j \geq N/2$, we put $j' = j - N/2$, and from (14.5) we have

$$b_{N/2+j'} = b_j$$
$$= \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j\, (2k)}{N}\right] + \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j\, (2k+1)}{N}\right]$$
$$= \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, (N/2+j')\, (2k)}{N}\right] + \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, (N/2+j')\, (2k+1)}{N}\right]$$
$$= \sum_{k=0}^{N/2-1} a_{2k} \underbrace{\exp\left[\frac{2\pi\, i\, N/2\, (2k)}{N}\right]}_{=1} \exp\left[\frac{2\pi\, i\, j'\, (2k)}{N}\right] + \sum_{k=0}^{N/2-1} a_{2k+1} \underbrace{\exp\left[\frac{2\pi\, i\, N/2\, (2k+1)}{N}\right]}_{=-1} \exp\left[\frac{2\pi\, i\, j'\, (2k+1)}{N}\right]$$
$$= \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j'\, k}{N/2}\right] - \exp\left[\frac{2\pi\, i\, j'}{N}\right] \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j'\, k}{N/2}\right].$$

In (14.6), we have the same sub-sums (red and blue):

$$b_j = \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right] + \exp\left[\frac{2\pi\, i\, j}{N}\right] \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right],\ j = 0, \ldots, N/2-1,$$
$$b_{N/2+j} = \sum_{k=0}^{N/2-1} a_{2k} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right] - \exp\left[\frac{2\pi\, i\, j}{N}\right] \sum_{k=0}^{N/2-1} a_{2k+1} \exp\left[\frac{2\pi\, i\, j\, k}{N/2}\right],\ j = 0, \ldots, N/2-1.$$

The red sum contains the even terms $a_0, a_2, a_4, \ldots$ and the blue sum contains the odd terms $a_1, a_3, a_5, \ldots$, i.e.,

the red sum contains	$a_0,$	$a_2,$	$a_4,$	$a_6,$	$\ldots$
the blue sum contains	$a_1,$	$a_3,$	$a_5,$	$a_7,$	$\ldots$

Since the red and blue sums for b_j and $b_{N/2+j}$ are the same, it is sufficient to compute each of the sums only one times. Therefore, evaluation of red sum (for each $j = 0, \ldots, N/2-1$) requires $O((N/2)^2)$ operations and for the blue one also $O((N/2)^2)$ operations, together

$$2O((N/2)^2) \text{ operations for the evaluation of each sum}$$

and

$$O(N) \text{ operations for the setting of } b_j \text{ from the computed sums,}$$

i.e., together

$$O(N) + 2O((N/2)^2).$$

If $N/2$ is still even (i.e., N was a multiple of 4) then each of the red and blue sums can be computed by the similar technique, i.e., again by splitting onto two same sums containing "odd" and "even terms":

the first part of red sum contains	$a_0,$	$a_4,$	$a_8,$	$a_{12},$	$\ldots$
the second part of red sum contains	$a_2,$	$a_6,$	$a_{10},$	$a_{14},$	$\ldots$
the first part of blue sum contains	$a_1,$	$a_5,$	$a_9,$	$a_{13},$	$\ldots$
the second part of blue sum contains	$a_3,$	$a_7,$	$a_{11},$	$a_{15},$	$\ldots$

The number of operation is

$$O(N) + 2O(N/2) + 4O((N/4)^2)$$

If $N = 2^M$, we can continue further in a similar way. The number of total operations:

$$O(N) + [O(N/2) + O(N/2)] + [O(N/4) + O(N/4) + O(N/4) + O(N/4)]$$
$$+ \left[\underbrace{O(N/16) + \cdots + O(N/16)}_{16 \text{ times}}\right] \ldots$$
$$+ \left[\underbrace{O(N/2^M) + \cdots + O(N/2^M)}_{2^M = N \text{ times}}\right] = O(NM) = O(N \log_2 N),$$

where M is the number of levels.

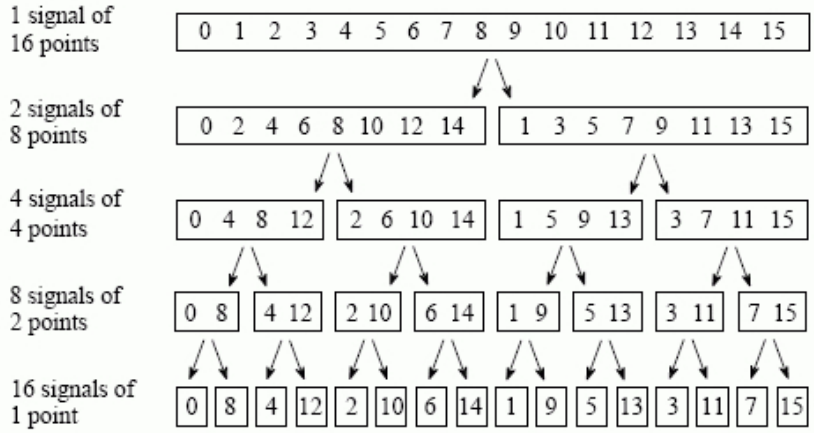


FIGURE 12-2
The FFT decomposition. An N point signal is decomposed into N signals each containing a single point. Each stage uses an *interlace decomposition*, separating the even and odd numbered samples.

Sample numbers in normal order		Sample numbers after bit reversal	
Decimal	*Binary*	*Decimal*	*Binary*
0	0000	0	0000
1	0001	8	1000
2	0010	4	0100
3	0011	12	1100
4	0100	2	0010
5	0101	10	1010
6	0110	6	0100
7	0111	14	1110
8	1000	1	0001
9	1001	9	1001
10	1010	5	0101
11	1011	13	1101
12	1100	3	0011
13	1101	11	1011
14	1110	7	0111
15	1111	15	1111

FIGURE 12-3
The FFT bit reversal sorting. The FFT time domain decomposition can be implemented by sorting the samples according to bit reversed order.

14.2 Implementation

See the file `FourierTrans1.f90`

14.2.1 Using the recursion – simple but not too much efficient

```
subroutine FFTrec(N,a,b,i)
```

14.2.2 Without the recursion – more efficient

```
function FFT(N,a,i)
```

14.2.3 Comparison of the computational times

ONLY example, can differs, the input data are random!

```
CPU time for DFT forward:     40.5200005
CPU time for DFT backward:    40.5879974
CPU time for FFT forward:     1.99966431E-02    (with recursion)
CPU time for FFT backward:    2.40020752E-02    (with recursion)
CPU time for FFT forward:     1.20010376E-02    (without recursion)
CPU time for FFT backward:    1.20010376E-02    (without recursion)
```

14.3 Software for FFT

- N can be arbitrary, computation can be split in more parts than two
- efficiency is increased if N is a high power of small numbers, 2, 3, 5 etc.
- software, e.g., www.netlib.org, search "fft"
- three examples: two advanced `dr_fft.f90`, `dr_fft2.f90` and one simple `dr_fft1.f90`.

Homeworks

Exercise 20. *Implementation of FFT*

- *the web page, link* `Fast Fourier Transformation (FFT)`*, direct link* *msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/NS_source/FFT/index.html*
- *go though the source code to see the algorithmization*
- *use code* `FourierTrans1.f90` *and compare the computational times for DFF, FFT with/without recursion, verify computational costs* $O(N^2)$ *and* $O(N \log_2 N)$

Exercise 21. *More practical examples*

- *install three codes from the archive* `fft.tgz`*, simply use* `make` *command and run the codes using drivers routines*
- *use some of these codes for data from* www.netlib.org/scilib/fft.dat*, you should write a simple code which read data, compare results with the results on this link.*

Chapter 15

Multigrid methods

We introduce only the basis idea of the multigrid methods, advanced ideas and analysis can be found, e.g., in [Hac85].

15.1 Model problem and its FD discretization

We consider the following 1D boundary value problem: we seek $u : (0,1) \to \mathbb{R}$ such that

$$-u''(x) = f(x), \qquad u(0) = u(1) = 0, \tag{15.1}$$

where $f : [0,1] \to \mathbb{R}$ is the given function.

We discretize (15.1) by the *finite difference method* (FDM). Let $N \geq 1$ be the given number of unknowns, we put

$$h := \frac{1}{N+1}, \qquad x_i = h\,i,\ i = 0, \ldots, N+1. \tag{15.2}$$

The second order derivative can be approximated by the relation (following from the Taylor series)

$$u''(x_i) = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2} + O(h^2), \qquad i = 1, \ldots, N. \tag{15.3}$$

The approximate value of $u(x_i),\ i = 0, \ldots, N+1$ is denoted by

$$u_i \approx u(x_i), \quad i = 0, \ldots, N+1. \tag{15.4}$$

Thus, from (15.3) and (15.4) , we have

$$u''(x_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \qquad i = 1, \ldots, N. \tag{15.5}$$

Moreover, we put

$$f_i \approx f(x_i), \quad i = 0, \ldots, N+1. \tag{15.6}$$

Then the FD discretization of (15.1) can be written in the following way. We seek $u_i \in \mathbb{R},\ i = 1, \ldots, N$ such that

$$\frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} = f_i, \qquad i = 1, \ldots, N, \tag{15.7}$$

where $u_0 := 0$ and $u_{N+1} := 0$. Relation (15.7) can be written as the linear algebraic problem

$$\mathbb{A}_h \boldsymbol{u}_h = \boldsymbol{f}_h, \tag{15.8}$$

where

$$\mathbb{A}_h = \{a_{ij}\}_{i,j=1}^N = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}, \quad \boldsymbol{u}_h = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{pmatrix}, \quad \boldsymbol{f}_h = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{pmatrix}, \tag{15.9}$$

cf. (15.4) and (15.6).

15.2 Classical iterative methods

The system (15.8) can be solved efficiently with direct methods. However, let us considered the *classical iterative methods* having the form

$$\boldsymbol{u}_h^{j+1} := \mathbb{M}\boldsymbol{u}_h^j + \mathbb{N}\boldsymbol{f}_h, \quad j = 0, 1, \ldots, \tag{15.10}$$

where $\mathbb{M}$ and $\mathbb{N}$ are obtained from a suitable decomposition of $\mathbb{A}$ and $\boldsymbol{u}_h^j$ denote the j-th approximation of $\boldsymbol{u}_h$. The matrix $\mathbb{M}$ is called the iteration matrix. Let us note that the iterative method (15.10) is converging for any initial choice $\boldsymbol{u}_h^0$ if and only if

$$\rho(\mathbb{M}) < 1 \tag{15.11}$$

where $\rho(\mathbb{M})$ is the spectral radius of $\mathbb{M}$ given by $\rho(\mathbb{M}) := \max_{j=1,\ldots,N} |\lambda_j(\mathbb{M})|$ with $\lambda_j(\mathbb{M})$, $j = 1, \ldots, N$ denoting eigenvalues of $\mathbb{M}$. Moreover, if $\rho(\mathbb{M})$ is close to 1 then the convergence is slow.

Here, we deal with the Jacobi method. Thus, we write

$$\mathbb{A}_h = \mathbb{D}_h - \mathbb{B}_h, \tag{15.12}$$

where $\mathbb{D}_h = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{NN})$ is the diagonal matrix and $\mathbb{B}_h := \mathbb{D}_h - \mathbb{A}_h$. Obviously,

$$\mathbb{D}_h = \frac{2}{h^2}\mathbb{I} \qquad \Longrightarrow \qquad \mathbb{D}_h^{-1} = \frac{h^2}{2}\mathbb{I} \tag{15.13}$$

The the *Jacobi method* reads

$$\boldsymbol{u}_h^{j+1} := \mathbb{D}_h^{-1}(\mathbb{B}_h \boldsymbol{u}_h^j + \boldsymbol{f}_h), \quad j = 0, 1, \ldots, \tag{15.14}$$

and $\boldsymbol{u}_h^0 \in \mathbb{R}^N$ is an initial approximation. The iterative method (15.14) can be written as

$$\boldsymbol{u}_h^{j+1} := \boldsymbol{u}_h^j - \mathbb{D}_h^{-1}(\mathbb{A}_h \boldsymbol{u}_h^j - \boldsymbol{f}_h), \quad j = 0, 1, \ldots. \tag{15.15}$$

The quantity $\boldsymbol{d}_h^j := \mathbb{A}_h \boldsymbol{u}_h^j - \boldsymbol{f}_h$ is called the *defect*.

Let us consider the *damped Jacobi method*, which is more interesting for our purposes:

$$\boldsymbol{u}_h^{j+1} := \boldsymbol{u}_h^j - \mathbb{D}_h^{-1}\vartheta(\mathbb{A}_h \boldsymbol{u}_h^j - \boldsymbol{f}_h), \quad j = 0, 1, \ldots, \tag{15.16}$$

where $\vartheta \in (0, 1]$. Due to (15.13), putting $\omega = \vartheta/2 \in (0, \frac{1}{2}]$, we have re-write (15.16) as

$$\boldsymbol{u}_h^{j+1} := \boldsymbol{u}_h^j - \omega h^2(\mathbb{A}_h \boldsymbol{u}_h^j - \boldsymbol{f}_h), \quad j = 0, 1, \ldots. \tag{15.17}$$

Consequently, the iteration matrix $\mathbb{M}$ from (15.16) has the form

$$\mathbb{M}_h := \mathbb{I} - \omega h^2 \mathbb{A}_h. \tag{15.18}$$

In order to proceed with the analysis we introduce the eigenvalues and eigenvectors of $\mathbb{M}_h$. We start with the matrix $\mathbb{A}_h$. It can be verified that

$$\frac{4}{h^2}\sin^2\left(\frac{k\pi h}{2}\right), \qquad k = 1, \ldots, N \tag{15.19}$$

are the eigenvalues of $\mathbb{A}$ and the corresponding eigenvectors are

$$\boldsymbol{v}_h^k = \left\{\sqrt{2h}\sin(lk\pi h)\right\}_{l=1}^N, \qquad k = 1, \ldots, N. \tag{15.20}$$

It follows from (15.18) and (15.19) that the eigenvalues of $\mathbb{M}_h$ are

$$\lambda_k(\omega) = 1 - 4\omega\sin^2\left(\frac{k\pi h}{2}\right), \qquad k = 1, \ldots, N. \tag{15.21}$$

Figure 15.1 shows the eigenvalues for $N = 20$ for the Jacobi method with $\omega = \frac{1}{2}$ (red) and the damped Jacobi method with $\omega = \frac{1}{4}$ (blue).

