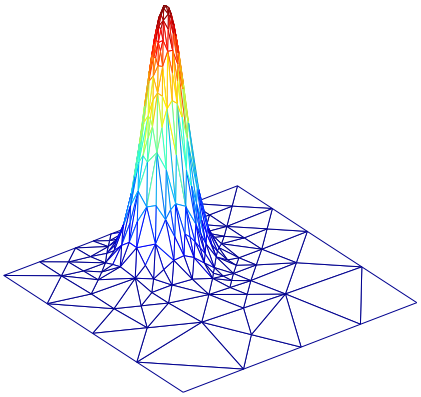
# The Spectral/*hp*-Finite Element Method for Partial Differential Equations



A. P. Engsig-Karup

Scientific Computing Section, DTU Compute
Technical University of Denmark, Bldg. 321
DK-2800 Kgs.-Lyngby, Denmark

December 25, 2025

# Contents

# List of Algorithms

# Preface

This set of lecture notes provides an elementary introduction to both the classical Finite Element Method (FEM) and the extended Spectral/*hp*-Finite Element Method for solving Partial Differential Equations (PDEs). Many problems in science and engineering can be formulated mathematically and involves one or more PDEs. The FEM is nowadays an important numerical discretization technique for approximately solving such mathematical equations on a computer.

The set of lecture notes has been written for engineering students for use in the short three-week course *02623 The Finite Element Method for Partial Differential Equations* given at the Technical University of Denmark. The basic aim of the current lecture notes follows that of the earlier lecture notes for the course [9], which is to describe the FEM in a way that supports the reader in implementing the method independently. The original set of course notes has been modified and updated and additional chapters describing the high-order extensions to form the Spectral/*hp*-Finite Element Method have been included. Thus the significant contributions of Chapters 1, 2 and 5 covering the classical Finite Element Method are in large parts due to V. A. Barker and J. Reffstrup.

With this set of lecture notes it should be possible for the reader to make a Spectral/*hp*-FEM toolbox in successive steps with the support given in the text. Emphasis is on the practical details supported with basic and sufficient theory to build the foundation in a three weeks period where the tools are developed and applied immediately. Furthermore, the aim of guiding the reader to develop a Spectral/*hp*-FEM toolbox is to provide a simple and generic framework for developing small prototype applications rather than directly approaching large-scale models. With this in mind, the goal is to let the reader encounter the typical problems involved in the practical implementation of these models and thereby gain a fundamental understanding of the algorithms and their practical implementation.

For the practical work, a number of templates described using pseudo programming code, should be understood and converted by the reader to a programming language in a concrete implementation. A number of exercises is given which in a step-by-step manner guides the reader toward developing the necessary subroutines which can be used to solve typical and fairly general PDEs in one or two spatial dimensions. In the course the chosen programming environment is Matlab, however, this is by no means a necessary requirement.

The mathematical level needed to grasp the details of this set of notes requires an elementary background in mathematical analysis and linear algebra. Each chapter is supplemented with hands-on exercises and the amount of material covered is intended to be balanced in such a way that each subject amounts to approximately one week of work including producing a small report to document and communicate the work effort.

This set of lecture notes is currently a working draft and may contain some (hopefully) minor errors. Any suggestions for improving the notes or feedback on errors and the content and its structure will be highly appreciated. Please report to the email apek@dtu.dk.

Allan P. Engsig-Karup, December 25, 2025.

# Introduction

Why should we be interested in the Finite Element Method (FEM)? The FEM is a general numerical method for solving both ordinary and partial differential equations in science and engineering, e.g. structural mechanics, fluid dynamics, electromagnetic, elasticity, etc. This set of lecture notes seek to build the basic foundation for understanding how to setup a discrete set of coupled equations to solve such differential equations numerically. Focus will be on a few prototype differential equations, e.g. the poisson problem which appear in steady-state thermodynamics for heat problems, which is useful to demonstrate the practical discretization and solution procedure. Generally speaking, the FEM is applied to problems where the solution is not easily obtained or impossible to obtain by analytical means. The name of the method has its origin in the fundamental idea to split up the solution domain into a number of smaller sub-domains or elements. The main advantages of the FEM are the ability to solve both linear and nonlinear problems and handle solution domains with arbitrarily complex geometry. Furthermore, the FEM framework is nowadays very extensively covered and applied in science and engineering and has a very mature theoretical foundation securing the methodological framework for developing robust numerical codes.

The first use of what we today call finite element methods appeared in the 1940s and the first commercial software package appeared 1964. R. W. Clough was the first to coin the term "finite elements" in 1960. In 1981 I. Babuška and co-workers for the first time introduced the ideas of $p$-order Finite Element Methods in theoretical papers [7, 8]. In 1984 A. T. Patera [43] coined what is referred to as "Spectral element methods" where the advantage of Spectral Methods [11, 13, 14, 29, 33] is combined with the ability of the FEM to handle complex geometries. The first book covering both $p$- and $hp$-versions of the FEM was published in 1991 by B. A. Szabo and I. Babuška [54]. The interested reader may consult more recent text books such as [18, 24, 36, 45, 58] to learn more about the theory and implementation details of high-order finite element methods.

The FEM is based on a variational method of approximation of the PDE. A global domain is divided into sub-domains which we will refer to as elements in what follows. By this approach the FEM becomes very well-suited to problems with complex geometries since we only need a way of representing the solution on each element which in principle can be of any shape. On the set of elements, a set of local approximations is constructed which can be patched together to form a global solution to the problem. Typically, the local approximations is based on polynomials and the order of these polynomials directly affects the formal accuracy of the method. The use of polynomials is a typical choice in FEM because they are straightforward to differentiate and integrate which is convenient for computer implementations.

In the classical FEM linear polynomials are employed and usually referred to as the $h$-version of the FEM because convergence can only be achieved by increasing the number of elements since the polynomial order is fixed on each element. In the $p$-version the number of elements is kept fixed and instead convergence is achieved through increasing the polynomial order of the elements. Combining the $h$- and the $p$-version leads to the so-called $hp$-FEM version where convergence can be achieved by combining the two strategies. The latter type of FEM discretization is also referred to as Spectral/$hp$ Element Methods when the element interpolation nodes are placed at the zeros of an appropriate family of orthogonal polynomials, since it is possible to employ high order polynomial basis functions and thereby fast exponential or spectral convergence in a multi-domain

setting can be achieved for smooth problems.

The FEM methods presented in this work, can be characterized as nodal methods where the solution can be interpreted directly according to the physics of the problem and the unknowns of the discrete problem is associated with a discrete representation of the domain referred to as a mesh. This is in contrast to the modal Galerkin methods where the unknowns are the modal coefficients to the local basis functions which have no direct physical interpretation.

# Chapter 1

# FEM in one space dimension

Assume we have a mathematical model which describes some physical systems in terms of a differential equation. To create a numerical model we need to make two fundamental choices

- How to satisfy the differential equation of interest?
- How to represent the solution?

By these two choices the outcome for any numerical method is a discrete finite representation of the differential equations, typically in the form of a system of coupled equations. The number of equations then corresponds to the number of unknowns in the discrete representation.

In the Finite Element Method (FEM) we divide the spatial domain into a number of so-called elements (or subdomains) which are non-overlapping. Thus, for the representation of the solution we are also faced with the task of

- How to generate a mesh?

On each of the elements which make up a finite element mesh, we seek to represent the solution using some polynomial expansion of arbitrary order within each element. Since the order is arbitrary and we prefer a computational environment where the setup is made generic, we need to determine

- How to compute polynomial expansions?
- How to evaluate integrals and derivatives?

in order to be able to develop formulation based on an element subdivision of the domain of interest.

To satisfy a differential equation using FEM, it is typical to use a Galerkin method. The Galerkin method can be considered a special case of the Method of Weighted Residuals (MWR) coined by Crandall [16] and surveys of the method can for example be found in [13, 22]. In short, the MWR is a fundamental technique to determine how to satisfy (or minimize the residual or truncation errors of) problems defined in terms of a set of governing differential equations when we seek to determine an approximate solution hereof.

With these choices and questions in mind we embark on the road toward understanding the basics of FEM together with focus on implementation details. We will start out by focussing on one-dimensional problems.

We shall see that the standard FEM discretization of model problems consists of a series of steps

1. Represent domain of interest with a Finite Element Mesh (mesh generation problem).

2. Choose appropriate basis functions to represent functions on the mesh.

3. Establish a weak formulation to be used as mathematical basis for constructing Finite Element approximations to the solution of boundary value or initial value problems.

4. For each element in mesh, determine local elemental contributions to global integrals in the weak formulation.

5. From local elemental contributions a procedure of global assembly makes it possible to compute systems of algebraic equations.

6. Modify the system of algebraic equation to incorporate and impose appropriate boundary conditions in correspondence with the model problem.

7. Compute Finite Element Solution by solving the system of algebraic equations.

8. Post process computed results.

## 1.1 Finite Element Mesh

Consider an interval $0 \leq x \leq L$ together with a set of points

$$0 = x_1 < x_2 < \cdots < x_M = L$$

For each of the $M$ points let $e_i$ denote the subinterval $x_i \leq x \leq x_{i+1}$. We call the points $x_1, x_2, ..., x_M$ nodes and the subintervals $e_1, e_2, ..., e_{M-1}$ elements. The nodes and elements make up a finite element mesh. We denote the length of each element $e_i$ by $h_i$ and determine it as

$$h_i = x_{i+1} - x_i, \quad i = 1, 2, ..., M-1 \tag{1.1}$$

making it possible to create elements with uniform as well as nonuniform lengths. The use of nonuniform sizes for the elements can be used adaptively to improve the overall accuracy and efficiency of computations. To completely describe the finite element mesh we can introduce two

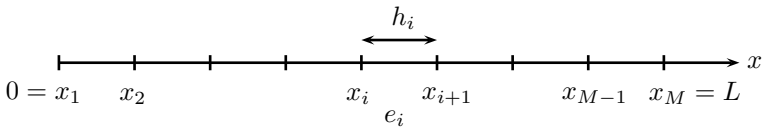



Figure 1.1: Sketch and notation for a one-dimensional mesh.

mesh data tables which can be generated by any suitable mesh generator. A data table `VX` stores the $x$-coordinates of the set of unique global vertex nodes, and an element connectivity table `EToV` stores for each element the numbers of the vertex nodes that are used to form the element.

The mesh element data tables for the set of points on the given interval can be setup as illustrated in table 1.1. The use of mesh data tables makes it possible to treat the elements of the mesh in a generic way and is a cornerstone in the practical implementation of any general FEM solver where the mesh generation procedure is considered independently of the discretization problem and setup. The strength of this approach is that once the numerical solver has been developed, different problems can subsequently be solved with minimal effort, since for every new problem the solver can be reused. Thus, with this type of setup the main user effort will be the pre- and post-processing steps where user input needs to be supplied in the initial stage of the mesh generation procedure and changing the setup parameters for the new problem.

| VX | |
|---|---|
| $i$ | $x_i$ |
| 1 | $x_1$ |
| 2 | $x_2$ |
| ⋮ | ⋮ |
| m | $x_m$ |
| ⋮ | ⋮ |
| $M$ | $x_M$ |

| EToV | | |
|---|---|---|
| $n$ | 1 | 2 |
| 1 | 1 | 2 |
| 2 | 2 | 3 |
| ⋮ | ⋮ | ⋮ |
| $m$ | $m$ | $m$+1 |
| ⋮ | ⋮ | ⋮ |
| $M$-1 | $M$-1 | $M$ |

Table 1.1: Element tables for a generic mesh in one dimension. Left: `VX`. Right: `EToV`. The first column with indices $i$ o $n$ does not need to be stored in the tables.

## 1.2 Finite Element Basis Functions

We seek to represent functions defined in the discrete topology defined by the mesh in a representation based on piecewise polynomial functions of the form

$$\hat{u}(x) = \sum_{i=1}^{M} \hat{u}_i N_i(x) \tag{1.2}$$

such that $\hat{u}(x) \approx u(x)$, where $M$ is the number of global nodes, $\hat{u}_i \cong u(x_i)$ is either a set of interpolated function values of the solution or approximate solution. The set of functions $N_i(x)$, $i = 1, ..., M$ is the set of global finite element basis functions. The global finite element basis functions are defined such that

$$N_i(x_i) = 1, \quad N_i(x_j) = 0, \quad j \neq i \tag{1.3}$$

and is therefore said to have the Cardinal property, which can also be expressed as

$$N_i(x_j) = \delta_{i,j} = \begin{cases} 0 & , i \neq j \\ 1 & , i = j \end{cases} \tag{1.4}$$

where $\delta_{i,j}$ is the Kronecker's delta. This property implies that the $i$'th global finite element basis function takes the value of unity at the $i$'th global node and zero at all other nodes. This implies that the coefficients $u_i$ in (1.2) corresponds to the discrete function values at specific points of the domain. The word "global" stresses that each function $N_i(x)$ is defined on the entire interval $x \in [0, L]$ even though each function can be made to have local support only, e.g. by construction such that it is nonzero on at most two adjacent elements.

Each of the global finite element basis functions can be represented in terms of the local basis functions which are defined on each of the elements. If the local basis functions are linear functions, i.e. polynomials of order one, then each of the global basis functions $N_i(x)$ is continuous on the solution domain $x \in [0, L]$ and linear on each element. For a one-dimensional mesh with $M = 3$ nodes, this is illustrated in Figure 1.2 for a nonuniform mesh consisting of two elements. For general meshes in one spatial dimension a typical global basis function is resembling a "hat" as illustrated in Figure 1.3.

In the following, we start out by defining the global basis functions analytically under the assumption that they are piecewise linear as in the classical finite element method. In chapter 3, we will generalize this approach to enable the use of higher order polynomials in the element basis.

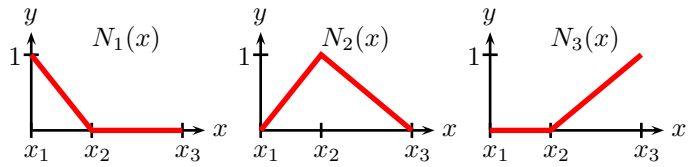



Figure 1.2: Illustration of the three global basis functions on a nonuniform mesh consisting of two elements where the local polynomial order is one for the basis functions within each element.

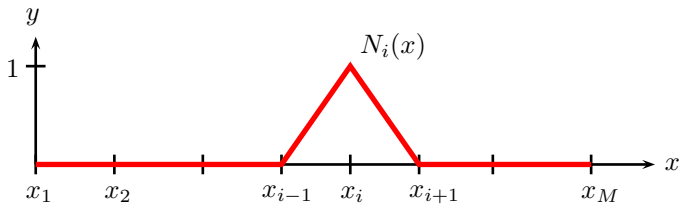



Figure 1.3: A typical global finite element basis function.

The global finite element basis functions are defined analytically as

$$
\begin{aligned}
N_1(x) &= \begin{cases} 1 - \frac{x-x_1}{h_1} & , x_1 \le x \le x_2 \\ 0 & , \text{otherwise} \end{cases} \\
N_i(x) &= \begin{cases} \frac{x-x_{i-1}}{h_{i-1}} & , x_{i-1} \le x \le x_i \\ 1 - \frac{x-x_i}{h_i} & , x_i \le x \le x_{i+1} \\ 0 & , \text{otherwise} \end{cases} \qquad , i = 2, 3, ..., M-1 \\
N_M(x) &= \begin{cases} \frac{x-x_{M-1}}{h_{M-1}} & , x_{M-1} \le x \le x_M \\ 0 & , \text{otherwise} \end{cases}
\end{aligned}
\tag{1.5}
$$

Since $N_i(x)$ is nonzero only on elements $e_{i-1}$ and $e_i$, it follows that on each element $e_i$ the only two nonzero functions are $N_i(x)$ and $N_{i+1}(x)$ as illustrated in Figure 1.4.

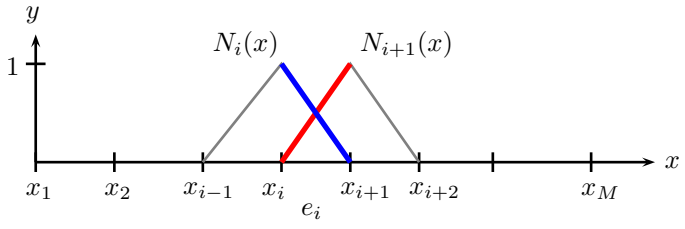



Figure 1.4: Global finite element basis functions which are nonzero over the $i$'th element.

This leads us to define two local finite element basis functions for element $e_i$ as

$$\begin{aligned} N_1^{(i)}(x) &= N_i(x) = 1 - \frac{x - x_i}{h_i}, \quad x_i \leq x \leq x_{i+1} \\ N_2^{(i)}(x) &= N_{i+1}(x) = \frac{x - x_i}{h_i}, \quad x_i \leq x \leq x_{i+1} \end{aligned} \tag{1.6}$$

The word "local" stresses that each function is defined only on a single element. We will see in section 1.6 that for the implementation of the finite element method for solving a differential equation we can exploit that calculations on the interval $x \in [0, L]$ can be reduced to calculations on individual elements. In this process the local basis functions are of central importance. The local basis functions of an element is illustrated in Figure 1.5.

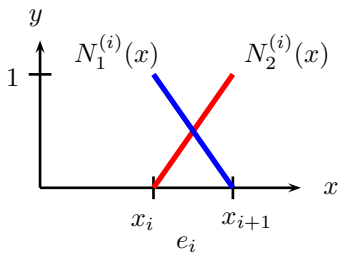



Figure 1.5: The local nodal finite element basis functions when the local polynomial order is one.

## 1.3 Interpolation

Finite element basis functions are easily used for linear interpolation. Let there be given a function $u(x)$ defined on the interval $0 \leq x \leq L$ and construct an interpolating function using the global finite element functions as

$$u_I(x) = \sum_{i=1}^{M} u(x_i) N_i(x), \quad 0 \leq x \leq L \tag{1.7}$$

where $N_i(x)$, $i = 1, ..., M$ are the global finite element basis functions for a given mesh. Since each $N_i(x)$ is continuous and piecewise linear, so is the interpolating function $u_I(x)$. Further, it follows from the Cardinal property of the basis functions that

$$u_I(x_i) = u(x_i), \quad i = 1, 2, .., M \tag{1.8}$$

Thus the interpolating function $u_I(x)$ is the continuous, piecewise linear function that interpolates $u(x)$ at the nodes. Thus, the interpolating function can be used to represent a discrete representation of a function. The functions $u(x)$ and $u_I(x)$ are illustrated in Figure 1.6.

Regarding the interpolation error $\epsilon(x) = u_I(x) - u(x)$, it can be shown (e.g. see [20]) that if $u(x)$ is twice differentiable then for $i = 1, 2, ..., M - 1$ the error can be bounded from above as

$$\max_{x \in e_i} |\epsilon(x)| \leq \frac{1}{8} h_i^2 \max_{x \in e_i} |u''(x)| \tag{1.9}$$

where $u''$ corresponds to the second derivative with respect to the dependent variable, i.e. $u'' = \frac{d^2 u}{dx^2}$. These are "local" error bounds. They lead directly to the "global" bound

$$\max_{0 \leq x \leq L} |\epsilon(x)| \leq \frac{1}{8} h^2 \max_{0 \leq x \leq L} |u''(x)| \tag{1.10}$$

where the largest element length $h$ is determined as

$$h = \max_{i=1,2,...,M-1} h_i \tag{1.11}$$

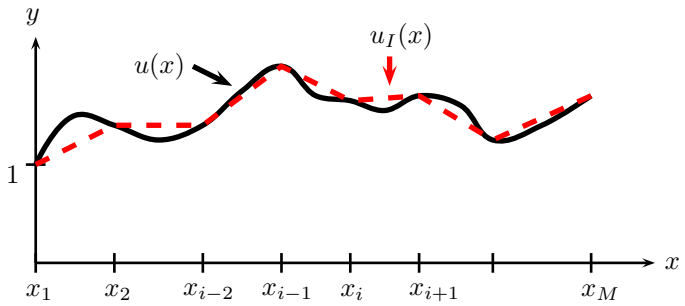



Figure 1.6: A global interpolating function $u_I(x)$ to a function $u(x)$.