We simply observe that the maximal eigenvalue is for $k = 1$, i.e., the spectral radius of $\mathbb{M}_h$ is

$$\rho(\mathbb{M}_h) = 1 - 4\omega\sin^2\left(\frac{\pi h}{2}\right) = 1 - \omega\pi^2 h^2 + O(h^4). \tag{15.22}$$

It means that for the Jacobi method ($\omega = \frac{1}{2}$), the convergence is decreasing for $h \to 0$. Moreover, for any other $\omega \in (0, \frac{1}{2})$ the convergence is still slower.

Finally, let us note that the eigenvectors $\boldsymbol{v}_h^k$, $k = 1, \ldots, N$ are also the eigenvectors of $\mathbb{M}_h$, i.e.,

$$\mathbb{M}_h \boldsymbol{v}_h^k = \lambda_k(\omega)\boldsymbol{v}_h^k, \qquad k = 1, \ldots, N. \tag{15.23}$$

Figure 15.2 shows examples of the eigenvectors. For increasing k, the entries of $\boldsymbol{v}_h^k$ are more and more oscillating.

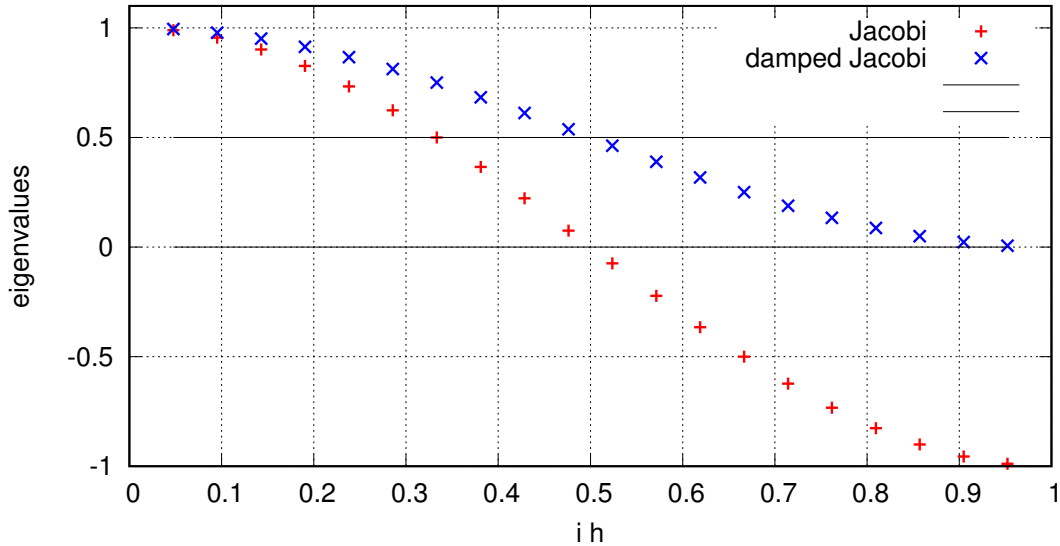


Figure 15.1: Eigenvalues of the matrix $\mathbb{M}_h$ for $\omega = \frac{1}{2}$ (red) and $\omega = \frac{1}{4}$ (blue).

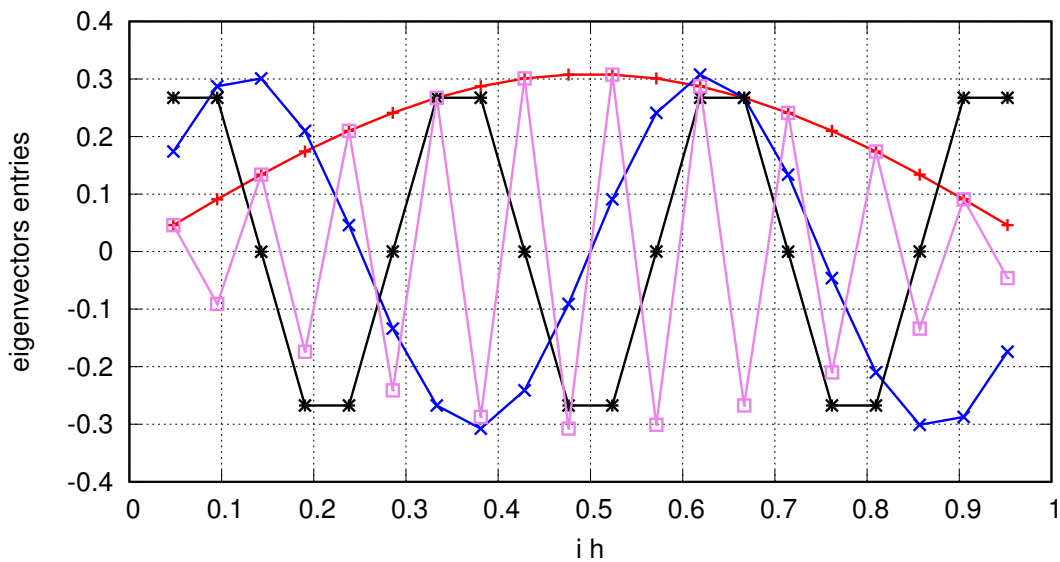


Figure 15.2: Eigenvectors $\boldsymbol{v}_h^1$ (red), $\boldsymbol{v}_h^4$ (blue), $\boldsymbol{v}_h^7$ (black) and $\boldsymbol{v}_h^{20}$ (violet) of the matrix $\mathbb{M}_h$

15.3 Smoothing effect of the damped Jacobi iterative methods

Although the Jacobi method is very inefficient for small h (large N) it has to so-called *smoothing property*. In order demonstrate it we fix $\omega = \frac{1}{4}$ in the following, hence

$$\mathbb{M}_h := \mathbb{I} - \frac{h^2}{4}\mathbb{A}_h. \tag{15.24}$$

Moreover, we put $\lambda_k := \lambda_k(\frac{1}{4}),\ k = 1, \ldots, N$.

From (15.17), we have

$$\boldsymbol{u}_h^{j+1} := \mathbb{M}_h \boldsymbol{u}_h^j + \frac{h^2}{4}\boldsymbol{f}_h, \quad j = 0, 1, \ldots. \tag{15.25}$$

Let $\boldsymbol{u}_h$ be the exact solution of (15.8) then it is valid

$$\boldsymbol{u}_h = \mathbb{M}_h \boldsymbol{u}_h + \frac{h^2}{4}\boldsymbol{f}_h. \tag{15.26}$$

We define the error of the j-th approximation by

$$\boldsymbol{e}_h^j := \boldsymbol{u}_h^j - \boldsymbol{u}_h, \qquad j = 0, 1, \ldots. \tag{15.27}$$

Using (15.25)–(15.26), we derive

$$\boldsymbol{e}_h^{j+1} = \boldsymbol{u}_h^{j+1} - \boldsymbol{u}_h = \left(\mathbb{M}_h \boldsymbol{u}_h^j + \frac{h^2}{4}\boldsymbol{f}_h\right) - \left(\mathbb{M}_h \boldsymbol{u}_h + \frac{h^2}{4}\boldsymbol{f}_h\right) = \mathbb{M}_h\left(\boldsymbol{u}_h^j - \boldsymbol{u}_h\right) = \mathbb{M}_h \boldsymbol{e}_h^j.$$

Using the induction, we derive

$$\boldsymbol{e}_h^j = \mathbb{M}_h^j \boldsymbol{e}_h^0, \qquad j = 0, 1, \ldots, \tag{15.28}$$

where $\mathbb{M}_h^j = \overbrace{\mathbb{M}_h\, \mathbb{M}_h\, \ldots\, \mathbb{M}_h}^{j \text{ times}}$ and $\boldsymbol{e}_h^0 \in \mathbb{R}^N$ is the initial error given by the initial approximation $\boldsymbol{u}_h^0 \in \mathbb{R}^N$.

Since the eigenvectors $\boldsymbol{v}_h^k,\ k = 1, \ldots, N$ (cf. (15.20)) form a basis of $\mathbb{R}^N$, we can write

$$\boldsymbol{e}_h^0 = \sum_{k=1}^{N} \alpha_k \boldsymbol{v}_h^k, \tag{15.29}$$

where $\alpha_k \in \mathbb{R},\ k = 1, \ldots, N$ are the coefficients.

In virtue of (15.23), (15.28) and (15.29), we derive

$$\boldsymbol{e}_h^j = \mathbb{M}_h^j \boldsymbol{e}_h^0 = \mathbb{M}_h^j \left(\sum_{k=1}^{N} \alpha_k \boldsymbol{v}_h^k\right) = \sum_{k=1}^{N} \alpha_k \mathbb{M}_h^j \boldsymbol{v}_h^k = \sum_{k=1}^{N} \alpha_k \lambda_k^j \boldsymbol{v}_h^k.$$

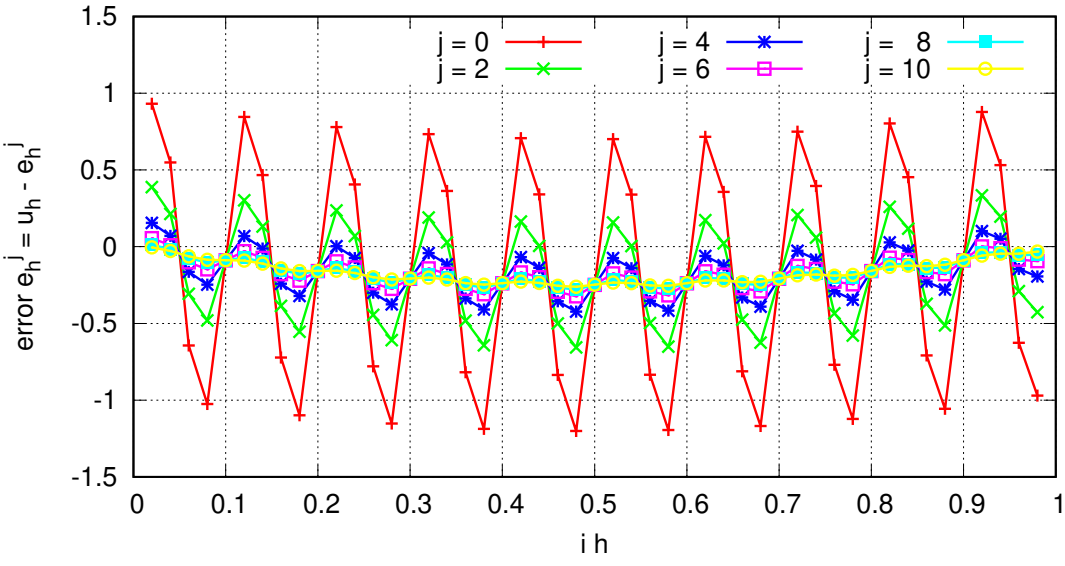


Figure 15.3: Demonstration of the smoothing property of the damped Jacobi method.

Obviously, due to (15.21), $|\lambda_k| < 1,\ k = 1, \ldots, N$ and then the sum converges (at least slowly) to 0. However, from Figure 15.1, we observe that

$$|\lambda_k| \leq \frac{1}{2} \qquad \text{for } k \geq N/2. \tag{15.30}$$

Therefore, the entries $\alpha_k,\ k \geq N/2$ of the initial error $\boldsymbol{e}_h^0$, corresponding to "high frequencies" (represented by the eigenvectors $\boldsymbol{v}_h^k$ – see Figure 15.2), are reduced at least by factor $\frac{1}{2}$ at each Jacobi iteration.

Therefore, we can expect that the high frequencies of the initial error will be eliminated after a small number of iteration.

Example 15.1. *Let us consider problem* (15.1) *with* $N = 50$, $f = 2$. *The exact solution is* $u(x) = x(1-x)$. *Let the initial approximation is chosen as a fast oscillation vector*

$$\boldsymbol{u}_h^0 := \left\{ \sin\left(\frac{20\pi i}{N+1}\right) \right\}_{i=1}^{N}. \tag{15.31}$$

The smoothing effect is demonstrated in Figure 15.3. However, the convergence is very slow, see Figure 15.4.