Sometimes there is given a set of values $w_1, w_2, ..., w_M$, and it is of interest to construct a function $w(x)$, e.g. for providing estimates of the function values in between the nodal values, $x \in [0, L]$ such that it interpolates the values exactly

$$w(x_i) = w_i, \quad i = 1, 2, ..., M. \tag{1.12}$$

This can be achieved by taking $w(x)$ as the continuous, piecewise-linear function

$$w(x) = \sum_{i=1}^{M} w_i N_i(x), \quad 0 \leq x \leq L. \tag{1.13}$$

## 1.4 The Finite Element Method

To formulate a finite element method for solving a differential equation we need to choose how to satisfy the PDE. This choice in conjunction with the choice of representing the approximate solution to the PDE using global basis functions defines the numerical method.

It is typical to use a Galerkin formulation in the Finite Element Method, and we will illustrate the method for solving (approximately) a model boundary value problem for a second-order differential equation in one space variable.

The problem is the following: Find a function $u(x)$, $0 \leq x \leq L$ that satisfies the differential equation

$$u'' - u = 0, \quad 0 \leq x \leq L \tag{1.14}$$

together with the boundary conditions

$$u(0) = c, \quad u(L) = d \tag{1.15}$$

The exact solution of this problem can be shown to be of the form

$$u(x) = c_1 e^x + c_2 e^{-x}, \quad 0 \leq x \leq L \tag{1.16}$$

where $c_1$ and $c_2$ are real-valued constants that can be determined using the boundary conditions.

Since we will be approximating $u(x)$ by a function whose first-order derivative has jump discontinuities at the nodes, it turns out to be necessary to reformulate the problem so as to remove the second-order derivative in (1.14). This is done by multiplying both sides of (1.14) by a function $v$ that satisfies the boundary conditions

$$v(0) = v(L) = 0 \tag{1.17}$$

and integrating over the interval $0 \leq x \leq L$, and then applying integration by parts. Thus

$$\int_0^L (u'' - u)v \, dx = 0$$

or

$$\int_0^L u''v \, dx - \int_0^L uv \, dx = 0$$

or

$$u'(L)v(L) - u'(0)v(0) - \int_0^L u'v' \, dx - \int_0^L uv \, dx = 0 \tag{1.18}$$

or, using (1.17),

$$\int_0^L (u'v' + uv) \, dx = 0 \tag{1.19}$$

This establishes that the solution of problem (1.14), (1.15) is also a solution of the problem of finding a function $u(x)$ that

1. satisfies (1.19) for all sufficiently smooth functions $v$ with property (1.17),
2. satisfies (1.15).

One can also show the converse: If a function $u(x)$ is a solution of this new problem then it is also a solution of (1.14), (1.15). We call this new problem the *weak formulation* of problem (1.14), (1.15). The weak formulation is defined from the differential equations and can during the derivation take into account the boundary conditions in the expression of the surface terms that result from integration by parts.

It should be realized that the requirement "sufficiently smooth" (which in fact is a requirement on $u(x)$ as well as $v(x)$) is a vague one. How smooth, precisely, is "sufficiently smooth"? A very rough answer is that $u$ and $v$ must be smooth enough for the integral in (1.19) to make sense. A rigorous discussion of this matter lies far beyond the scope of these notes. Fortunately, it turns out that continuous, piecewise polynomials, the kind of functions used in the finite element method, are smooth enough.

We will now apply the finite element method to the weak formulation, considering first the simple mesh shown in Figure 1.2. The idea is to approximate $u(x)$ by a function of the form

$$\hat{u}(x) = \hat{u}_1 N_1(x) + \hat{u}_2 N_2(x) + \hat{u}_3 N_3(x) \tag{1.20}$$

where $N_1(x)$, $N_2(x)$ and $N_3(x)$ are the global finite element basis functions for the given mesh and $\hat{u}_1$, $\hat{u}_2$ and $\hat{u}_3$ are coefficients to be determined. Since $\hat{u}(x)$ is to be an approximation of $u(x)$, we impose the boundary conditions (1.15) on $\hat{u}(x)$ and require that

$$\hat{u}(0) = c, \quad \hat{u}(L) = d$$

From (1.20) and (1.4) we see that

$$\hat{u}(0) = \hat{u}_1 N_1(0) + \hat{u}_2 N_2(0) + \hat{u}_3 N_3(0) = \hat{u}_1$$

$$\hat{u}(L) = \hat{u}_1 N_1(L) + \hat{u}_2 N_2(L) + \hat{u}_3 N_3(L) = \hat{u}_3$$

Consequently, $\hat{u}(x)$ satisfies the given boundary conditions when we put

$$\hat{u}_1 = c, \quad \hat{u}_3 = d \tag{1.21}$$

Since $\hat{u}_1$ and $\hat{u}_3$ are now known, it remains only to determine $\hat{u}_2$. We do this by substituting $\hat{u}(x)$ for $u(x)$ in (1.19) and putting $v(x)$ equal to $N_2(x)$. Note that $N_2(x)$ vanishes at the endpoints of the interval, as required by (1.17). Thus we have

$$\int_0^L (\hat{u}' N_2' + \hat{u} N_2)\, dx = 0$$

or, using (1.20) and collecting terms,

$$\hat{u}_1 \int_0^L (N_1' N_2' + N_1 N_2)\, dx + \hat{u}_2 \int_0^L (N_2' N_2' + N_2 N_2)\, dx + \hat{u}_3 \int_0^L (N_3' N_2' + N_3 N_2)\, dx = 0$$

or, switching the order of the factors in the terms of the integrands,

$$\hat{u}_1 \int_0^L (N_2' N_1' + N_2 N_1)\, dx + \hat{u}_2 \int_0^L (N_2' N_2' + N_2 N_2)\, dx + \hat{u}_3 \int_0^L (N_2' N_3' + N_2 N_3)\, dx = 0$$

Finally, we write this in the form

$$a_{2,1}\hat{u}_1 + a_{2,2}\hat{u}_2 + a_{2,3}\hat{u}_3 = 0 \tag{1.22}$$

where

$$a_{2,1} = \int_0^L (N_2' N_1' + N_2 N_1)\, dx\,, \quad a_{2,2} = \int_0^L (N_2' N_2' + N_2 N_2)\, dx$$

$$a_{2,3} = \int_0^L (N_2' N_3' + N_2 N_3)\, dx$$

After computing these coefficients and using (1.21), we can solve (1.22) for $\hat{u}_2$.

We consider now the case of a general mesh on $x \in [0, L]$, as illustrated in Figure 1.1. Here $\hat{u}(x)$ has the form

$$\hat{u}(x) = \sum_{j=1}^{M} \hat{u}_j N_j(x) \tag{1.23}$$

As before, we impose the conditions

$$\hat{u}_1 = c, \quad \hat{u}_M = d \tag{1.24}$$

to make $\hat{u}(0) = c$ and $\hat{u}(L) = d$. To find the remaining $(M-2)$ coefficients, we substitute (1.23) for $u(x)$ in (1.19) and put $v(x)$ equal to, successively, $N_2(x)$, $N_3(x)$,...,$N_{M-1}(x)$. Note that each of these functions satisfies requirement (1.17). The result of this procedure, after simplification, may be expressed as

$$\sum_{j=1}^{M} a_{i,j}\hat{u}_j = 0\,, \qquad i = 2, 3, ..., M-1$$

where

$$a_{i,j} = \int_0^L (N_i' N_j' + N_i N_j)\, dx$$

Since $N_i(x)$ and $N_j(x)$ 'overlap' only for $j = i-1, i, i+1$, this system of equations reduces to

$$a_{i,i-1}\hat{u}_{i-1} + a_{i,i}\hat{u}_i + a_{i,i+1}\hat{u}_{i+1} = 0\,, \qquad i = 2, 3, ..., M-1 \tag{1.25}$$

where

$$a_{i,i-1} = \int_{x_{i-1}}^{x_i} (N_i' N_{i-1}' + N_i N_{i-1})\, dx$$

$$a_{i,i} = \int_{x_{i-1}}^{x_i} ((N_i')^2 + N_i^2)\, dx + \int_{x_i}^{x_{i+1}} ((N_i')^2 + N_i^2)\, dx$$

$$a_{i,i+1} = \int_{x_i}^{x_{i+1}} (N_i' N_{i+1}' + N_i N_{i+1})\, dx$$

This is a system of $(M-2)$ linear algebraic equations in the $(M-2)$ unknowns $\hat{u}_2$, $\hat{u}_3$,...,$\hat{u}_{M-1}$. It can be noticed that we obtain (1.22) when $M = 3$.

## 1.5 Element Matrices

We have expressed $a_{i,i-1}$, $a_{i,i}$ and $a_{i,i+1}$ above in terms of integrals over individual elements. When these integrals are rewritten in terms of the local finite element basis functions (1.6) illustrated in Figure 1.5, the result is

$$
\begin{aligned}
a_{i,i-1} &= \int_{x_{i-1}}^{x_i} [(N_2^{(i-1)})'(N_1^{(i-1)})' + N_2^{(i-1)} N_1^{(i-1)}]\, dx \\
a_{i,i} &= \int_{x_{i-1}}^{x_i} [(N_2^{(i-1)})'^{\,2} + (N_2^{(i-1)})^2]\, dx + \int_{x_i}^{x_{i+1}} [(N_1^{(i)})'^{\,2} + (N_1^{(i)})^2] dx \\
a_{i,i+1} &= \int_{x_i}^{x_{i+1}} [(N_1^{(i)})'(N_2^{(i)})' + N_1^{(i)} N_2^{(i)}] dx
\end{aligned}
$$

It is convenient to associate with each element $e_i$ an *element matrix* defined by

$$
\mathcal{K}^{(i)} = \begin{bmatrix} k_{1,1}^{(i)} & k_{1,2}^{(i)} \\ \\ k_{2,1}^{(i)} & k_{2,2}^{(i)} \end{bmatrix}
$$

where

$$
k_{r,s}^{(i)} = \int_{x_i}^{x_{i+1}} [(N_r^{(i)})'(N_s^{(i)})' + N_r^{(i)} N_s^{(i)}]\, dx\,, \qquad r, s = 1, 2 \tag{1.26}
$$

It is possible to derive from the analytical expressions for $N_r^{(i)}$ and $N_s^{(i)}$ given in (1.6) the result

$$
\mathcal{K}^{(\mathbf{i})} = \begin{bmatrix} (\frac{1}{h_i} + \frac{h_i}{3}) & (-\frac{1}{h_i} + \frac{h_i}{6}) \\ \\ (-\frac{1}{h_i} + \frac{h_i}{6}) & (\frac{1}{h_i} + \frac{h_i}{3}) \end{bmatrix} \tag{1.27}
$$

Observing that

$$
a_{i,i-1} = k_{2,1}^{(i-1)}, \quad a_{i,i} = k_{2,2}^{(i-1)} + k_{1,1}^{(i)}, \quad a_{i,i+1} = k_{1,2}^{(i)} \tag{1.28}
$$

we can substitute these relations in (1.25). When $M = 3$ we obtain (1.22) in the form

$$
k_{2,1}^{(1)} \hat{u}_1 + (k_{2,2}^{(1)} + k_{1,1}^{(2)}) \hat{u}_2 + k_{1,2}^{(2)} \hat{u}_3 = 0 \tag{1.29}
$$

In the general case of (1.25) it is convenient to express the system in matrix-vector form with one equation for each unknown coefficient, as illustrated for $M = 6$ by

$$
\begin{bmatrix}
k_{2,1}^{(1)} & (k_{2,2}^{(1)} + k_{1,1}^{(2)}) & k_{1,2}^{(2)} & & & \\
k_{2,1}^{(2)} & (k_{2,2}^{(2)} + k_{1,1}^{(3)}) & k_{1,2}^{(3)} & & & \\
 & k_{2,1}^{(3)} & (k_{2,2}^{(3)} + k_{1,1}^{(4)}) & k_{1,2}^{(4)} & & \\
 & & k_{2,1}^{(4)} & (k_{2,2}^{(4)} + k_{1,1}^{(5)}) & k_{1,2}^{(5)} &
\end{bmatrix}
\begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \\ \hat{u}_5 \\ \hat{u}_6 \end{bmatrix}
= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
$$

Since $\hat{u}_1 = c$ and $\hat{u}_6 = d$, the terms $k_{2,1}^{(1)} \hat{u}_1$ and $k_{1,2}^{(5)} \hat{u}_6$ in the first and last rows, respectively, of this system are known and can be moved to the right-hand side. This yields

$$
\begin{bmatrix}
(k_{2,2}^{(1)} + k_{1,1}^{(2)}) & k_{1,2}^{(2)} & & \\
k_{2,1}^{(2)} & (k_{2,2}^{(2)} + k_{1,1}^{(3)}) & k_{1,2}^{(3)} & \\
 & k_{2,1}^{(3)} & (k_{2,2}^{(3)} + k_{1,1}^{(4)}) & k_{1,2}^{(4)} \\
 & & k_{2,1}^{(4)} & (k_{2,2}^{(4)} + k_{1,1}^{(5)})
\end{bmatrix}
\begin{bmatrix} \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \\ \hat{u}_5 \end{bmatrix}
= \begin{bmatrix} -k_{2,1}^{(1)} c \\ 0 \\ 0 \\ -k_{1,2}^{(5)} d \end{bmatrix}
$$

Note the coefficient matrix is now square. By a trivial extension of this system we can include $\hat{u}_1$ and $\hat{u}_6$ in the vector of unknowns as follows:

$$\begin{bmatrix} 1 & & & & & \\ & (k_{2,2}^{(1)} + k_{1,1}^{(2)}) & k_{1,2}^{(2)} & & & \\ & k_{2,1}^{(2)} & (k_{2,2}^{(2)} + k_{1,1}^{(3)}) & k_{1,2}^{(3)} & & \\ & & k_{2,1}^{(3)} & (k_{2,2}^{(3)} + k_{1,1}^{(4)}) & k_{1,2}^{(4)} & \\ & & & k_{2,1}^{(4)} & (k_{2,2}^{(4)} + k_{1,1}^{(5)}) & \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \\ \hat{u}_5 \\ \hat{u}_6 \end{bmatrix} = \begin{bmatrix} c \\ -k_{2,1}^{(1)}\, c \\ 0 \\ 0 \\ -k_{1,2}^{(5)}\, d \\ d \end{bmatrix}$$

We henceforth denote this $M$'th-order system (for any $M \geq 3$) by

$$\mathbf{A}\hat{\mathbf{u}} = \mathbf{b} \tag{1.30}$$

The coefficient matrix $\mathbf{A}$ is seen to be tridiagonal and symmetric (since by (1.28) $k_{i,j}^{(n)} = k_{j,i}^{(n)}$) for the model problem in one space dimension we have considered. One can show it is also positive definite; i.e., the eigenvalues of $\mathbf{A}$ are positive. Knowledge about such properties makes it possible to employ efficient solvers for solving the system.

## 1.6 The Computation

The computational tasks are those of computing $\mathbf{A}$ and $\mathbf{b}$ and of solving (1.30). Algorithms 1 and 2 below deal with the computation of $\mathbf{A}$ and $\mathbf{b}$. The process of constructing the system coefficient matrix and right hand side vector is known as "global assembly". Since $\mathbf{A}$ is symmetric, it suffices to compute the entries on the main diagonal ($a_{i,i}$, $i = 1, 2, ..., M$) and those on the upper bidiagonal ($a_{i,i+1}$, $i = 1, 2, ..., M-1$). The algorithms are written in a pseudo-programming language which makes use of an $M \times M$ array, $\mathbf{A}$, and an $M \times 1$ array, $\mathbf{b}$, for the storage of $\mathbf{A}$ and $\mathbf{b}$, respectively. The choice of an $M \times M$ array for $\mathbf{A}$ is based on considerations of clarity alone; it makes it easier to show what needs to be computed. In a practical implementation a sparse data structure should always be used for $\mathbf{A}$.

**Algorithm 1:** Global assembly of upper triangular part of coefficient matrix $\mathbf{A}$ (1D).

Allocate storage for $\mathbf{A}$ and $\mathbf{b}$
**for** $i := 1$ **to** $M - 1$
  Compute $k_{1,1}^{(i)}$, $k_{1,2}^{(i)}$ and $k_{2,2}^{(i)}$ from (1.27).
  $a[i,i] := a[i,i] + k_{1,1}^{(i)}$
  $a[i,i+1] := k_{1,2}^{(i)}$
  $a[i+1,i+1] := k_{2,2}^{(i)}$

This computation has ignored the boundary conditions and the system for the case $M = 3$ takes the preliminary form

$$\begin{bmatrix} k_{1,1}^{(1)} & k_{1,2}^{(1)} & \\ k_{2,1}^{(1)} & (k_{2,2}^{(1)} + k_{1,1}^{(2)}) & k_{1,2}^{(2)} \\ & k_{2,1}^{(2)} & k_{2,2}^{(2)} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{1.31}$$

Comparing the result of Algorithm 1 with $\mathbf{A}$ and $\mathbf{b}$ as illustrated immediately before (1.30) for a case with $M = 6$, one sees that the corrections done as a part of Algorithm 2 are needed.

**Algorithm 2:** Imposing boundary conditions by modification of system (1D).

$b[1] := c, \quad b[2] := b[2] - a[1,2] * c, \quad a[1,1] := 1, \quad a[1,2] := 0$
$b[M] := d, \quad b[M-1] := b[M-1] - a[M-1,M] * d$
$a[M,M] := 1, \quad a[M-1,M] := 0$

Note that Algorithm 2 exploits symmetry in the element matrix in computing $b_2$:

$$b[2] := b[2] - a[1,2] * c \;=\; b[2] - k_{1,2}^{(1)} * c \;=\; b[2] - k_{2,1}^{(1)} * c$$

Note, too, that if the computation of $b[M-1]$ were changed to

$$b[M-1] := -a[M-1,M] * d$$

then the result would be incorrect in the case $M = 3$, where (1.30) reduces to

$$\begin{bmatrix} 1 & & \\ & (k_{2,2}^{(1)} + k_{1,1}^{(2)}) & \\ & & 1 \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \end{bmatrix} = \begin{bmatrix} c \\ -k_{2,1}^{(1)} c - k_{1,2}^{(2)} d \\ d \end{bmatrix} \tag{1.32}$$

As mentioned above, an $M \times M$ array is an impractical data structure for $\mathbf{A}$ since only $(2M-1)$ of the $M^2$ entries of this matrix are needed. The recommended procedure is either to store the nonzero elements appearing on the main diagonal and upper bidiagonal of $\mathbf{A}$ in two single-indexed arrays of length $M$ and $M-1$, respectively, or to store these two diagonals as columns in an $M \times 2$ array. However, in Matlab it is possible to make use of the compressed row storage format which is supported in Matlab by just allocating the necessary sparse storage using the `spalloc` command. Note that to use the sparse capabilities of Matlab one has to be careful not to do significant manipulations of the matrix structure and nonzero elements, since this can result in significant overhead due to the involved data movements in the memory space.

We consider now the problem of solving (1.30), given $\mathbf{A}$ and $\mathbf{b}$. Since $\mathbf{A}$ is symmetric positive definite and tridiagonal, the best numerical algorithms are Gaussian elimination without pivoting and the Cholesky method, e.g. see [25]. The computational work in both cases is $\mathcal{O}(M)$, but it should be noted that the Cholesky method requires about half the storage and half of the computational cost of the LU factorization resulting from Gaussian elimination.

Finally, rather than constructing and solving the $M$'th-order system (1.30), one could easily work directly with a system of reduced order $(M-2)$ where the known coefficients are eliminated from the system of equations. For example, compare the systems of order four and six immediately before (1.30). Our main reason for not doing this is to prepare the reader for solving problems in two space dimensions, where the larger system without any elimination of the equations for the coefficients is much the simpler to deal with.

## 1.7 Error Bounds

Let $u(x)$ be the solution of problem (1.14), (1.15), and let $\hat{u}(x)$ be given by

$$\hat{u}(x) = \sum_{i=1}^{M} \hat{u}_i N_i(x)$$

where the coefficients $\hat{u}_1$, $\hat{u}_1$,...,$\hat{u}_M$ are determined by (1.30). It can be shown that there exists a constant $C$ such that

$$\max_{0 \le x \le L} |\hat{u}(x) - u(x)| \le Ch^2 \tag{1.33}$$

where $h$ is the largest element length in the mesh, cf. (1.11).