15.4 Basic idea of the multigrid method

In previous section, we observed that the high-frequencies components of the initial error are eliminated quickly, but the convergence is slow. However, when the error is smooth

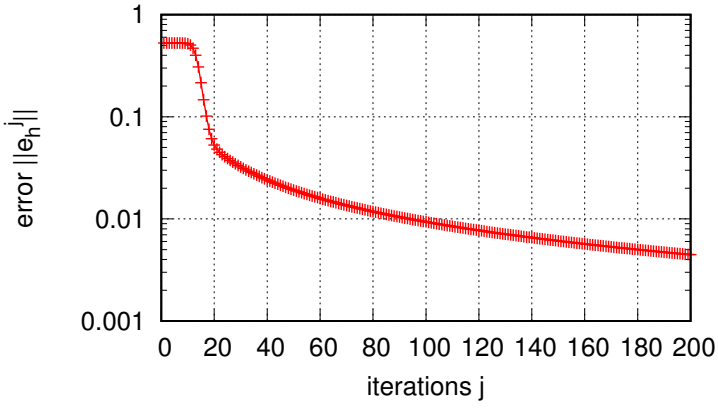


Figure 15.4: Convergence of the damped Jacobi method.

(without oscillations) it is sufficient to approximate it on a coarser grid, let us say with the step $2h$. Then the low-frequencies on the grid with step h becomes faster on the grid with step $2h$ and they could be eliminated faster.

Let us assume that $\boldsymbol{u}_h^j$ is the approximate solution (15.8) such that the corresponding error

$$\boldsymbol{e}_h^j = \boldsymbol{u}_h^j - \boldsymbol{u}_h \tag{15.32}$$

is already smooth. Obviously,

$$\mathbb{A}_h \boldsymbol{e}_h^j = \boldsymbol{d}_h^j, \qquad \text{where } \boldsymbol{d}_h^j = \mathbb{A}_h \boldsymbol{u}_h^j - \boldsymbol{f}_h. \tag{15.33}$$

If we are able to solve (15.33) and to obtain a sufficiently accurate approximation of $\boldsymbol{e}_h^j$ denoted by $\tilde{\boldsymbol{e}}_h^j$, then the term $\boldsymbol{u}_h^j - \tilde{\boldsymbol{e}}_h^j$ exhibits a sufficiently accurate approximation of $\boldsymbol{u}_h$. However, the solution of (15.33) by the damped Jacobi method is inefficient as the solution (15.8). But since $\boldsymbol{e}_h^j$ is already smooth, we can solve (15.33) on a coarser grid and the result interpolate back to the actual grid. More precisely, the algebraic problem (15.33) corresponds to the FD discretization of a problem like (15.1). Hence, we consider this boundary-value problem and discretize it by FD with the step $2h$.

Let N be odd for simplicity and we consider the discretization of (15.1) using FD scheme with step $2h = \frac{2}{N+1}$. Let $\mathbb{A}_{2h}$ be the corresponding FD matrix of the size $\frac{N-1}{2} \times \frac{N-1}{2}$ given by

$$\mathbb{A}_{2h} = \frac{1}{4h^2} \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}. \tag{15.34}$$

Further, we have to introduce an operator which recomputes the defect $\boldsymbol{d}_h = (d_{h,1}, d_{h,2}, \ldots, d_{h,N})$ to the defect $\boldsymbol{d}_{2h} = (d_{2h,1}, d_{2h,2}, \ldots, d_{2h,(N-1)/2})$ on the coarser grid. This operator is called the *restriction* and it maps vectors from $\mathbb{R}^N \to \mathbb{R}^{(N-1)/2}$. One possibility is to set

$$d_{2h,i} := d_{h,2i}, \qquad i = 1, 2, \ldots, (N-1)/2. \tag{15.35}$$

However, this choice is not too much efficient since it ignores info from the odd entries of $\boldsymbol{d}_h$. The better is

$$d_{2h,i} := \frac{1}{4}\left(d_{h,2i-1} + d_{h,2i} + d_{h,2i+1}\right), \qquad i = 1, 2, \ldots, (N-1)/2. \tag{15.36}$$

The restriction (15.36) can be written in the matrix form

$$\boldsymbol{d}_{2h} = \mathbb{R}^h_{2h}\boldsymbol{d}_h, \tag{15.37}$$

where $\mathbb{R}^h_{2h} \in R^{(N-1)/2\times N}$ is the *restriction operator* (matrix) having the form

$$\mathbb{R}^h_{2h} = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & & & & \\ & & 1 & 2 & 1 & & & & & \\ & & & & 1 & 2 & 1 & & & \\ & & & & & & \ddots & \ddots & \ddots & \\ & & & & & & & 1 & 2 & 1 \end{pmatrix}. \tag{15.38}$$

Now, we restrict the problem (15.33) to the grid with step $2h$ which means that we seek vector $\boldsymbol{e}_{2h} \in \mathbb{R}^{(N-1)/2}$ such that

$$\mathbb{A}_{2h}\boldsymbol{e}_{2h} = \boldsymbol{d}_{2h}, \tag{15.39}$$

where $\mathbb{A}_{2h}$ is the matrix (15.34) and $\boldsymbol{d}_{2h}$ is the restriction of the defect $\boldsymbol{d}^j_h$ given by $\boldsymbol{d}_{2h} = \mathbb{R}^h_{2h}\boldsymbol{d}^j_h$.

Further, we solve (iteratively) the problem (15.39) and obtain an approximation $\tilde{\boldsymbol{e}}_{2h} = (\tilde{e}_{2h,1}, \ldots, \tilde{e}_{2h,(N-1)/2}) \in \mathbb{R}^{(N-1)/2}$. This vector has to be *prolongated* back to the grid with step h resulting the vector $\tilde{\boldsymbol{e}}_h = (\tilde{e}_{h,1}, \ldots, \tilde{e}_{h,N})$, e.g., as

$$\tilde{e}_{h,i} = \begin{cases} \tilde{e}_{2h,i/2} & \text{for } i \text{ even}, \\ \frac{1}{2}\left(\tilde{e}_{2h,(i-1)/2} + \tilde{e}_{2h,(i+1)/2}\right) & \text{for } i \text{ odd}. \end{cases} \tag{15.40}$$

This is equivalent with the matrix form

$$\tilde{\boldsymbol{e}}_h = \mathbb{P}^{2h}_h\tilde{\boldsymbol{e}}_{2h}, \tag{15.41}$$

where $\mathbb{P}_h^{2h} \in R^{N\times(N-1)/2}$ is the *prolongation operator* (matrix) having the form

$$\mathbb{P}_h^{2h} = \frac{1}{2}\begin{pmatrix} 1 & & & & & & & \\ 2 & & & & & & & \\ 1 & 1 & & & & & & \\ & 2 & & & & & & \\ & 1 & 1 & & & & & \\ & & 2 & & & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & & 2 & 0 \\ & & & & & 1 & 1 \\ & & & & & & 2 \\ & & & & & & 1 \end{pmatrix}. \tag{15.42}$$

Having the vector $\tilde{\boldsymbol{e}}_h$ from (15.41), in virtue of (15.32), we set

$$\tilde{\boldsymbol{u}}_h^j := \boldsymbol{u}_h^j - \tilde{\boldsymbol{e}}_h \tag{15.43}$$

as a better approximation of $\boldsymbol{u}_h$ than $\boldsymbol{u}_h^j$. The approximation $\tilde{\boldsymbol{u}}_h^j$ contains the errors of the high-frequencies (but not to low-frequencies which are assumed to be eliminated on the coarse grid with step $2h$) and thus some additional smoothing is necessary.

15.5 Two-grid algorithm

We summarize the previous determination. By symbol $\mathcal{S}^\nu$ we formally denote the smoothing by ν iterations (using damped Jacobi method), i.e., (15.25) can be written as

$$\boldsymbol{u}_h^j = \mathcal{S}^j \boldsymbol{u}_h^0, \qquad j = 0, 1, \dots. \tag{15.44}$$

The following algorithm generates a sequence of approximation $\boldsymbol{u}_h^{(k)}$ where each term is a result of smoothing and coarse grid correction which exhibits several damped Jacobi iterations. (Hence, the meaning of the symbol $\boldsymbol{u}_h^{(k)}$ is different from $\boldsymbol{u}_h^j$.)

Algorithm 15.2. Two-grid algorithm

1. *let* $\nu_1 > 0$ *and* $\nu_2 > 0$ *be integers,*
2. *let* $\boldsymbol{u}_h^{(0)}$ *be the given initial approximation,*
3. *put* $k := 0, 1, 2, \dots$
   (a) *pre-smoothing* $\tilde{\boldsymbol{u}}_h := \mathcal{S}^{\nu_1}\boldsymbol{u}_h^{(k)}$,
   (b) *defect* $\boldsymbol{d}_h := \mathbb{A}_h\tilde{\boldsymbol{u}}_h - \boldsymbol{f}_h$,
   (c) *restriction* $\boldsymbol{d}_{2h} := \mathbb{R}_{2h}^h \boldsymbol{d}_h$

(d) *solution of the coarse grid :* $\boldsymbol{e}_{2h} := (\mathbb{A}_{2h})^{-1}\boldsymbol{d}_{2h}$, *usually done by an iterative method, hence* $\tilde{\boldsymbol{e}}_{2h} \approx \boldsymbol{e}_{2h}$ *is available only,*

(e) *prolongation of the error:* $\tilde{\boldsymbol{e}}_h := \mathbb{P}_h^{2h}\tilde{\boldsymbol{e}}_{2h}$

(f) *solution correction:* $\bar{\boldsymbol{u}}_h := \tilde{\boldsymbol{u}}_h - \tilde{\boldsymbol{e}}_h$,

(g) *post-smoothing* $\boldsymbol{u}_h^{(k+1)} := \mathcal{S}^{\nu_2}\bar{\boldsymbol{u}}_h$,

(h) *if the prescribed accuracy achieved then STOP.*

The computational performance of two-grid method can be demonstrated by the following example.

Example 15.3. *Similarly as in Example 15.1, let us consider problem (15.1) with* $N = 50$, $f = 2$. *The exact solution is* $u(x) = x(1-x)$. *Let the initial approximation is chosen as a fast oscillation vector as in (15.31). The observations are the following:*

- *Figure 15.5, (a): damped Jacobi method after 1000, 2000, . . . , 6000 iterations,*
- *Figure 15.5, (b): two-grid method after 1, 2, . . . , 6 cycles, smoothing with* $\nu_2 = 10$ *with damped Jacobi iterations, problem on the coarse grid solved by the damped Jacobi method with 1000 iterations*
- *Figure 15.5, (c): two-grid method after 1, 2, . . . , 6 cycles, smoothing with* $\nu_2 = 10$ *with damped Jacobi iterations, problem on the coarse grid solved directly.*

The efficiency of MG method is evident

Moreover, Figure 15.6 shows the same cases for the initial approximation $\boldsymbol{u}_h^0 = 0$ *which is more natural. This initial approximation does not contain the oscillations. Thus the convergence is slightly faster but the differences are evidently the same.*

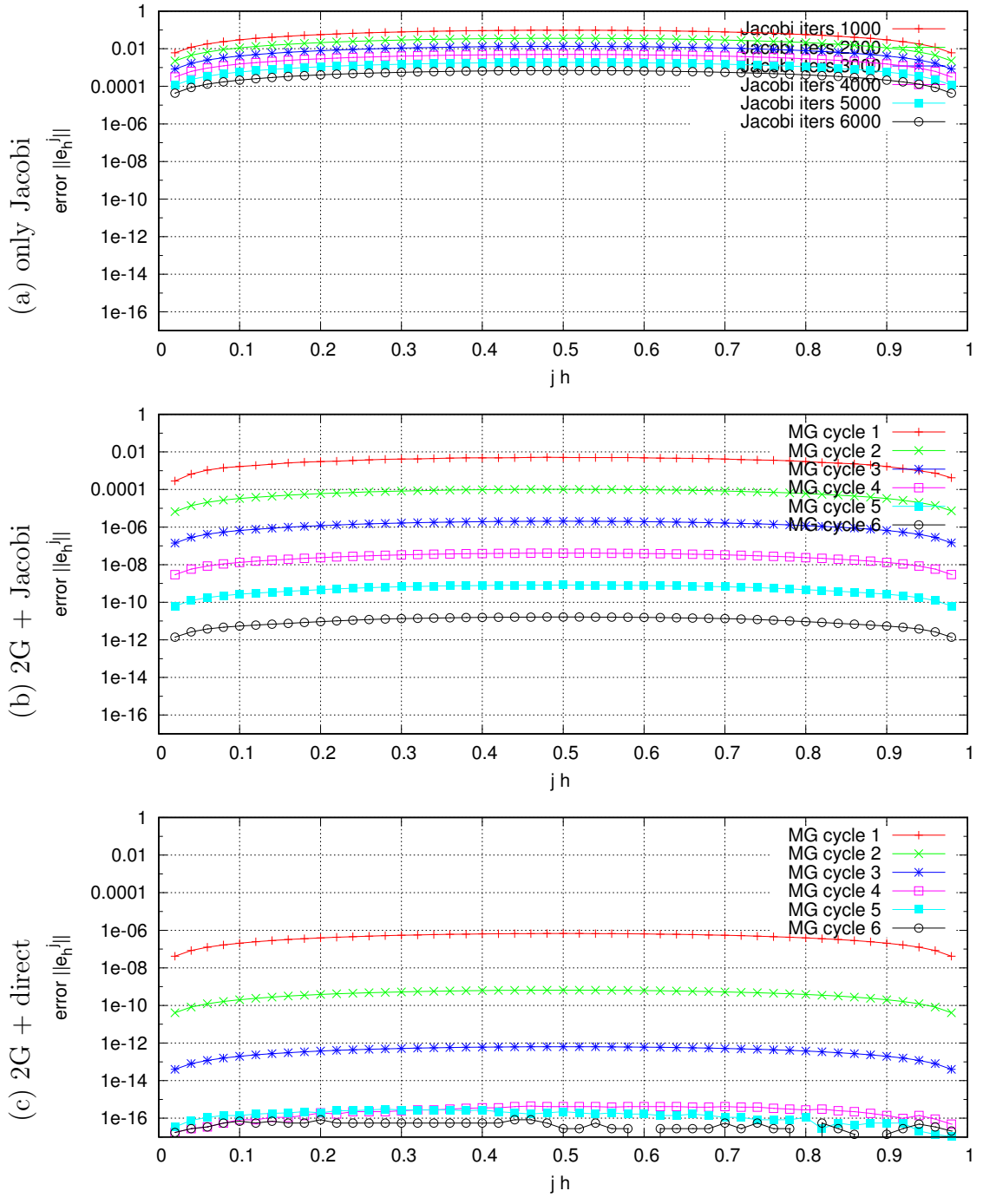


Figure 15.5: Comparison of the distribution of the error of damped Jacobi and MG method with $\boldsymbol{u}_h^0$ given by (15.31).