The function $\hat{u}(x)$ is in general different from the interpolating function $u_I(x)$ in (1.7) which interpolates $u(x)$ at the nodes. However, (1.10) and (1.33) show that both functions are second-order approximations to $u(x)$. It should be mentioned that the interpolating property

$$|u_I(x_i) - u(x_i)| = 0, \qquad i = 1, 2, ..., M$$

which is valid for specific nodes, does *not* imply

$$\max_{0 \le x \le L} |u_I(x) - u(x)| \le \max_{0 \le x \le L} |\hat{u}(x) - u(x)|$$

That is, we need to take into account that the largest point-wise errors is not guaranteed to be found on a node point. It might as well be in between nodes.

There are a number of ways of measuring the error function $(\hat{u}(x) - u(x))$. One is to consider the so-called *infinity norm*

$$||\hat{u}(x) - u(x)||_\infty \equiv \max_{0 \le x \le L} |\hat{u}(x) - u(x)|$$

which is just what we have done in (1.33) above. Another is to look at the *energy norm*

$$||\hat{u}(x) - u(x)||_E \equiv \sqrt{\int_0^L [(\hat{u}(x) - u(x))^2 + (\hat{u}'(x) - u'(x))^2]\, dx}$$

which is associated specifically with the weak formulation (1.19) of the differential equation (1.14).

It turns out that the function $\hat{u}(x)$ produced by the finite element method *minimizes* the energy norm of the error. More precisely, if $w(x)$ is a function of the form

$$w(x) = c\, N_1(x) + d\, N_M(x) + \sum_{i=2}^{M-1} c_i N_i(x)$$

where $c_2$, $c_3$,..., $c_{M-1}$ are arbitrary, then

$$||\hat{u}(x) - u(x)||_E \le ||w(x) - u(x)||_E$$

Note that this implies

$$||\hat{u}(x) - u(x)||_E \le ||u_I(x) - u(x)||_E$$

For further discussion of errors see for example [53] and [4].

# Exercises

## Exercise 1.1

The goal of this exercise is to derive analytical expressions for defining a FEM method to solve a boundary value problem and solve using a discrete set of mesh nodes defining both uniform and nonuniform meshes .

Consider the FEM applied to the boundary value problem for a second-order differential equation

$$u'' - u = 0, \quad 0 \leq x \leq L$$

with the following boundary conditions

$$u(0) = c, \quad u(L) = d$$

a) Derive the exact expression for the element matrix $K_i$ for the $i$'the element using the variable transformation $y = (x - x_i)/h_i$. Assume that linear polynomials are employed on each element.

b) Let $L = 2$, $c = 1$ and $d = e^2$. Let $M = 3$, i.e. the interval is divided into only two elements, $e_1$ and $e_2$, of lengths $h_1$ and $h_2$, respectively. Solve for the unknown $\hat{u}_i$ in the following cases: i) uniform mesh with $h_1 = h_2$ and ii) nonuniform mesh with $h_1 = 2h_2$.

c) Consider the case with a uniform mesh with mesh sizes $h_1 = h_2$. The solution to the FEM equations determines a continuous, piecewise linear function $\hat{u}(x)$ that approximates the exact solution $u(x)$ to the problem. Plot $\hat{u}(x)$ and $u(x)$. In the same figure, include a plot of the interpolating function $u_I(x)$ that interpolates $u(x)$ at the nodes $x_i$, $i = 1, 2, ..., M$.

d) Repeat c) for the nonuniform case ii) in b).

e) Explain the difference between $u_I(x)$ and $\hat{u}(x)$. (HINT: How do we represent and define the two functions on the discrete domain? What distinguishes these definitions from each other?)

f) Describe the basic steps that are needed to generate a set of FEM equations to the boundary value problem.

## Exercise 1.2

The goal of this exercise is to write your first FEM procedure for solving a boundary value problem in one dimension. This requires that some of the algorithms of Chapter 1 is implemented.

a) Write a Matlab program based on the skeleton code on the next page that solves the boundary value problem given in Exercise 1.1 using the classical FEM where the local solution is represented using linear polynomials. (HINT: for better reuse of code and for easy testing it is recommended to write your own code pieces as Matlab functions whenever it is appropriate.)

**Head:**
`function [u] = BVP1D(L,c,d,x)`

**Test case:**
Input: `L=2, c=1, d=exp(2),`
`x = [0.0, 0.2, 0.4, 0.6, 0.7, 0.9, 1.4, 1.5, 1.8, 1.9, 2.0]`

b) Make appropriate changes to the function `BVP1D` so that the number of nodes $M$ can replace the array `x` as input in the argument list. If $M$ is used in the call, then `x` should be computed such that the nodes are equidistant within the script. Note that the uniformity of the elements makes it possible to simplify the assembly process.

**Head:**
`function [u,x] = BVP1D(L,c,d,M)`

**Test case**
Input: `L=2, c=1, d=exp(2), M=11.`

c) Validate your program by comparing with solutions of Exercise 1.1.

d) Compare the results computed using the function with (1.33). What is the convergence rate of the solution when the mesh size decreases, i.e. $h \to 0$? (Hint: determine an estimate for $p$ valid in the asymptotic limit $h \to 0$ assuming that the convergence rate is $\mathcal{O}(h^p)$.)

### Skeleton of Matlab program code

```
function [u] = BVP1D(L,c,d,x)
% Purpose: Solve second-order boundary value problem using FEM.
% Author(s): <YOUR NAMES HERE>

%% INPUT PARAMETERS
%     L : Domain length
%     c : Left boundary condition
%     d : Right boundary condition
%     x : 1D mesh vector x(1:{M})

%% GLOBAL ASSEMBLY
% Assemble A (the upper triangle only) and b. (Algorithm 1)

<INSERT YOUR CODE HERE>

%% IMPOSE BOUNDARY CONDITIONS
% (Algorithm 2)

<INSERT YOUR CODE HERE>

%% SOLVE SYSTEM
% Solve using the Cholesky factorization of A to solve A*u=b
[U,flag] = chol(A);
if flag == 0
  u = U \ (U' \ b);
else
  disp('A is not positive definite'), return
end

%% OUTPUT
% Visualize solution and output solution

<INSERT YOUR CODE HERE>
```

## Exercise 1.3

The goal of this exercise is develop a procedure for adaptively finding an optimal mesh node distribution which both minimizes the global error and has a uniform error distribution.

First, we will consider how to determine the largest error on an element when we assume that the exact solution is $u(x) = e^x$. With this in mind, we derive expressions for determining the maximum error within an element.

Let $v(x)$ be an arbitrary continuous, piecewise linear function with respect to the nodes

$$0 = x_1 < x_2 < x_3 < ... < x_M = 2$$

and let

$$v_i = v(x_i), \quad i = 1, 2, ..., M$$

If we regard $v(x)$ as an approximation to $u(x)$, then the corresponding error function is

$$F(x) = v(x) - u(x), \quad 0 \leq x \leq 2$$

It is convenient to describe the error with a single number, and we therefore define

$$E = \max_{0 \leq x \leq 2} |F(x)|$$

Consider now the computation of $E$. Since

$$E = \max\{E_1, E_2, ..., E_{M-1}\}$$

where

$$E_i = \max_{x_i \leq x \leq x_{i+1}} |F(x)|, \quad i = 1, 2, ..., M-1$$

we can turn our attention to the problem of computing $E_i$. Let's assume that the exact solution is $u(x) = e^x$. Because $v(x)$ is linear on element $e_i$, it is found that

$$F(x) = v(x) - u(x) = A_i x + B_i - e^x, \quad x_i \leq x \leq x_{i+1}$$

where

$$A_i = \frac{v_{i+1} - v_i}{x_{i+1} - x_i}, \quad B_i = v_i - A_i x_i$$

It is well known that a function that is monotonic on a closed finite interval attains its greatest absolute value at one (or both) of the end points. Applying this to the error function $F(x)$, we find the following:

**Case 1:** $v_{i+1} \leq v_i$**.**

Here $v(x)$ is constant or monotonically decreasing while $e^x$ is monotonically increasing. Hence $F(x)$ is monotonically decreasing, and

$$E_i = \max\{|F(x_i)|, |F(x_{i+1})|\}$$

**Case 2:** $v_{i+1} > v_i$ **and** $\xi_i \in [x_i, x_{i+1}]$**, where** $\xi_i = \ln(A_i)$**.**

$F(x)$ has a relative extremum at $x = \xi_i$ (i.e., $F'(\xi_i) = 0$) and is monotonic on each of the subintervals $[x_i, \xi_i]$ and $[\xi_i, x_{i+1}]$. Hence

$$E_i = \max\{|F(x_i)|, |F(x_{i+1})|, |F(\xi_i)|\}$$

**Case 3:** $v_{i+1} > v_i$ **and** $\xi_i \notin [x_i, x_{i+1}]$**, where** $\xi_i = \ln(A_i)$**.**

Here $F(x)$ is monotonic on $[x_i, x_{i+1}]$ (because there is no extremum in this interval), so

$$E_i = \max\{|F(x_i)|, |F(x_{i+1})|\}$$

From the above we obtain a simple algorithm for computing $E_i$:

**Algorithm 3:** Compute error on an element (specific to exercise 1.3).

Compute $A_i$ and $B_i$
$E_i := \max\ \{\ |F(x_i)|, |F(x_{i+1})|\ \}$
**if** $A_i > 0$
    Compute $\xi_i$
    **if** $\xi \in [x_i, x_{i+1}]$
        $E_i := \max\{E_i, |F(\xi_i)|\}$

a) The purpose of this question it to investigate the error function $e(x) = \hat{u} - u(x)$, where $\hat{u}(x)$ is the FEM solution of the boundary values problem from Exercise 1.1 and $u(x)$ is the exact solution. Write a Matlab function based on Algorithm 3 that computes the element errors `Eelm(1:M-1)` and global error `E` from `x(1:M)` and `v(1:M)`.