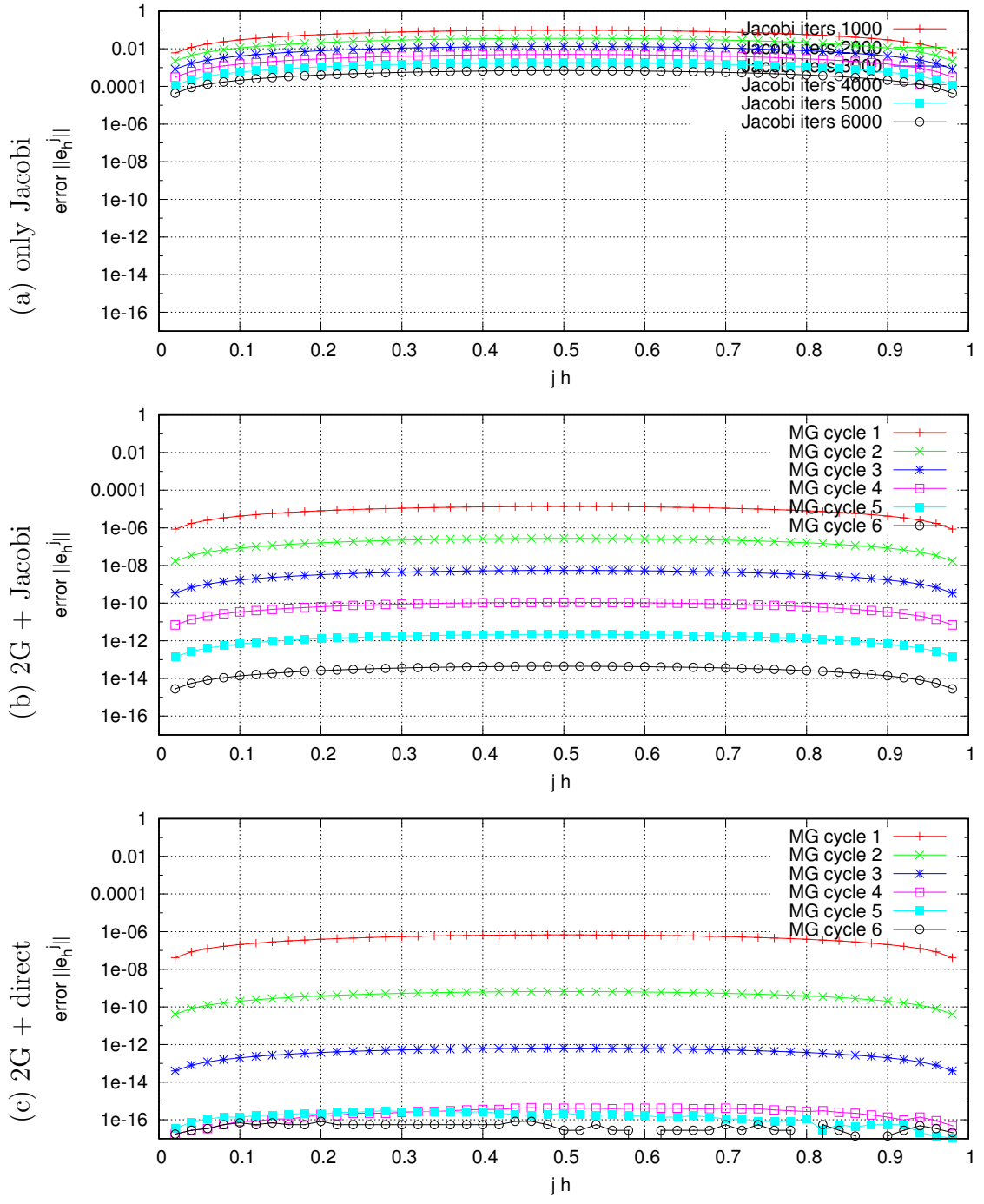


Figure 15.6: Comparison of the distribution of the error of damped Jacobi and MG method with $\boldsymbol{u}_h^0 = 0$..

15.6 Additional comments

15.6.1 Theoretical results

For numerical analysis, see, e.g., [Hac85]. General results:

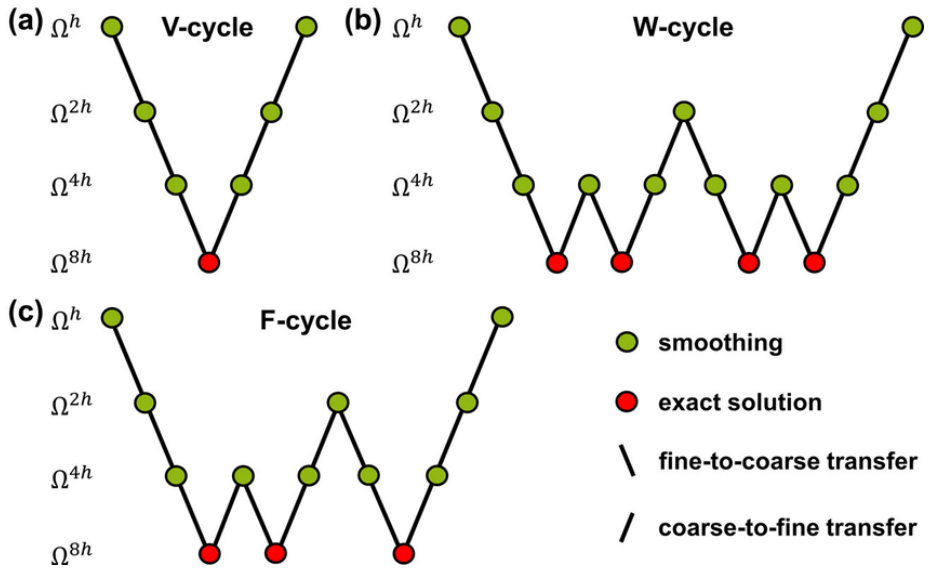
$$\|\text{error after } n \text{ steps}\| \leq \theta \|\text{error after 1 step}\|, \tag{15.45}$$

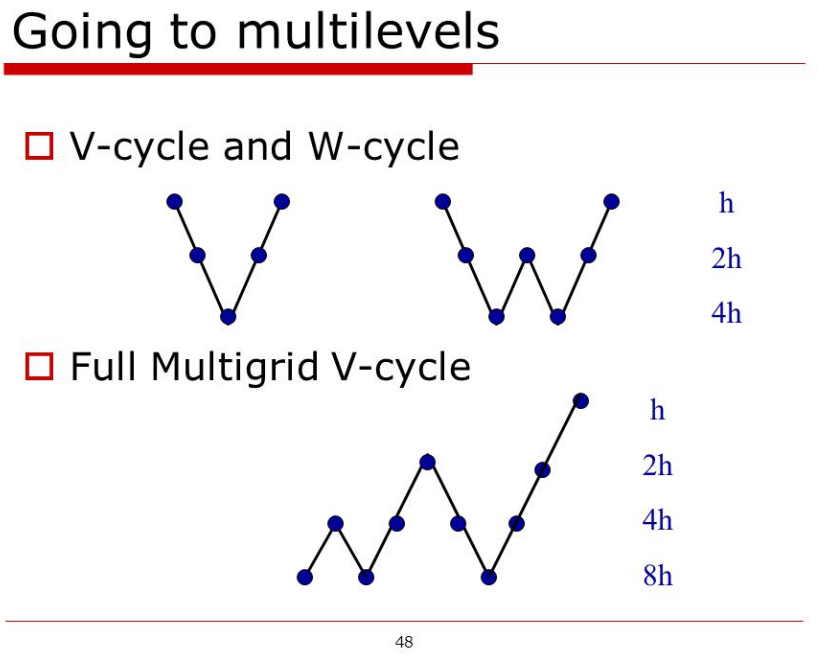
where $\theta < 1$ is independent of h. Typically $\theta = \frac{1}{10}$. Number of operations (in ideal situation) $O(N)$!

However, in practice, the prescribed tolerance depends on the discretization error, i.e, $O(h^2)$. It is possible to use *full multigrid* (FMG) method.

15.6.2 Multilevel techniques

The problem on the coarse grid (15.39) can be solved again by the same approach. This means that we carry out few smoothing iterations, evaluate the defect and restrict it to grid with the size $4h$. There solve the problem and prolongate back again. There exist several variants (figure taken from [JSdlKdRB16]):





[1]

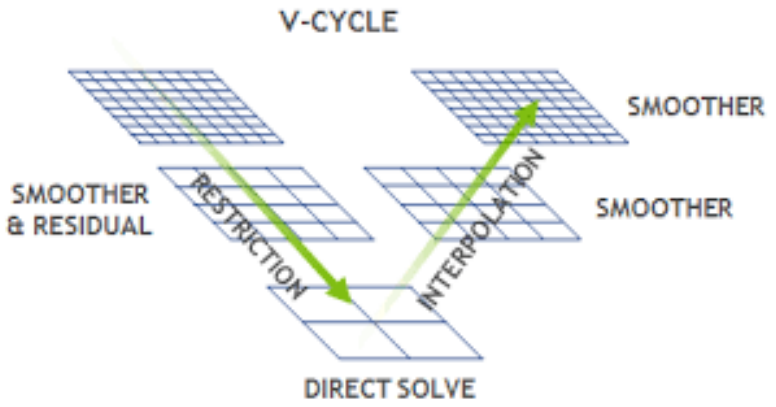
Analysis shows that time is well spent on the coarse grids. So the W-cycle that stays coarse longer is generally superior to a V-cycle. Often, the rate of the convergence of W-cycle is better but it requires more computational work.

The *full multigrid cycle* is asymptotically better than V or W. Full multigrid starts on the coarsest grid. The solution on the $8h$ grid is interpolated to provide a good initial vector $\boldsymbol{u}_{4h}$ on the $4h$ grid. A V-cycle between $4h$ and $8h$ improves it. Then interpolation predicts the solution on the $2h$ grid, and a deeper V-cycle makes it better (using $2h$, $4h$, $8h$). Interpolation of that improved solution onto the finest grid gives an excellent start to the last and deepest V-cycle.

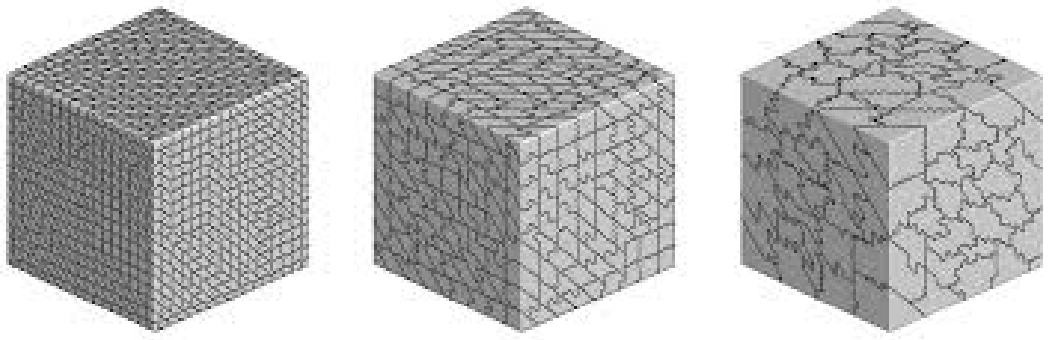
15.6.3 Multigrid methods for 2D and 3D

Can be generalized to 2D and 3D problems, more technically and (mostly) practically complicated.

[1] https://slideplayer.com/slide/3187496/



[2]



[3]

15.6.4 Algebraic multigrid methods

Similar techniques but only on the algebraic level, without a direct connection to the discretization of PDE. It seeks the "low" and "high" frequencies in the matrix.

15.6.5 p-variant of the multigrid methods

For higher-order methods, but the coarser grids are replaced by lower polynomial approximation degrees.

15.6.6 Nonlinear multigrid methods

Generalization of the MG idea to nonlinear problems.

[2] https://devblogs.nvidia.com/high-performance-geometric-multigrid-gpu-acceleration/
[3] https://fun3d.larc.nasa.gov/papers/AIAA-2009-4138.pdf

Homeworks

Exercise 22. *Using the code* `MG.f90`*, reproduce the smoothing property from Example 15.1. What does happen if we put* $\boldsymbol{u}_h^0 = 0$ *instead of* (15.31)*?*

Exercise 23. *Using the code* `MG.f90`*, compare MG method with the direct solver and the iterative solver on the coarse grid level.*

Exercise 24. *Using the code* `MG.f90`*, try the restriction operator given by* (15.35)*.*

Exercise 25. *Modify the code and test it for the Jacobi method without the damping.*

Exercise 26. *Modify the code and test it for the Gauss-Seidel method.*

Exercise* 27. *Write own code or modify the code* `MG.f90` *for MG method using more than two-grids.*

Chapter 16

UMFPACK

UMFPACK is a set of routines for solving unsymmetric sparse linear systems, $Ax = b$, using the Unsymmetric-pattern MultiFrontal method and direct sparse LU factorization. It is written in ANSI/ISO C, with a MATLAB interface. UMFPACK relies on the Level-3 Basic Linear Algebra Subprograms (BLAS) (dense matrix multiply) for its performance. This code works on Windows and many versions of Unix (Sun Solaris, Red Hat Linux, IBM AIX, SGI IRIX, and Compaq Alpha).