**Head:**
```
function [Eelm,E] = errelm(x,v)
```

**Test case:**
$L = 2, c = 1, d = e^2$ which makes the exact solution $u(x) = e^x$, $0 \le x \le 2$.

b) Let $v(x)$ be the finite element solution computed in Exercise 1.2 b). Compute the local element errors $E_i$ and the global error $E$ using `errelm`.

c) Let $M = 11$. Define a simple algorithm to find the interior node coordinates $x_2, x_3, ..., x_{M-1}$ that minimize $E$ to have a desired accuracy in the interval $E - \min_{1 \le i \le M-1} E_i \le 10^{-6}$. It may be assumed that

i) The optimal set of nodes is characterised by the property $E_1 = E_2 = ... = E_{M-1}$.

ii) $E_i \cong C_i h_i^2$, where $C_i$ is independent of $h_i$.

(HINT: How can we update the individual node position using the local error estimates for each element given in ii)?).

## Exercise 1.4

a) Repeat Exercise 1.3 b) and c) where now $v(x) = u_I(x)$ the finite element interpolant of $u(x)$. Hence $v_i = e^{x_i}$, $i = 1, 2, ..., M$. In part c), the computed results should be compared with the error estimate (1.10).

b) Compare the results of this exercise with those of Exercise 1.3.

## Exercise 1.5

The goal of this exercise is go through the fundamental steps of formulating and implementing a Finite Element Method for a linear advection-diffusion equation. We consider the following model boundary value problem for the linear advection-diffusion equation

$$\begin{aligned} -(\epsilon u')' + (\Psi u)' &= f, \qquad x \in ]0,1[ \\ u(0) = u(1) &= 0. \end{aligned} \tag{1.34}$$

For simplicity we assume that the coefficients $\epsilon > 0$ and $\Psi$ are real-valued constants on $]0,1[$.

a) Assume that $u$ is a twice continuously differentiable function solving (1.34). Verify that $u$ also solves the variational problem

$$a(u,v) = \ell(v), \tag{1.35}$$

for every continuous piecewise-smooth function $v$, such that $v(0) = v(1) = 0$, where $a(u,v) = \int_0^1 \epsilon u'v' \mathrm{d}x - \int_0^1 \Psi uv' \mathrm{d}x$ and $\ell(v) = \int_0^1 fv \mathrm{d}x$.

b) Follow the procedure outlined in Sections 1.4-1.5 of the course notes and arrive at the system of linear algebraic equations $\mathbf{A}\hat{\mathbf{u}} = \mathbf{b}$, corresponding to the finite element discretization of variational problem (1.35). Write down the expressions for the elemental stiffness matrices and right hand sides. Is the resulting matrix $\mathbf{A}$ symmetric?

c) If the matrix $\mathbf{A}$ is *positive definite*, that is, $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} > 0$ for all non-zero vectors $\mathbf{v}$, then it is necessarily non-singular and therefore the linear algebraic system $\mathbf{A}\hat{\mathbf{u}} = \mathbf{b}$ admits a unique solution for every right-hand side $\mathbf{b}$.

Show that $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} > 0$ for every vector $\mathbf{v}$, such that $\mathbf{v}_1 = \mathbf{v}_M = 0$ and thus the linear algebraic system resulting from (1.35) always admits a solution.

*Note*, that you have to show *two* things: (i) $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} \geq 0$ for every vector $\mathbf{v}$ satisfying the boundary conditions; and (ii) $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} = 0$ implies $\mathbf{v} = \mathbf{0}$.

The following hints are given: $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} = a(\sum_{i=1}^{M} v_i N_i, \sum_{j=1}^{M} v_j N_j)$; $\int vv' \mathrm{d}x = 1/2 \int (v^2)' \mathrm{d}x$.

d) Let $f = 1$ (and $\Psi \neq 0$); the analytical solution to (1.34) in this case is

$$u(x) = \frac{1}{\Psi}\left(\frac{1 + (\exp(\Psi/\epsilon) - 1)x - \exp(x\Psi/\epsilon)}{\exp(\Psi/\epsilon) - 1}\right)$$

Plot the solution for fixed $\Psi = 1$ and various values of $\epsilon \in \{\, 1, 0.01, 0.0001 \,\}$.

e) Make appropriate modifications to the program you have written for solving Exercise 1.2, so that is solves the boundary value problem (1.35) instead and verify its convergence. Use the same parameters as in step d). Comment on the variation of the quality of the computed FEM solution with $\varepsilon$.

## Exercise 1.6

In this exercise the goal is to develop an algorithm that can be used for automatic Adaptive Mesh Refinement (AMR). Such an algorithm consists of two ingredients, namely, a method for refining the mesh and a method for selecting which mesh elements to be refined. The method for refining the mesh will be based on $h$-type refinement where existing elements are subdivided into two new element of equal size to reduce errors locally. The selection of elements for refinement will be based on assuming that an optimal mesh have errors distributed equally across elements of the mesh.

In the following, we will consider how to automatically adapt a mesh to the function

$$u(x) = e^{-800(x-0.4)^2} + 0.25e^{-40(x-0.8)^2}$$

across the interval $0 \leq x \leq 1$ such that an interpolating piece-wise polynomial (1.7) achieves a user-defined accuracy level for the accurate representation of $u(x)$.

a) A simple way to guide the selection of mesh elements for refinement, is to estimate the error decrease rate that will result if the $i$'th element is subdivided in two smaller elements of equal size. We will employ a metric which measures the change in approximate solution across elements by refinement. The metric is defined in terms of the $L_2$ (mean-square) norm as

$$\Delta err_i = ||\Pi_{h/2,i}u - \Pi_{h,i}u||_{L_2(e_i)}, \quad ||f||_{L_2(\Omega)} \equiv \left(\int_\Omega |f|^2 dx\right)^{1/2}$$

$\Pi_{h,i}u$ is an operator which defines the piece-wise interpolation of $u(x)$ onto the $i$'th element with mesh size $h$ (see equation (1.7)). How can the metric be computed? Detail this. Then, write a Matlab routine which can compute the estimated error decrease rate based on this metric. The estimate should be based on the mesh tables `VX` and `EToV` (learn about such tables in beginning of Chapter 2) which defines a coarse mesh. `err` is a vector with $K$ elements such that `err(i)` is the estimated error for the $i$'th element in the coarse mesh.

**Head:**

```
function [err] = compute_error_decrease(fun,VX,EToV);
```

b) Write a Matlab routine which can subdivide a coarse mesh and produce a refined mesh by refining only elements which has been marked for refinement using an element index vector `idxMarked`.

**Head:**

```
function [EToVfine,xfine] = refine_marked(EToVcoarse,xcoarse,idxMarked)
```

c) Write a Matlab routine which invokes AMR to produce a mesh for representing $u(x)$ using a piece-wise polynomial representation such that $\Delta err_i < 10^{-4}$ starting from a mesh consisting of three equi-sized elements. Different criteria can be used for selecting elements for refinement, e.g.

$$\Delta err_i > \alpha \cdot \texttt{tol}, \quad 0 < \alpha \leq 1$$

or some kind of fraction of the element with the largest errors

$$\Delta err_i > \alpha \max_i \Delta err_i$$

d) Present plots of the computed solution, the element size distribution for the final mesh and for evaluating the performance of the algorithm that are used in c) (e.g. error vs. mesh points).

If time permits, feel free to experiment with other functions for $u(x)$, metrics for error estimation, error indication (e.g. measuring jumps at interfaces) and element selection criteria.

## Exercise 1.7

Consider the FEM applied to the following boundary value problem

$$u''(x) - u(x) = f(x), \quad 0 \leq x \leq 1, \quad u(0) = c, \quad u(1) = d$$

The goal of this exercise will be to write a Matlab program which can solve this boundary value problem using the classical FEM combined with an Adaptive Mesh Refinement (AMR) algorithm. The AMR should be based on $h$-refinement where the mesh elements are subdivided as in Exercise 1.6. This will require two basic ingredients, namely, a method for doing mesh refinements (reuse method from Exercise 1.6) and a way to estimate the errors (*a posteriori* where the computed FEM solution is used to estimate the accuracy) in order to be able to guide the refinements until a desired overall accuracy level has been achieved.

a) Determine $f(x)$, $c$ and $d$ using the Method of Manufactured solution where it is assumed that the exact solution is $u(x) = e^{-800(x-0.4)^2} + 0.25e^{-40(x-0.8)^2}$. Visualize this solution for $0 \leq x \leq 1$ and comment on expectations to an optimal mesh for the problem. It is not allowed to use the knowledge of this exact solution in the AMR procedure in this exercise.

b) Write a Matlab routine which can compute error estimates based on the $L_2$-norm on a per-element basis. The error estimate can be expressed in terms of computed solution values (`uhc` and `uhf`) obtained on a coarse (`xc`) and a refined mesh (`xf`). It is advantageous to take into account which elements that have been marked for refinement (`idxmarked`) and store information about how elements are then subdivided (`Old2New`) in appropriate arrays. If you see other ways of doing this setup feel free to do so.

**Head:** (suggestion)
```
function [err] = errorestimate(xc,xf,uhc,uhf,EToVc,EToVf,Old2New)
```

c) Write a Matlab routine which can subdivide a coarse mesh and produce a refined mesh by refining only elements which has been marked for refinement (Same as in Exercise 1.6 b)).

d) Derive the weak formulation of the boundary value problem in the usual way and identify elemental contributions to $A$ and $\mathbf{b}$ in the linear system that results from the discretization. It is allowed to assume that $f(x)$ can be represented in terms of a continuous piecewise polynomial function (see (1.2)). By this approximation the contributions to a right hand side vector $\mathbf{b}$ can be based on identifying the elemental contributions from

$$\int_0^L f(x)v(x)dx \approx \int_0^L \left( \sum_{j=1}^{M} \hat{f}_j N_j(x) \right) N_i(x)dx = \sum_{j=1}^{M} \hat{f}_j \int_0^L N_i(x)N_j(x)dx$$

e) Write a new Matlab routine by modifying `BVP1D` developed in Exercise 1.2. The new routine should solve the boundary value problem stated in this exercise with a nonzero right hand side function $f(x)$ (`func`).

**Head:**
```
function [u] = BVP1Drhs(L,c,d,x,func)
```

f) Present relevant plots of computed solutions, element size distributions for converged solutions and evaluate the performance of the FEM + AMR algorithm implemented (e.g. figures showing error vs. DOF and CPU time vs. DOF).