You will need to install AMD library to use UMFPACK. The UMFPACK and AMD subdirectories must be placed side-by-side within the same parent directory. AMD is a stand-alone package that is required by UMFPACK. UMFPACK can be compiled without the BLAS but your performance will be much less than what it should be.

Installation is a little more complicated, see the manual.

16.1 Instalation of UMFPACK: link of Fortran and C++ languages

Chapter 17

Software for visualization

17.1 Gnuplot

17.2 Paraview

Chapter 18

Notes on a posteriori error estimates

18.1 Residual error estimates

Let $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ be a bounded polygonal computational domain with boundary $\Gamma = \Gamma_D \cup \Gamma_N$ where Γ_D and Γ_N are disjoint and $\Gamma_D \neq \emptyset$. We consider the Laplace equation: find $u : \Omega \to \mathbb{R}$

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_D \quad \text{on } \Gamma_D, \\ \nabla u \cdot \boldsymbol{n} &= g_N \quad \text{on } \Gamma_N, \end{aligned} \tag{18.1}$$

where f, u_D and g_N are given functions.

Let $H^1_D(\Omega) := \{v \in H^1(\Omega) : \ v|_{\Gamma_D} = 0\}$ and $u^* \in H^1(\Omega)$ be a function such that $u^*|_{\Gamma_D} = u_D$. The "standard" weak formulation of (18.1) reads: find $u \in H^1(\Omega)$ such that

$$\text{(i)} \quad u - u^* \in H^1_D(\Omega), \tag{18.2}$$

$$\text{(ii)} \quad \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x + \int_{\Gamma_N} g_N v \,\mathrm{d}S \quad \forall v \in H^1_D(\Omega). \tag{18.3}$$

For simplicity, we assume that $u_D = 0$, otherwise we consider $u' := u - u^*$.

Let $\mathscr{T}_h$ be a simplicial *shape-regular* mesh of Ω whose elements are denotes by $K \in \mathscr{T}_h$. Let $W_h \subset H^1_D(\Omega)$ be a finite dimensional space of continuous piecewise polynomial functions over $\mathscr{T}_h$. Then the approximate solution of (18.2) – (18.3) reads: find $u_h \in W_h$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v_h \,\mathrm{d}x = \int_\Omega f v_h \,\mathrm{d}x + \int_{\Gamma_N} g_N v_h \,\mathrm{d}S \quad \forall v_h \in W_h. \tag{18.4}$$

Let $u \in H^1_D(\Omega)$ and $u_h \in W_h$ be the weak and approximate solutions, respectively. From (18.3), we obtain the identity

$$\int_\Omega \nabla(u - u_h) \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x + \int_{\Gamma_N} g_N v \,\mathrm{d}S - \int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x =: R_h[u_h](v), \tag{18.5}$$

$$v \in H^1_D(\Omega),$$

where $R_h[u_h](\cdot)$ denotes the residual of the approximate problem (18.4). Obviously,

$$R_h[u_h](v_h) = 0 \qquad v_h \in W_h, \tag{18.6}$$

which is the Galerkin orthogonality of the error.

We recall the identity following from the Cauchy-Schwartz inequality:

$$\|\nabla v\|_\Omega = \sup_{w\in H^1_D(\Omega), w\neq 0} \frac{1}{\|\nabla w\|_\Omega} \int_\Omega \nabla v \cdot \nabla w \,\mathrm{d}x \qquad \forall v \in H^1_D(\Omega), \tag{18.7}$$

where $\|\cdot\|_\Omega = \|\cdot\|_{L^2(\Omega)}$. We note that (18.7) defines the seminorm which is an equivalent norm on $H^1_D(\Omega)$.

Therefore, relations (18.5)–(18.7) implies

$$\|\nabla(u - u_h)\|_\Omega = \sup_{v\in H^1_D(\Omega), v\neq 0} \frac{R_h[u_h](v)}{\|\nabla v\|_\Omega}. \tag{18.8}$$

We are going to manipulate with (18.5). Let $\mathscr{F}_h^{\mathrm{I}}$ denote a set of all interior edges of the mesh $\mathscr{T}_h$, $\mathscr{F}_h^{\mathrm{D}}$ the set of all edges lying on Γ_D, $\mathscr{F}_h^{\mathrm{N}}$ the set of all edges lying on Γ_N and $\mathscr{F}_h = \mathscr{F}_h^{\mathrm{I}} \cup \mathscr{F}_h^{\mathrm{D}} \cup \mathscr{F}_h^{\mathrm{N}}$. The symbol γ denotes a generic edge from $\mathscr{F}_h$. Using the Green theorem, we have

$$\begin{aligned} R_h[u_h](v) &= \int_\Omega fv \,\mathrm{d}x + \int_{\Gamma_N} g_N v \,\mathrm{d}S - \sum_{K\in\mathscr{T}_h} \int_K \nabla u_h \cdot \nabla v \,\mathrm{d}x \\ &= \int_\Omega fv \,\mathrm{d}x + \int_{\Gamma_N} g_N v \,\mathrm{d}S + \sum_{K\in\mathscr{T}_h} \int_K \Delta u_h \cdot v \,\mathrm{d}x - \sum_{K\in\mathscr{T}_h} \int_{\partial K} \nabla u_h \cdot \boldsymbol{n} v \,\mathrm{d}S \\ &= \sum_{K\in\mathscr{T}_h} \int_K (f - \Delta u_h) v \,\mathrm{d}x + \sum_{\gamma\in\mathscr{F}_h^{\mathrm{N}}} \int_\gamma (g_N - \nabla u_h \cdot \boldsymbol{n}) v \,\mathrm{d}S - \sum_{\gamma\in\mathscr{F}_h^{\mathrm{I}}} \int_\gamma [\![\nabla u_h \cdot \boldsymbol{n}]\!] v \,\mathrm{d}S, \end{aligned} \tag{18.9}$$

where $[\![\cdot]\!]$ denotes a jump of the argument on the edge $\gamma \in \mathscr{F}_h^{\mathrm{I}}$ in the direction of unit normal $\boldsymbol{n}$. Denoting

$$\begin{aligned} R_K(u_h) &:= f - \Delta u_h, \quad K \in \mathscr{T}_h, \\ R_\Gamma(u_h) &:= \begin{cases} 0, & \gamma \in \mathscr{F}_h^{\mathrm{D}}, \\ g_N - \nabla u_h \cdot \boldsymbol{n}, & \gamma \in \mathscr{F}_h^{\mathrm{N}}, \\ -[\![\nabla u_h \cdot \boldsymbol{n}]\!], & \gamma \in \mathscr{F}_h^{\mathrm{I}}, \end{cases} \end{aligned} \tag{18.10}$$

we rewrite (18.9) as

$$R_h[u_h](v) = \sum_{K\in\mathscr{T}_h} \int_K R_K(u_h) v \,\mathrm{d}x + \sum_{\gamma\in\mathscr{F}_h} \int_\gamma R_\Gamma(u_h) v \,\mathrm{d}S. \tag{18.11}$$

Let $\Pi w \in W_h$ be a continuous, piecewise linear Scott-Zhang interpolation of $w \in H^1_D(\Omega)$, [EG04, Section 1.6.2], [Ver13, Section 1.3.3 or Section 3.5] or the original paper [SZ90]. Then there exists $c > 0$ such that

$$\begin{aligned} \|w - \Pi w\|_K &\le c h_K \|\nabla w\|_{\omega_K} \qquad \forall w \in H^1_D(\Omega), \\ \|w - \Pi w\|_\gamma &\le c h_\gamma^{1/2} \|\nabla w\|_{\omega_\gamma} \qquad \forall w \in H^1_D(\Omega), \end{aligned} \tag{18.12}$$

where ω_K denotes a patch of elements sharing at least a vertex with $K \in \mathscr{T}_h$ and ω_γ denotes a patch of elements sharing at least a vertex with $\gamma \in \mathscr{F}_h$; h_K and h_γ denote the diameters of $K \in \mathscr{T}_h$ and $\gamma \in \mathscr{F}_h$.

Using (18.6), we have the identity

$$R_h[u_h](v) = R_h[u_h](v - \Pi v), \qquad v \in H^1_D(\Omega). \tag{18.13}$$

Applying the Cauchy-Schwartz inequality, we obtain from (18.11) – (18.13) the estimate

$$\begin{aligned} &R_h[u_h](v) = R_h[u_h](v - \Pi v) \qquad (18.14)\\ &\le \sum_{K \in \mathscr{T}_h} \|R_K(u_h)\|_K \|v - \Pi v\|_K + \sum_{\gamma \in \mathscr{F}_h} \|R_\Gamma(u_h)\|_\gamma \|v - \Pi v\|_\gamma \\ &\le c \sum_{K \in \mathscr{T}_h} h_K \|R_K(u_h)\|_K \|\nabla v\|_{\omega_K} + c \sum_{\gamma \in \mathscr{F}_h} h_\gamma^{1/2} \|R_\Gamma(u_h)\|_\gamma \|\nabla v\|_{\omega_\gamma} \\ &\le c \left(\sum_{K \in \mathscr{T}_h} h_K^2 \|R_K(u_h)\|_K^2 + \sum_{\gamma \in \mathscr{F}_h} h_\gamma \|R_\Gamma(u_h)\|_\gamma^2 \right)^{1/2} \left(\sum_{K \in \mathscr{T}_h} \|\nabla v\|_{\omega_K}^2 + \sum_{\gamma \in \mathscr{F}_h} \|\nabla v\|_{\omega_\gamma}^2 \right)^{1/2} \\ &\le \tilde{c} \left(\sum_{K \in \mathscr{T}_h} h_K^2 \|R_K(u_h)\|_K^2 + \sum_{\gamma \in \mathscr{F}_h} h_\gamma \|R_\Gamma(u_h)\|_\gamma^2 \right)^{1/2} \|\nabla v\|_\Omega, \end{aligned}$$

where in the last step we use the shape-regularity of $\mathscr{T}_h$ implying the existence of a constant $c^* > 0$ such that

$$\sum_{K \in \mathscr{T}_h} \|\nabla v\|_{\omega_K}^2 + \sum_{\gamma \in \mathscr{F}_h} \|\nabla v\|_{\omega_\gamma}^2 \le c^* \|\nabla v\|_\Omega^2 \qquad \forall v \in H^1_D(\Omega). \tag{18.15}$$

Finally, (18.8) and (18.14) imply

$$\|\nabla(u - u_h)\|_\Omega \le \bar{c} \left(\sum_{K \in \mathscr{T}_h} h_K^2 \|R_K(u_h)\|_K^2 + \sum_{\gamma \in \mathscr{F}_h} h_\gamma \|R_\Gamma(u_h)\|_\gamma^2 \right)^{1/2}, \tag{18.16}$$

which is the classical residual based a posteriori error estimate.

18.2 Dual weighted residual error estimates

In many practical applications, we are not interested in the solution u of the given partial differential equations as such, but in the value of a certain *quantity of interest*, which depends on the solution. This quantity is given by a solution-dependent (target) functional denoted hereafter as $J(u)$. Therefore, the output of the numerical solution is the value $J(u_h)$ and the goal is the estimation of the error $J(u) - J(u_h)$. This lead to the framework of the goal-oriented error estimates, a more detailed explanation and analysis can be found, e.g., in [BR03, BR01, GS02, Har07].

18.2.1 Primal problem

Let Ω be a computational domain with boundary Γ. We consider a linear *variational problem*: find $u \in W$ such that

$$a(u, \varphi) = \ell(\varphi) \qquad \forall \varphi \in V, \tag{18.17}$$

where $u \in W$ is a weak solution, $a(\cdot, \cdot) : W \times V \to \mathbb{R}$ is a bilinear form, $\ell(\cdot) : V \to \mathbb{R}$ is a linear form, and W, V are Banach spaces. We assume that (18.17) is well-posed, i.e., it admits a unique weak solution,

In order to solve (18.17) numerically, we consider the finite element spaces W_h and V_h. We admit the nonconforming approximation, i.e., $W_h \not\subset W$ and $V_h \not\subset V$. Therefore, we define the spaces $W(h) = W + W_h$ and $V(h) = V + V_h$. If $W_h \subset W$, we put $W(h) = W$, and similarly for $V_h \subset V$.

In order to define an approximate solution of (18.17), we introduce the (bi)linear forms

$$a_h : W(h) \times V(h) \to \mathbb{R} \qquad \text{and} \qquad \ell_h : V(h) \to \mathbb{R}. \tag{18.18}$$

The approximate solution of the *primal problem* (18.17) $u_h \in W_h$ satisfies

$$a_h(u_h, \varphi_h) = \ell_h(\varphi_h) \qquad \forall \varphi_h \in V_h. \tag{18.19}$$

We assume that the numerical scheme (18.19) is *consistent*, i.e.,

$$a_h(u, \varphi) = \ell_h(\varphi) \qquad \forall \varphi \in V(h) \tag{18.20}$$

where $u \in W$ is the weak solution of (18.17). This implies the *Galerkin orthogonality of the error* of the primal problem

$$a_h(u_h - u, \varphi_h) = 0 \qquad \forall \varphi_h \in V_h. \tag{18.21}$$

Finally, we define the *residual of the primal problem* by

$$R_h[u_h](\varphi) := \ell_h(\varphi) - a_h(u_h, \varphi) = a_h(u - u_h, \varphi), \quad \varphi \in V(h), \tag{18.22}$$

where the last equality follows from the consistency (18.20) and the linearity of a_h.