If time permits, feel free to experiment with other functions for $u(x)$, metrics for error estimation, error indication and element selection criteria.

# Chapter 2

# FEM in two space dimensions

The extension of the ideas for FEM in one space dimension into two dimensions is conceptually trivial, however, do require some extra effort for the implementation. The implementation and solution procedures follow the same steps as in one space dimension.

## 2.1 Finite Element Basis functions

We consider now problems in two space dimensions. Figure 2.1 shows a two-dimensional closed domain $\overline{\Omega}$ with interior $\Omega$ and boundary $\Gamma$. A finite element mesh, consisting of triangular elements with nodes at the vertices, is imposed on $\overline{\Omega}$. Note that the boundary of the mesh does not coincide

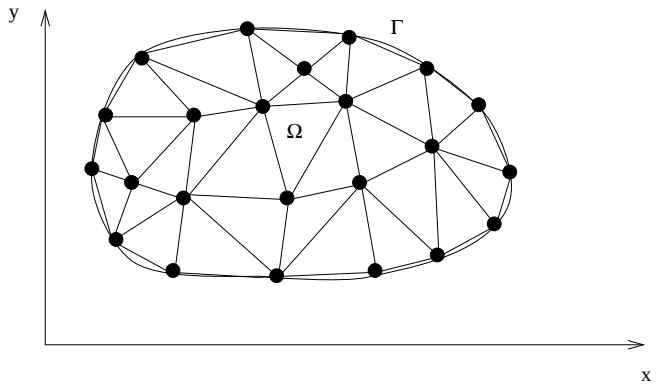



Figure 2.1: A domain with interior $\Omega$ and boundary $\Gamma$ , and a finite element mesh.

with $\Gamma$ because the latter is curved. As the mesh is refined by increasing the number of element within the domain and on it's boundary, however, the difference between the two boundaries may then be reduced.

We denote the elements of a given mesh by $e_n$ , $n = 1, 2, ..., N$ and the nodes by $(x_i, y_i)$, $i = 1, 2, ..., M$. The value $i$ is the *global node number* of node $(x_i, y_i)$. We also need to assign *local node numbers* to the nodes in any given element. The local node numbers are always 1, 2 and 3, as illustrated in Figure 2.2. (See also Figure 2.5 below).

For each node $(x_i, y_i)$ we introduce a global finite element basis function, $N_i(x, y)$, defined on $\overline{\Omega}$ as follows:

1. $N_i(x, y)$ is continuous on the entire mesh and linear on each element,

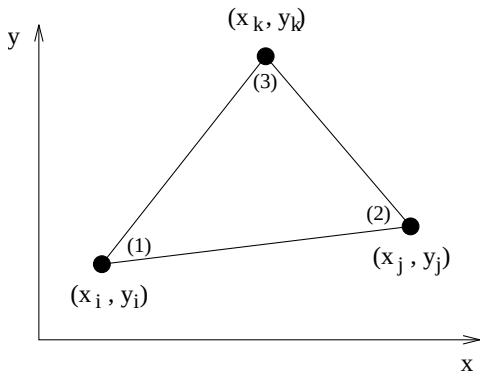



Figure 2.2: Local (1, 2, 3) and global $(i, j, k)$ node numbers in the typical element.

2. $N_i(x_i, y_i) = 1, \quad N_i(x_j, y_j) = 0, \quad j \neq i\,.$

A typical function of this type is shown in Figure 2.3. While it is defined on the whole mesh, it is nonzero only on the elements to which node $(x_i, y_i)$ belongs.

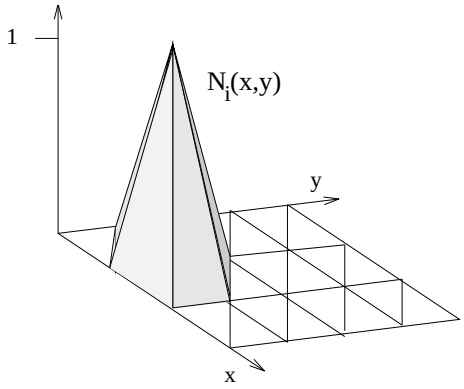



Figure 2.3: A global finite element basis function.

On a given element $e_n$ only three of these functions are nonzero. We call these the local finite element basis functions for $e_n$ and denote them $N_r^{(n)}(x, y)$, $r = 1, 2, 3$. They are defined by

$$N_1^{(n)}(x, y) = N_i(x, y), \quad (x, y) \in e_n$$
$$N_2^{(n)}(x, y) = N_j(x, y), \quad (x, y) \in e_n$$
$$N_3^{(n)}(x, y) = N_k(x, y), \quad (x, y) \in e_n$$

where we are using the correspondence between the local and global node numbers shown in Figure 2.2. An example of these local functions is given in Figure 2.4.

We now derive explicit formulas for the local basis functions. It is convenient for this purpose to use the local ordering of nodes in $e_n$ ; i.e., in this discussion the nodes will be denoted $(x_1, y_1)$, $(x_2, y_2)$ and $(x_3, y_3)$.

Consider first $N_1^{(n)}(x, y)$. Since this function is linear, it is of the form

$$N_1^{(n)}(x, y) = a_1 + b_1 x + c_1 y, \quad (x, y) \in e_n$$

Using the fact that the local basis functions can be constructed from Lagrange polynomials on the $n$'th element

$$N_1^{(n)}(x_1, y_1) = 1, \quad N_1^{(n)}(x_2, y_2) = 0, \quad N_1^{(n)}(x_3, y_3) = 0$$

we obtain the relations

$$a_1 + b_1 x_1 + c_1 y_1 = 1, \quad a_1 + b_1 x_2 + c_1 y_2 = 0, \quad a_1 + b_1 x_3 + c_1 y_3 = 0$$

This is a system of three linear equations in the three unknowns $a_1$, $b_1$ and $c_1$. Provided that the triangle $e_n$ is not degenerate (i.e., provided the three nodes do not lie on a straight line), it is easy to show that this system has the unique solution

$$a_1 = \frac{1}{2\Delta}\tilde{a}_1, \quad b_1 = \frac{1}{2\Delta}\tilde{b}_1, \quad c_1 = \frac{1}{2\Delta}\tilde{c}_1$$

where

$$\tilde{a}_1 = x_2 y_3 - x_3 y_2, \quad \tilde{b}_1 = y_2 - y_3, \quad \tilde{c}_1 = x_3 - x_2$$

and

$$\Delta = \frac{1}{2}[x_2 y_3 - y_2 x_3 - (x_1 y_3 - y_1 x_3) + x_1 y_2 - y_1 x_2] \tag{2.1}$$

The corresponding formulas for $N_2^{(n)}(x,y)$ and $N_3^{(n)}(x,y)$ can be similarly derived. We summarize the results as follows:

$$\begin{aligned} N_1^{(n)}(x,y) &= \frac{1}{2\Delta}(\tilde{a}_1 + \tilde{b}_1 x + \tilde{c}_1 y) \\ N_2^{(n)}(x,y) &= \frac{1}{2\Delta}(\tilde{a}_2 + \tilde{b}_2 x + \tilde{c}_2 y) \\ N_3^{(n)}(x,y) &= \frac{1}{2\Delta}(\tilde{a}_3 + \tilde{b}_3 x + \tilde{c}_3 y) \end{aligned} \tag{2.2}$$

where

$$\tilde{a}_i = x_j y_k - x_k y_j \ , \quad \tilde{b}_i = y_j - y_k \ , \quad \tilde{c}_i = x_k - x_j \tag{2.3}$$

for $(i,j,k) = (1,2,3), (2,3,1), (3,1,2)$. For example,

$$\tilde{c}_2 = x_1 - x_3, \quad \tilde{a}_3 = x_1 y_2 - x_2 y_1$$

The parameter $\Delta$ above is equal to the area of element $e_n$ (and is therefore positive) if the local ordering of the nodes in $e_n$ is counterclockwise. If the local ordering is clockwise then $\Delta$ is equal to minus the area.

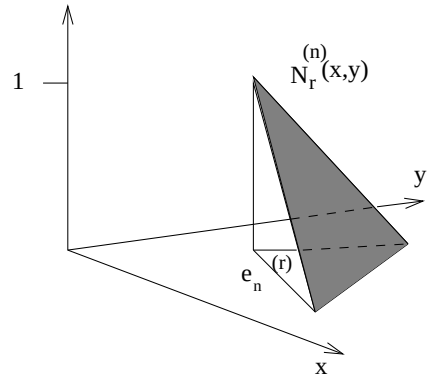



Figure 2.4: A local finite element basis function.

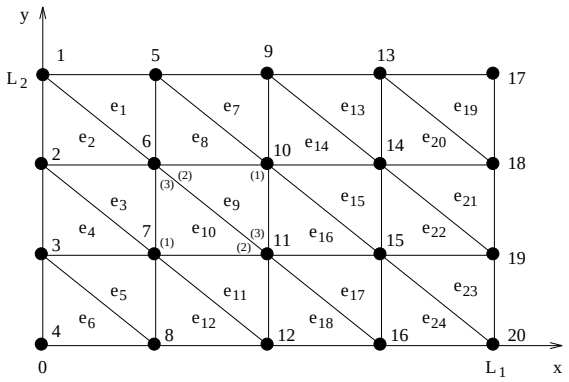



Figure 2.5: A finite element mesh on a rectangular domain.

| EtoV | | | |
|---|---|---|---|
| *n* | *1* | *2* | *3* |
| 1 | 5 | 1 | 6 |
| 2 | 2 | 6 | 1 |
| 3 | 6 | 2 | 7 |
| 4 | 3 | 7 | 2 |
| . | . | . | . |
| 24 | 16 | 20 | 15 |

Table 2.1: Element table for the mesh in Figure 2.5. The first column with indices $n$ does not need to be stored in the table.

## 2.2 A mesh on a rectangular domain

It is convenient to have a simple finite element mesh to refer to later, and for this purpose we give in Figure 2.5 a complete mesh on a rectangular domain. The local counterclock-wise ordering of nodes shown for $e_9$ and $e_{10}$ applies to all elements.