18.2.2 Quantity of interest and the adjoint problem

As mentioned above, we are interested in the *quantity of interest* $J(u) \in \mathbb{R}$ given by the linear functional in the form

$$J(u) = (j_\Omega, u)_\Omega + (j_\Gamma, \mathscr{C} u)_\Gamma \,, \tag{18.23}$$

where j_Ω and j_Γ are given integrable functions on Ω and Γ, respectively, $\mathscr{C}$ is a boundary differential operator on Γ, and the symbols $(\cdot,\cdot)_\Omega$ and $(\cdot,\cdot)_\Gamma$ denote the $L^2(\Omega)$ and $L^2(\Gamma)$ scalar products, respectively.

In order to estimate the error $J(u) - J(u_h)$, we consider the *adjoint* (or *dual*) *problem* to (18.17) in the form: find $z : \Omega \to \mathbb{R}$ such that

$$a(w, z) = J(w) \qquad \forall w \in W. \tag{18.24}$$

Furthermore, we introduce the *approximate solution* of the *adjoint problem* (18.24) by $z_h \in V_h$ such that

$$a_h(w_h, z_h) = J(w_h) \qquad \forall w_h \in W_h, \tag{18.25}$$

where a_h and J are given by (18.18) and (18.23), respectively. Moreover, we assume that the numerical scheme (18.25) is *adjoint consistent*, i.e.,

$$a_h(w, z) = J(w) \qquad \forall w \in W(h), \tag{18.26}$$

where $z \in V$ is the weak solution of the adjoint problem (18.24). Relations (18.25) – (18.26) imply the *Galerkin orthogonality of the error* of the adjoint problem

$$a_h(w_h, z_h - z) = 0 \qquad \forall w_h \in W_h. \tag{18.27}$$

Finally, we define the *residual of the adjoint problem* by

$$R_h^*[z_h](w) := J(w) - a_h(w, z_h) = a_h(w, z - z_h), \quad w \in W(h), \tag{18.28}$$

where the last equality follows from the adjoint consistency (18.26).

18.2.3 Abstract goal-oriented error estimates

We are ready to derive abstract error estimates of the quantity of interest, i.e, the difference between the (unknown) exact value $J(u)$ and its approximation $J(u_h)$, which is obtained by computing u_h first and then by the evaluating J at u_h.

Let u and z be the exact solutions of the primal and adjoint problems (18.17) and (18.24), respectively, and similarly, let u_h and z_h be the approximate solutions of the primal and adjoint problems (18.19) and (18.25), respectively. Using the linearity of J and the adjoint consistency (18.26), we have the identity

$$J(u) - J(u_h) = J(u - u_h) = a_h(u - u_h, z). \tag{18.29}$$

The Galerkin orthogonality of the error (18.21) implies

$$a_h(u-u_h,z)=a_h(u-u_h,z-\varphi_h)=R_h[u_h](z-\varphi_h)\quad\forall\varphi_h\in V_h, \tag{18.30}$$

where the second equality follows from the definition of the primal residual $R_h[u_h](\cdot)$ (18.22). Hence, we obtain from (18.29) – (18.30) relation

$$J(u)-J(u_h)=R_h[u_h](z-\varphi_h)\quad\forall\varphi_h\in V_h, \tag{18.31}$$

which is called the *primal error identity*.

Moreover, setting $\varphi_h:=z_h$ in (18.31), exploiting the Galerkin orthogonality of the error of the adjoint problem (18.27) and the definition of the residual of the adjoint problem (18.28), we have

$$J(u)-J(u_h)=a_h(u-u_h,z-z_h)=a_h(u-w_h,z-z_h)=R_h^*[z_h](u-w_h)\quad\forall w_h\in W_h, \tag{18.32}$$

which is called the *adjoint error identity*. Obviously, there is the *residual equivalence* between the primal and adjoint residuals

$$R_h[u_h](z-\varphi_h)=R_h^*[z_h](u-w_h)\quad\forall\varphi_h\in V_h\ \forall w_h\in W_h. \tag{18.33}$$

18.2.4 Computable goal-oriented error estimates

The right-hand sides of (18.31) and (18.32) contain the exact adjoint solution z and the exact primal solution u, respectively, which are unknown. Therefore, in order to have a computable error estimate, we have to use approximations

$$u\approx u_h^+\in W_h^+\quad\text{ and }\quad z\approx z_h^+\in V_h^+, \tag{18.34}$$

where $W_h^+\subset W(h)$ and $V_h^+\subset V(h)$ are "richer" finite dimensional spaces than W_h and V_h, respectively. We note that the choice $u_h^+\in W_h$ and $z_h^+\in V_h$ leads to nullification of the right-hand sides of (18.31) and (18.32) due to the Galerkin orthogonalities (18.21) and (18.27), respectively. We set

$$u_h^+=\mathscr{R}(u_h),\qquad z_h^+=\mathscr{R}^*(z_h), \tag{18.35}$$

where $\mathscr{R}:W_h\to W_h^+$ and $\mathscr{R}^*:V_h\to V_h^+$ denote formally higher order reconstruction operators.

Then using (18.34) – (18.35), we obtain from (18.31) and (18.32) the computable goal-oriented error estimates

$$\begin{aligned}&J(u-u_h)\approx R_h[u_h](z_h^+-\varphi_h),\quad\varphi_h\in V_h\\&J(u-u_h)\approx R_h^*[z_h](u_h^+-w_h),\quad w_h\in W_h.\end{aligned} \tag{18.36}$$

Both right-hand sides in (18.36) are independent of the choice of φ_h and w_h due to the Galerkin orthogonalities of error. However, this is not true in practical computations when u_h and z_h suffer from algebraic errors. Then the choice of φ_h and w_h has impact on the resulting estimates. In order to obtain accurate error estimates, it is advantageous to minimize the arguments of $R_h[u_h](\cdot)$ and $R_h^*[z_h](\cdot)$. There are two natural possibilities:

$$\text{(i)} \quad \varphi_h := z_h, \qquad w_h := u_h, \tag{18.37a}$$

$$\text{(ii)} \quad \varphi_h := \Pi^* z_h^+, \quad w_h := \Pi u_h^+, \tag{18.37b}$$

where $\Pi : W_h^+ \to W_h$ and $\Pi^* : W_h^+ \to W_h$ are suitable projections. Hence, for both choices from (18.37), we obtain from (18.36) the estimates

$$J(u - u_h) \approx \tfrac{1}{2} \left(R_h[u_h](z_h^+ - z_h) + R_h^*[z_h](u_h^+ - u_h) \right), \tag{18.38a}$$

$$J(u - u_h) \approx \tfrac{1}{2} \left(R_h[u_h](z_h^+ - \Pi^* z_h^+) + R_h^*[z_h](u_h^+ - \Pi u_h^+) \right). \tag{18.38b}$$

The choice (18.37b) (and the corresponding estimate (18.38b)) is more suitable for the forthcoming anisotropic error estimates, which serve as the base of the anisotropic mesh adaptation process.

18.3 Dual weighted residuals for the Laplace equation

Let us consider again problem (18.1), with its weak formulation (18.2)–(18.3) and numerical approximation (18.4).

Let the quantity of interest is given by (cf. (18.23))

$$J(v) = (j_\Omega, v)_{L^2(\Omega)} + (j_{\Gamma_D}, \nabla v \cdot \boldsymbol{n})_{\Gamma_D} + (j_{\Gamma_N}, v)_{\Gamma_N}, \quad v \in H^1(\mathscr{T}_h), \tag{18.39}$$

where $j_{\Gamma_D}, j_{\Gamma_N} \in L^2(\Gamma)$ and $j_\Omega \in L^2(\Omega)$ are given weight functions and $(\cdot,\cdot)_M$ denotes the L^2-scalar product over the set $M \in \mathbb{R}^n$, $n = 1, 2, 3$.

Then the adjoint problem to (18.3) reads: find $z \in H^1(\Omega)$ such that

$$\text{(i)} \quad z - j_{\Gamma_D}^* \in H_D^1(\Omega), \tag{18.40}$$

$$\text{(ii)} \quad \int_\Omega \nabla w \cdot \nabla z \,\mathrm{d}x = \int_\Omega j_\Omega w \,\mathrm{d}x + \int_{\Gamma_N} j_{\Gamma_N} w \,\mathrm{d}S \quad \forall w \in H_D^1(\Omega), \tag{18.41}$$

where $H_D^1(\Omega)$ is the same space as in (18.2)–(18.3) and $j_{\Gamma_D}^* \in H_D^1(\Omega)$ denotes a function such that $j_{\Gamma_D}^* = j_{\Gamma_D}$ on Γ_D, cf. (18.39).

In order to derive goal-oriented error estimate for the approximate solution given by (18.4), we employ the primal error identity (18.31) and the definition of the residual (18.5), i.e.

$$\begin{aligned} J(u) - J(u_h) &= R_h[u_h](z - \varphi_h) \\ &= \int_\Omega f(z - \varphi_h) \,\mathrm{d}x + \int_{\Gamma_N} g_N(z - \varphi_h) \,\mathrm{d}S - \int_\Omega \nabla u_h \cdot \nabla(z - \varphi_h) \,\mathrm{d}x \end{aligned} \tag{18.42}$$

for any $\varphi_h \in V_h$. Using the same procedure as in (18.9), (18.11), we obtain, similarly as in the fist inequality of (18.14), the estimate

$$J(u) - J(u_h) = R_h[u_h](z - \varphi_h) \tag{18.43}$$
$$\leq \sum_{K \in \mathscr{T}_h} \|R_K(u_h)\|_K \|z - \varphi_h\|_K + \sum_{\gamma \in \mathscr{F}_h} \|R_\Gamma(u_h)\|_\gamma \|z - \varphi_h\|_\gamma, \quad \varphi_h \in V_h,$$

where the element and edge residuals $R_K(u_h)$ and $R_\Gamma(u_h)$, respectively, are given by (18.10).

We observe that, in contrary to (18.14) (and thus to (18.16)), the local residuals $\|R_K(u_h)\|_K$ and $\|R_\Gamma(u_h)\|_\gamma$ are not taken as a pure sum but they are weighted by local element and face weights $\|z - \varphi_h\|_K$ and $\|z - \varphi_h\|_\gamma$, respectively. Therefore, we speak about the dual weighted residual error estimates.

In similar way, we can derive the alternative formula from (18.32) as

$$J(u) - J(u_h) = R_h^*[z_h](u - \varphi_h) \tag{18.44}$$
$$\leq \sum_{K \in \mathscr{T}_h} \|R_K^*(z_h)\|_K \|u - \varphi_h\|_K + \sum_{\gamma \in \mathscr{F}_h} \|R_\Gamma^*(z_h)\|_\gamma \|u - \varphi_h\|_\gamma, \quad \varphi_h \in V_h,$$

where the dual element and edge residuals $R_K^*(u_h)$ and $R_\Gamma^*(u_h)$, respectively, are given by formulas similar to (18.10).

Finally, computable error estimates can be obtain by the use of the higher-order approximation (18.34)–(18.35), i.e. for the case (18.38a) we have

$$J(u) - J(u_h) \approx R_h[u_h](z_h^+ - z_h) \tag{18.45}$$
$$\leq \sum_{K \in \mathscr{T}_h} \|R_K(u_h)\|_K \|z_h^+ - z_h\|_K + \sum_{\gamma \in \mathscr{F}_h} \|R_\Gamma(u_h)\|_\gamma \|z_h^+ - z_h\|_\gamma.$$

18.4 Goal-oriented mesh adaptation

We describe how the error estimate (18.45) can be used for the mesh adaptation. We note that there are several other possibilities. Although the estimate (18.45) has been derived for the Laplace equation discretized by conforming finite element method, the solution of another (linear) problem by any (finite element based) numerical method leads to the same (or very similar) formula with different relations for the local residuals $R_K(u_h)$ and $R_\Gamma(u_h)$. Some examples and more detailed analysis can by found in [DM22].

We set

$$\eta^{\mathrm{I}}(u_h, z_h) := R_h[u_h](z_h^+ - z_h), \tag{18.46}$$

where $R_h[u_h](\cdot)$ is the residual of the given (primal) problem evaluated at the approximate solution $u_h \in W_h$, $z_h \in V_h$ is the approximate solution of the adjoint problem and $z_h^+ \in V_h^+$ is the corresponding higher-order approximation.

Following from formula (18.45), we have the error estimate of type I

$$J(u) - J(u_h) \approx \eta^{\mathrm{I}}(u_h, z_h), \tag{18.47}$$

where η^{I} is given by (18.46). For the mesh adaptation, we need an information about the local distribution of the error. Therefore, let $\{\chi_K,\ K \in \mathscr{T}_h\}$ by a partition of the unity, i.e. $\chi_K : \Omega \to \mathbb{R}$, $K \in \mathscr{T}_h$ such that $\sum_{K\in\mathscr{T}_h} \chi_K = 1$. Then due to the linearity of $R_h[\boldsymbol{u}_{2h}](\cdot)$ we set

$$\eta^{\mathrm{I}}_K(u_h, z_h) := R_h[u_h](\chi_K(z_h^+ - z_h)), \quad K \in \mathscr{T}_h, \tag{18.48}$$

consequently $\eta^{\mathrm{I}}(u_h, z_h) = \sum_{K\in\mathscr{T}_h} \eta^{\mathrm{I}}_K(u_h, z_h)$. The quantities η^{I}_K are called the local error estimate of type I.

Alternatively, we can use the inequality in (18.45) and obtain

$$\begin{aligned}
\eta^{\mathrm{I}}(u_h, z_h) &= R_h[u_h](z_h^+ - z_h) \qquad (18.49)\\
&\le \sum_{K\in\mathscr{T}_h} \|R_K(u_h)\|_K \|z_h^+ - z_h\|_K + \sum_{\gamma\in\mathscr{F}_h} \|R_\Gamma(u_h)\|_\gamma \|z_h^+ - z_h\|_\gamma\\
&= \sum_{K\in\mathscr{T}_h} \|R_K(u_h)\|_K \|z_h^+ - z_h\|_K + \sum_{K\in\mathscr{T}_h} \sum_{\gamma\in\partial K} \delta_K \|R_\Gamma(u_h)\|_\gamma \|z_h^+ - z_h\|_\gamma\\
&= \sum_{K\in\mathscr{T}_h} \underbrace{\left(\|R_K(u_h)\|_K \|z_h^+ - z_h\|_K + \sum_{\gamma\in\partial K} \delta_\gamma \|R_\Gamma(u_h)\|_\gamma \|z_h^+ - z_h\|_\gamma \right)}_{=\eta^{\mathrm{II}}_K(u_h,z_h)} =: \eta^{\mathrm{II}}(u_h, z_h),
\end{aligned}$$

where $\delta_\gamma = \frac{1}{2}$ if γ is an interior edge and $\delta_\gamma = 1$ if γ belongs to Γ. The quantity η^{II} is called the error estimate of type II and η^{II}_K, $K \in \mathscr{T}_h$ are the corresponding local estimates.

Let η denote either η^{I} or η^{II} and similarly let η_K denote either η^{I}_K or η^{II}_K for $K \in \mathscr{T}_h$. Let $\mathrm{TOL} > 0$ be a given tolerance, the goal of the computation is to adapt the mesh $\mathscr{T}_h$ such that the corresponding approximate solutions satisfy

$$\eta(u_h, z_h) \le \mathrm{TOL}. \tag{18.50}$$

The natural mesh adaptive procedure is given by Algorithm 18.1.

The step of Algorithm 18.14 can be carried out by several techniques. Hereafter, we mention two of them.

Isotropic mesh refinement

For each $K \in \mathscr{T}_h$, we evaluate η_K. Then we mark elements for the refinement: either the fix ratio of elements having the highest value of η_K or elements such that $\eta_K > \rho \max_{K\in\mathscr{T}_h} \eta_K$, where $\rho \in (0, 1)$. The marked elements are refined by a technique from Section 13.3.

Algorithm 18.1: Mesh adaptive algorithm

1: let TOL > 0 and initial mesh $\mathscr{T}_h$ be given
2: solve the primal and dual problems on $\mathscr{T}_h$ resulting u_h and z_h
3: **while** $\eta(u_h, z_h) >$ TOL **do**
4: using η_K construct a new mesh $\mathscr{T}_h^N$
5: recompute u_h and z_h on $\mathscr{T}_h^N$
6: $\mathscr{T}_h := \mathscr{T}_h^N$
7: solve the primal and dual problems on $\mathscr{T}_h$ resulting u_h and z_h
8: **end while**

Anisotropic mesh refinement

This technique admits not only the refinement/coarsening of the mesh but also the modification of the shape and orientation of elements. The detailed description can be found in [DM22]. The main idea is the following: each element K

- set the size (=volume) of elements using η using equi-distribution error principle, i.e., the goal is that $\eta_K = \text{const}$ for all $K \in \mathscr{T}_h$,
- find the optimal shape of each element using estimate of the type II, namely, the residuals $R_K(u_h)$ and $R_\Gamma(u_h)$ in η_K^{II} (cf. (18.45)) are kept fixed and we optimize the shape by minimizing the weights $\|z_h^+ - z_h\|_K$ and $\|z_h^+ - z_h\|_\gamma$,
- the new mesh is constructed using anisotropic mesh adaptation method, see `http://msekce.karlin.mff.cuni.cz/~dolejsi/Vyuka/AMA.pdf`

Chapter 19

Parareal method for ODE

This text is the copy of en.wikipedia.org/wiki/Parareal. Parareal is a parallel algorithm from numerical analysis and used for the solution of initial value problems. It was introduced in 2001 by Lions, Maday and Turinici [LMT01]. Since then, it has become one of the most widely studied parallel-in-time integration methods.

19.1 Introduction

The goal is to solve an initial value problem of the form

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(t,u) \quad \text{for} \quad t \in [t_0, T] \quad \text{with} \quad u(t_0) = u^0. \tag{19.1}$$

The right hand side f is assumed to be a smooth (possibly nonlinear) function. It can also correspond to the spatial discretization of a partial differential equation in a method of lines approach. We wish to solve this problem on a temporal mesh of $N+1$ equally spaced points $(t_0, t_1, \ldots, t_N)$, where $t_{j+1} = t_j + \Delta T$ and $\Delta T = (T - t_0)/N$. Carrying out this discretization we obtain a partitioned time interval consisting of time slices $[t_j, t_{j+1}]$ for $j = 0, \ldots, N-1$.

The objective is to calculate numerical approximations U_j to the exact solution $u(t_j)$ using a serial time-stepping method (e.g. Runge-Kutta) that has high numerical accuracy (and therefore high computational cost). We refer to this method as the fine solver $\mathcal{F}$, which propagates an initial value U_j at time t_j to a terminal value U_{j+1} at time t_{j+1}. The goal is to calculate the solution (with high numerical accuracy) $\mathcal{F}$ such that we obtain

$$U_{j+1} = \mathcal{F}(t_j, t_{j+1}, U_j), \quad \text{where} \quad U_0 = u^0. \tag{19.2}$$

The problem with this (and the reason for attempting to solve in parallel in the first place) solution is that it is computationally infeasible to calculate in real-time.

19.2 Main idea of parareal method

Instead of using a single processor to solve the initial value problem (as is done with classical time-stepping methods (19.2)), Parareal makes use of N processors. The aim to is to use N processors to solve N smaller initial value problems (one on each time slice) in parallel. For example, in a MPI based code, N would be the number of processes, while in an OpenMP based code, N would be equal to the number of threads.

Parareal makes use of a second time-stepping method to solve this initial value problem in parallel, referred to as the coarse solver $\mathcal{G}$. The coarse solver works the same way as the fine solver, propagating an initial value over a time interval of length ΔT, however it does so at much lower numerical accuracy than $\mathcal{F}$ (and therefore at much lower computational cost). Having a coarse solver that is much less computationally expensive than the fine solver is the key to achieving parallel speed-up with Parareal.

Henceforth, we will denote the Parareal solution at time t_j and iteration k by U_j^k.

Zeroth Iteration Firstly, run the coarse solver serially over the entire time interval $[t_0, T]$ to calculate an approximate initial guess to the solution:

$$U_{j+1}^0 = \mathcal{G}(t_j, t_{j+1}, U_j^0), \quad j = 0, \ldots, N-1. \tag{19.3}$$

Subsequent Iterations Next, run the fine solver on each of the time slices, in parallel, from the most up-to-date solution values:

$$\mathcal{F}(t_j, t_{j+1}, U_j^{k-1}), \quad j = 0, \ldots, N-1. \tag{19.4}$$

Now update the parareal solution values sequentially using the predictor-corrector:

$$U_{j+1}^k = \mathcal{G}(t_j, t_{j+1}, U_j^k) + \mathcal{F}(t_j, t_{j+1}, U_j^{k-1}) - \mathcal{G}(t_j, t_{j+1}, U_j^{k-1}), \quad j = 0, \ldots, N-1. \tag{19.5}$$

At this stage, one can use a stopping criterion to determine whether the solution values are no longer changing each iteration. For example, one may test this by checking if

$$|U_j^k - U_j^{k-1}| < \varepsilon \quad \forall\ j \leq N, \tag{19.6}$$

and some tolerance $\varepsilon > 0$. If this critertion is not satisfied, subsequent iterations can then be run by applying the fine solver in parallel and then the predictor-corrector. Once the criterion is satisfied, however, the algorithm is said to have converged in $k \leq N$ iterations. Note that other stopping criterion do exist and have been successfully tested in Parareal.

19.2.1 Convergence of the method

Parareal should reproduce the solution that is obtained by the serial application of the fine solver and will converge in a maximum of N iterations [GV07]. For Parareal to provide speedup, however, it has to converge in a number of iterations significantly smaller than the number of time slices, i.e. $k \ll N$.

In the Parareal iteration, the computationally expensive evaluation of $\mathcal{F}(t_j, t_{j+1}, U_j^{k-1})$ can be performed in parallel on N processing units. By contrast, the dependency of U_{j+1}^k on $\mathcal{G}(t_j, t_{j+1}, U_j^k)$ means that the coarse correction has to be computed in serial order.

Typically, some form of Runge-Kutta method is chosen for both coarse and fine integrator, where $\mathcal{G}$ might be of lower order and use a larger time step than $\mathcal{F}$. If the initial value problem stems from the discretization of a PDE, $\mathcal{G}$ can also use a coarser spatial discretization, but this can negatively impact convergence unless high order interpolation is used.

19.2.2 Computational costs

Under some assumptions, a simple theoretical model for the speedup of Parareal can be derived. Although in applications these assumptions can be too restrictive, the model still is useful to illustrate the trade offs that are involved in obtaining speedup with Parareal.

First, assume that every time slice $[t_j, t_{j+1}]$ consists of exactly N_f steps of the fine integrator and of N_c steps of the coarse integrator. This includes in particular the assumption that all time slices are of identical length and that both coarse and fine integrator use a constant step size over the full simulation. Second, denote by τ_f and τ_c the computing time required for a single step of the fine and coarse methods, respectively, and assume that both are constant. This is typically not exactly true when an implicit method is used, because then runtimes vary depending on the number of iterations required by the iterative solver.

Under these two assumptions, the runtime for the fine method integrating over P time slices can be modelled as

$$c_{\text{fine}} = NN_f\tau_f. \tag{19.7}$$

The runtime of Parareal using P processing units and performing k iterations is

$$c_{\text{parareal}} = (k+1)NN_c\tau_c + kN_f\tau_f. \tag{19.8}$$

Speedup of Parareal then is

$$S_p = \frac{c_{\text{fine}}}{c_{\text{parareal}}} = \frac{1}{(k+1)\frac{N_c}{N_f}\frac{\tau_c}{\tau_f} + \frac{k}{N}} \leq \min\left\{\frac{N_f\tau_f}{N_c\tau_c}, \frac{N}{k}\right\}. \tag{19.9}$$

These two bounds illustrate the trade off that has to be made in choosing the coarse method: on the one hand, it has to be cheap and/or use a much larger time step to make the first bound as large as possible, on the other hand the number of iterations k has to be kept low to keep the second bound large. In particular, Parareal's parallel efficiency is bounded by

$$E_p = \frac{S_p}{N} \leq \frac{1}{k}, \tag{19.10}$$

that is by the inverse of the number of required iterations.

Chapter 20

Paralell computations through MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures with distributed memory. It admits to pararellize (and accelerate) computations using several processors. The basic aspects of (non only) MPI computation are the following:

- we have one code, which is executed at one or more processors, see Section 20.1.3,
- if no MPI action is implemented then all memory allocation of the code and all computations are carried out separately on all processors,
- its is possible to pararellize the computational operations
- its is possible to pararellize the memory allocation
- data among the processors can be share and/or interchanged by MPI command, see Section 20.2

Efficiency of the parallel implementation:

- *computational time – speed up*: T_1/T_N, where T_1 and T_N are the computational time for the code running using 1 and N processors, respectively. In ideal situation is $T_1/T_N = N$,
- *memory* – M_1/M_N, where M_1 is the total amount of memory of the computation executed by 1 processors, M_N is the total amount of memory of the computation executed by N processors, i.e., the sum of memory for all M processors. In ideal situation $M_1/M_N = 1$.

20.1 MPI + fortran: installation & execution

Using *Ubuntu*, the installation of several libraries is required

```
sudo apt install openmpi-bin openmpi-common openmpi-doc libopenmpi-dev
sudo apt install openssh-client openssh-server
```

20.1.1 Code

Example of a code using MPI computation (file `DWR.f90`):

```
 program DWR_tdp
  implicit none
  include "mpif.h"
  integer :: myid, comm_all, nproc, ierr

  call MPI_INIT(ierr)   ! MPI initialization

  comm_all  = MPI_COMM_WORLD    ! Communicator
  call MPI_COMM_RANK(comm_all,myid,ierr)
  call MPI_COMM_SIZE(comm_all,nproc,ierr)

  ! comm_all  ... communicator, the use is below
  ! nproc  ... number of processors
  ! myid  ... index of the processor from 0 to nproc

  !  itself code follows
  write(*,*) "I am proc ", myid, " from ", nproc

  call MPI_FINALIZE(ierr)  ! MPI finalization
end program
```

20.1.2 Makefile

Example of `Makefile`

```
TARGETS = DWR.o
FFLAGS=  -fPIC -fopenmp -O2 -w -ffpe-trap=invalid,zero,overflow
LIBS=-lscalapack-openmpi \
     -lmkl_gf_lp64 -lmkl_sequential -lmkl_core

FXX=mpif90

all: DWR
DWR: $(TARGETS)
       $(FXX) $(FFLAGS) -o DWR $^ $(LIBS)

%.o:%.f90
       $(FXX) $(FFLAGS) -c $?
```

20.1.3 Execution of the code

Example of the running of the code (using 4 processors):

```
  mpirun -np 4 ./DWR
```

20.2 Examples of several commands of MPI

The following integer values have to be available:

```
  comm_all  = MPI_COMM_WORLD    ! Communicator
  call MPI_COMM_RANK(comm_all,myid,ierr)     ! myid  ... index of the processor
  call MPI_COMM_SIZE(comm_all,nproc,ierr)    ! nproc ... number of processors
```

where `ierr` is (here and hereafter) an integer, if `ierr > 0` there is a trouble.

`call MPI_Bcast(b, length, MPI_DOUBLE_PRECISION, send_proc, comm_all, ierr)`, where `b` is the double array of size `length` allocated at all processors, the values of `b` at processor `send_proc` (an integer number) are copied to all other processors

---

```
 call MPI_ALLREDUCE(MPI_IN_PLACE, Ju, size(Ju), MPI_DOUBLE_PRECISION,  &
          MPI_SUM, comm_all, ierr)
```

the double precision array `Ju` on all processors are summed together and refreshed at all processors

Interchange of data between `send_proc` and `recv_proc` processors.

```
  integer, dimension(:), allocatable :: request
  integer, dimension(:,:), allocatable :: statarray

 lrequest = <maxial number of requests (sendings)>
 lstatarray1 = MPI_STATUS_SIZE
 lstatarray2 = lrequest

 allocate( request(1: lrequest) )
 allocate( statarray(1: lstatarray1, 1:lstatarray2) )

do i=1,N
! preparation of commits
  itag = <value>

  if(recv_proc == myid) then
    length = nsize
    ireq = ireq + 1
    call MPI_Irecv(b, length, MPI_DOUBLE_PRECISION, &
         send_proc, itag, comm_all, request(ireq), ierr)

  elseif(send_proc == myid) then
    length = nsize
    ireq = ireq + 1
    call MPI_Isend(b, length, MPI_DOUBLE_PRECISION, &
         recv_proc, itag, comm_all, request(ireq), ierr)
   endif
end do

! all commits are send at once
call MPI_waitall(nreq, request, statarray, ierr)
```

```
call MPI_ALLGATHERV(b_loc, proc_size(myid), &
     MPI_DOUBLE_PRECISION, b, proc_size(:), &
     proc_offsets(:), MPI_DOUBLE_PRECISION, comm_all,ierr)
```

for each processor, a local array `b_loc` is available, the sizes of these arrays are stored in array `proc_size(:)`. Array `b` is a global array (at all processor), which arises by join of the local arrays "step-by-step". Integer array `proc_offsets` stores the off-sets of the local arrays.

Chapter 21

Additional usefull numerical software packages

21.1 LASPACK

LASPack is a package for solving large sparse systems of linear equations like those which arise from discretization of partial differential equations. Simple code with a fast implementation written in C language.

21.1.1 Installation

Download the library

```
  tar xfz laspack.tgz
  cd laspack
 ./install
```

Library is installed in directory `~/lib/` as files

```
-rw-r--r--  1 dolejsi dolejsi 180640 kvě  6 08:28 liblaspack.a
-rw-r--r--  1 dolejsi dolejsi   5810 kvě  6 08:28 libxc.a
```

Moreover, there arise a directory `~/include/laspack` containing all heading files (*.h)

21.1.2 Use of LASPACK for your own code

File `Makefile`

```
TARGETS1=main.o problem.o scalar.o euler.o mesh.o fem.o polynom.o geometry.o
     integration.o matrix.o file-io.o f_data.o

CFLAGS=-Wall -ansi -fPIC -Wno-deprecated
```

```
CXX=g++
all: Dgfem
Dgfem: $(TARGETS1)
 $(CXX) $(CFLAGS) -o Dgfem $^  -L/home/dolejsi/lib -llaspack

clean:
 -rm -f Dgfem *.o *.so
%.o:%.cpp
 $(CXX) $(CFLAGS) -c -I/home/dolejsi/include $?
```

String `-L/home/dolejsi/lib -llaspack` gives the path to the library.

If your file `matrix.cpp` contains calls of LASPACK subroutines then the corresponding header file `matrix.h` has to contains the LASPACK headers:

```
 * matrix.h - header file for basic finite elements

#ifndef __MATRIX_HEADER_FILE__

#define __MATRIX_HEADER_FILE__
#include <iostream.h>

extern "C" {

#include <laspack/vector.h>
#include <laspack/errhandl.h>
#include <laspack/operats.h>
#include <laspack/qmatrix.h>
#include <laspack/precond.h>
#include <laspack/itersolv.h>
#include <laspack/rtc.h>

}
```

21.2 FreeFEM++

`FreeFem++` is a software to solve partial differential equations numerically. As its name says, it is a free software (see copyright for full detail) based on the Finite Element Method; it is not a package, it is an integrated product with its own high level programming language. This software runs on all unix OS (with g++ 2.95.2 or later, and X11R6) , on Window95, 98, 2000, NT, XP, and MacOS X.

21.2.1 Installation

For some platforms, you can download executable library of `FreeFEM++`. This section describes instalation from source files.[1]

After downloading the archive `freefem++-3.30.tar.gz` into you directory, use the following commands, which installs `FreeFEM++` into the current directory:

```
tar xfz freefem++-3.30.tar
cd freefem++-3.30/
more README
less INSTALL

./configure
make
make install
make clean
```

The command `./configure` automatically finds the correct translators and create the file `Makefile`. After command `make` the instalation leads to several executable files, e.g., `src/nw/FreeFem++`.

21.2.2 Running of FreeFEM++

For details see manual, here are only few commands for an example.

```
cd examples++-tutorial
../src/nw/FreeFem++ LaplaceP1.edp
../src/nw/FreeFem++ mycode.edp
../src/nw/FreeFem++ adapt.edp
```

21.3 UMFPACK

UMFPACK is a set of routines for solving unsymmetric sparse linear systems, $Ax = b$, using the Unsymmetric-pattern MultiFrontal method and direct sparse LU factorization. It is written in ANSI/ISO C, with a MATLAB interface. UMFPACK relies on the Level-3 Basic Linear Algebra Subprograms (BLAS) (dense matrix multiply) for its performance. This code works on Windows and many versions of Unix (Sun Solaris, Red Hat Linux, IBM AIX, SGI IRIX, and Compaq Alpha).

You will need to install AMD library to use UMFPACK. The UMFPACK and AMD subdirectories must be placed side-by-side within the same parent directory. AMD is a

[1] FreeFEM++ requires several software Linux packages which may miss on your computer.

stand-alone package that is required by UMFPACK. UMFPACK can be compiled without the BLAS but your performance will be much less than what it should be.

Installation is a little more complicated, see the manual.

21.4 Software for visualization

See also Lecture Notes about Finite Element Method.

21.4.1 gnuplot

Simple and efficient code namely for 2D graphs.

21.4.2 Techplot

Commercial software, available at our faculty.

Appendix

A possible code for Homework 18

```
  program stab_euler

    alpha = 2.

    tol = 1E-4
    a = 0.
    b = 10.

    h = 2E-2

    x = a

    ! IC
    y = exp(-alpha*(x-1)**2)

    write(11,'(6es12.4)') h, x, y, exp(-alpha*(x-1)**2)

10  continue
    ynew = y + h * f(x, y, alpha)

    ypp = ( f(x+h, ynew, alpha) - f(x, y, alpha)) /h
    hnew = (2*tol / abs(ypp) )**0.5
```

```
    x = x + h
    y = ynew
    write(11,'(6es12.4)') h, x, y, exp(-alpha*(x-1)**2), ypp, hnew

    ! comment the following line for a fixed time step
    h = hnew

    if( x < b) goto 10

  end program stab_euler


  function f(x, y, alpha)
    real :: f
    real :: x, y, alpha

    f = -2* alpha * (x-1) * y

  end function f
```

A possible code for Homework 19

```
       program stiff

c234567890123456789012345678901234567890123456789012345678901234567890123456789012
      implicit DOUBLE PRECISION  (a-h,o-z)
      dimension a(2,2), aa(2,2), u(2), u_n(2)

      ifile = 10

      write(*,*) "Put method type (1=explixit), (2= implicit)"
      read(*,*) i_method

      write(*,*) "Put $b$"
      read(*,*) x_max

      write(*,*) "Put time step $h$"
      read(*,*) tau

      if(i_method .eq. 1) then
         print*,'Explicit Euler method'
```

```
      elseif(i_method .eq. 2) then
         print*,'Implicit Euler method'
      else
         print*,'Unknown method'
         stop
      endif

      print*,'Solution at interval  (0., ',x_max,')'
      print*,'step h = ',tau
      print*

      a(1,1) = 998.0D+00
      a(1,2) = 1998.0D+00
      a(2,1) = -999.0D+00
      a(2,2) = -1999.0D+00

      k_max = x_max/tau
      time = 0.

c     IC
       u(1) = 1.0D+00
       u(2) = 0.0D+00

       open(ifile, file='u_h' , status='UNKNOWN')

       if(i_method .eq. 1) then
          write(ifile,*) 0., u(1), u(2), 0
          do i=1, k_max

             u_n(1) = u(1) + tau*(a(1,1)* u(1) + a(1,2)*u(2) )
             u_n(2) = u(2) + tau*(a(2,1)* u(1) + a(2,2)*u(2) )
             time = time + tau
             u(1) = u_n(1)
             u(2) = u_n(2)

             write(*,*) time, u(1), u(2), i
             write(ifile,*) time, u(1), u(2), i
          enddo

       elseif(i_method .eq. 2) then

          write(ifile,*) 0., u(1), u(2), 0
          do i=1, k_max
```

```
      aa(1,1) = 1.D+00 - tau*a(1,1)
      aa(1,2) =  - tau*a(1,2)
      aa(2,1) =  - tau*a(2,1)
      aa(2,2) = 1.D+00 - tau*a(2,2)

      det = aa(1,1)*aa(2,2) - aa(1,2)*aa(2,1)

      det1 = u(1)*aa(2,2) - u(2)*aa(1,2)
      det2 = u(2)*aa(1,1) - u(1)*aa(2,1)

      u_n(1) = det1/det
      u_n(2) = det2/det

      time = time + tau
      u(1) = u_n(1)
      u(2) = u_n(2)


      write(*,*) time, u(1), u(2), i
      write(ifile,*) time, u(1), u(2), i
   enddo

 endif

 close(ifile)

 print*,'Computation finished'
end
```

Bibliography

[BR01] R. Becker and R. Rannacher. An optimal control approach to a-posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.

[BR03] Wolfgang Bangerth and Rolf Rannacher. *Adaptive Finite Element Methods for Differential Equations*. Lectures in Mathematics. ETH Zürich. Birkhäuser Verlag, 2003.

[DM22] V. Dolejší and G. May. *Anisotropic hp-Mesh Adaptation Methods*. Birkhäuser, 2022.

[EG04] Alexandre Ern and Jean-Luc Guermond. *Theory and Practice of Finite Elements*. Springer, 2004.

[GS02] M. Giles and E. Süli. Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality. *Acta Numerica*, 11:145–236, 2002.

[GV07] Martin J. Gander and Stefan Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM J. Sci. Comput.*, 29(2):556–578, 2007.

[Hac85] Wolfgang Hackbusch. *Multigrid methods and applications*, volume 4 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1985.

[Har07] Ralf Hartmann. Adjoint Consistency Analysis of Discontinuous Galerkin Discretizations. *SIAM J. Numer. Anal.*, 45(6):2671–2696, 2007.

[JSdlKdRB16] P. Jaysava, D. V. Shantsev, S. de la Kethulle de Ryhove, and T. Bratteland. Fully anisotropic 3-d em modelling on a lebedev grid with a multigrid preconditioner. *Geophysical Journal International*, 2016.

[LMT01] J.-L. Lions, Y. Maday, and G. Turinici. A "parareal" in time discretization of pde's [résolution d'edp par un schéma en temps "pararéel"]. *Comptes Rendus de l'Academie des Sciences - Series I: Mathematics*, 332(7):661–668, 2001.

[Pat69] T. N. L. Patterson. Table errata: "The optimum addition of points to quadrature formulae" (Math. Comp. **22** (1968), 847–856; addendum, ibid. **22** (1968), no. 104, loose microfiche suppl. C1-C11). *Math. Comp.*, 23(108):892, 1969.

[QSS00] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer-Verlag, 2000.

[SZ90] R. Scott and S. Zhang. Finite element interpolation of nonsmooth functions satisfying boundary conditions. *Math. Comp.*, 54(190):483–493, 1990.

[Ver] Glossary of verification and validation terms. http://www.grc.nasa.gov/WWW/ wind/valid/tutorial/glossary.html.

[Ver13] R. Verfürth. *A Posteriori Error Estimation Techniques for Finite Element Methods*. Numerical Mathematics and Scientific Computation. Oxford University Press, 2013.

[Wat02] D. S. Watkins. *Fundamentals of Matrix Computations*. Pure and Applied Mathematics, Wiley-Interscience Series of Texts, Monographs, and Tracts. Wiley, New York, 2002.