The connection between the local and global node numbers is shown in Table 2.1. This is an example of an Element-To-Vertex connectivity table `EToV` for a mesh in two space dimensions.

For use in implementations, e.g. in defining and imposing boundary conditions, it is necessary to determine the outer normal vectors to the boundary edges of elements bordering the domain boundaries. If the order of the local element vertex nodes is counterclock-wise, then for a boundary edge defined from two vertex nodes $v_1 = (x_1, y_1)^T$ and $v_2 = (x_2, y_2)^T$ we can determine a tangential vector to the edge from the differences

$$\Delta x = x_2 - x_1, \quad \Delta y = y_2 - y_1 \tag{2.4}$$

The tangential vector for the edge is

$$\mathbf{t}_{12} = (t_1, t_2)^T = (\Delta x, \Delta y)^T \tag{2.5}$$

which is by definition orthogonal to the normal vector. Hence an outer normalized vector to the element edge is found to be

$$\mathbf{n} = (n_1, n_2)^T = \frac{(t_2, -t_1)^T}{\sqrt{t_1^2 + t_2^2}} \tag{2.6}$$

## 2.3 Interpolation

Let there be given a function $u(x, y)$ defined on $\overline{\Omega}$, and consider the function

$$u_I(x, y) = \sum_{i=1}^{M} u(x_i, y_i) N_i(x, y), \qquad (x, y) \in \overline{\Omega} \tag{2.7}$$

Since each $N_i(x, y)$ is continuous and piecewise linear, so is $u_I(x, y)$. Further, it is easily seen that

$$u_I(x_i, y_i) = u(x_i, y_i), \quad i = 1, 2, ..., M$$

Thus $u_I(x, y)$ interpolates $u(x, y)$ at the nodes. Geometrically, $u_I(x, y)$ describes a surface made up of triangles joined at the edges.

We now give bounds for the interpolation error $(u_I(x, y) - u(x, y))$ under the assumption that the second-order derivatives of $u(x, y)$ are bounded; i.e., there exists a constant $M_2$ such that

$$|u_{xx}| \leq M_2\,, \quad |u_{xy}| \leq M_2\,, \quad |u_{yy}| \leq M_2\,, \qquad (x, y) \in \overline{\Omega}$$

We suppose for simplicity that the boundary of the mesh coincides with $\Gamma$.

Then a constant $C$ exists, independent of $u$ and the mesh, such that for $i = 1, 2, ..., N$,

$$|u_I(x, y) - u(x, y)| \leq C M_2 h_i^2\,, \qquad (x, y) \in e_i$$

where $h_i$ is the largest edge length in $e_i$. (See [53]). It then follows that

$$|u_I(x, y) - u(x, y)| \leq C M_2 h^2\,, \qquad (x, y) \in \overline{\Omega} \tag{2.8}$$

where $h$ is the largest edge length in the entire mesh.

Finite element basis functions are useful for defining a function $w(x, y)$ everywhere in $\overline{\Omega}$ given only the values $w_i$, $i = 1, 2, ..., M$ at the nodes. The formula is

$$w(x, y) = \sum_{i=1}^{M} w_i N_i(x, y), \qquad (x, y) \in \overline{\Omega} \tag{2.9}$$

## 2.4 Stationary heat conduction

In this section we show how the analysis of stationary heat conduction in a two-dimensional body leads to a partial differential equation for its temperature. The relevant quantities for this discussion are:

| | |
|---|---|
| $q_1(x, y)$: | the heat flux in the $x$ direction |
| $q_2(x, y)$: | the heat flux in the $y$ direction |
| $\lambda_1(x, y)$: | the heat conductivity in the $x$ direction |
| $\lambda_2(x, y)$: | the heat conductivity in the $y$ direction |
| $u(x, y)$: | temperature |
| $\tilde{q}(x, y)$: | a heat source (if $\tilde{q}(x, y) > 0$) or heat sink (if $\tilde{q}(x, y) < 0$) |

We will need Fourier's law, which states that

$$q_1(x, y) = -\lambda_1(x, y) u_x(x, y), \qquad q_2(x, y) = -\lambda_2(x, y) u_y(x, y), \qquad (x, y) \in \overline{\Omega}$$

In the following discussion $(x, y)$ denotes a *fixed* point in the interior of the body. Consider a small square, $S$, with center at (x,y) and edges $\sigma_1$, $\sigma_2$, $\sigma_3$ and $\sigma_4$, as shown in Figure 2.6. For

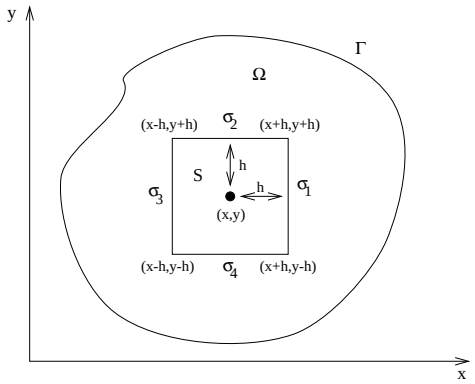



Figure 2.6: A square, $S$, in $\Omega$.

sufficiently small values of $h$ the entire square is in the body. Whether heat enters or leaves $S$ at a given point on its boundary depends on the sign of $u_x$ or $u_y$ at that point and the part of the boundary the point is on. Assuming for simplicity that these derivatives are nonzero at the point $(x, y)$, then for small enough values of h each derivative will have constant sign everywhere in the interior and on the boundary of $S$. In the case when both derivatives are negative, it is easily seen that heat enters $S$ at $\sigma_3$ and $\sigma_4$ and leaves at $\sigma_1$ and $\sigma_2$. More precisely, we have:

Heat in : $\int_{\sigma_3} q_1\, ds + \int_{\sigma_4} q_2\, ds$

Heat out : $\int_{\sigma_1} q_1\, ds + \int_{\sigma_2} q_2\, ds$

Heat produced : $\int\int_S \tilde{q}\, dx\, dy$

The law of conservation of thermal energy leads to the *heat balance equation*

$$(Heat\ out) \quad - \quad (Heat\ in) \quad = \quad (Heat\ produced)$$

or

$$\int_{\sigma_1} q_1\, ds + \int_{\sigma_2} q_2\, ds - \int_{\sigma_3} q_1\, ds - \int_{\sigma_4} q_2\, ds = \int\int_S \tilde{q}\, dx\, dy \tag{2.10}$$

While we have assumed here that both $u_x$ and $u_y$ are negative, it is easy to confirm that (2.10) is valid regardless of the signs of these derivatives.

Consider now the integral on $\sigma_1$. This edge consists of the points $(x + h, y + s)$, $-h \le s \le h$, and we have the expansion

$$q_1(x + h, y + s) = q_1 + h(q_1)_x + s(q_1)_y + \frac{1}{2!}[h^2(q_1)_{xx} + 2hs(q_1)_{xy} + s^2(q_1)_{yy}] + \mathcal{O}(h^3)$$

where, on the right-hand side, $q_1$ and its derivatives are evaluated at the center, $(x, y)$, of $S$. A simple calculation shows then that

$$\int_{\sigma_1} q_1\, ds = \int_{-h}^{h} q_1(x + h, y + s)\, ds = 2hq_1 + 2h^2(q_1)_x + h^3[(q_1)_{xx} + \frac{1}{3}(q_1)_{yy}] + \mathcal{O}(h^4)$$

The same analysis for the integral on $\sigma_3$ yields

$$\int_{\sigma_3} q_1\, ds = \int_{-h}^{h} q_1(x - h, y + s)\, ds = 2hq_1 - 2h^2(q_1)_x + h^3[(q_1)_{xx} + \frac{1}{3}(q_1)_{yy}] + \mathcal{O}(h^4)$$

so

$$\int_{\sigma_1} q_1\, ds - \int_{\sigma_3} q_1\, ds = 4h^2(q_1)_x + \mathcal{O}(h^4)$$

Similarly, one finds that

$$\int_{\sigma_2} q_2\, ds - \int_{\sigma_4} q_2\, ds = 4h^2 (q_2)_y + \mathcal{O}(h^4)$$

Further computation (we omit the details) shows that

$$\int\int_S \tilde{q}\, dx\, dy = 4h^2 \tilde{q}(x,y) + \mathcal{O}(h^4)$$

Combining the three last results with (2.10) and collecting terms that are $\mathcal{O}(h^4)$ yields the relation

$$4h^2[(q_1)_x + (q_2)_y] = 4h^2\tilde{q} + \mathcal{O}(h^4)$$

Dividing by $4h^2$ and letting $h \to 0$ we obtain then

$$(q_1)_x + (q_2)_y = \tilde{q}$$

Finally, from Fourier's law above we deduce the following partial differential equation for temperature:

$$(\lambda_1(x,y)u_x)_x + (\lambda_2(x,y)u_y)_y = -\tilde{q}(x,y) \tag{2.11}$$

An important special case of (2.11) is that when

$$\lambda_1(x,y) = \lambda_2(x,y) = \lambda \ \ (constant)$$

(2.11) then reduces to *Poisson's equation*

$$u_{xx} + u_{yy} = -\frac{\tilde{q}(x,y)}{\lambda} \tag{2.12}$$

If, in addition, $\tilde{q} = 0$ then we have *Laplace's equation*

$$u_{xx} + u_{yy} = 0 \tag{2.13}$$

## 2.5 Boundary Conditions

Equation (2.11) has many solutions. To be able to find a unique solution, we must supplement (2.11) with an appropriate boundary condition. The most common boundary conditions are the following:

*Dirichlet boundary condition:*

$$u = f(x,y), \quad (x,y) \in \Gamma \tag{2.14}$$

*Neumann boundary condition:*

$$\lambda_1(x,y)\, u_x\, n_1(x,y) + \lambda_2(x,y)\, u_y\, n_2(x,y) = -q(x,y), \quad (x,y) \in \Gamma \tag{2.15}$$

*Robin boundary condition:*

$$\lambda_1(x,y)\, u_x\, n_1(x,y) + \lambda_2(x,y)\, u_y\, n_2(x,y) = -h(x,y)[u - u_0(x,y)], \quad (x,y) \in \Gamma \tag{2.16}$$

Physically, the function $f(x,y)$ is a given temperature on the boundary. $q(x,y)$ is the heat flux on the boundary with respect to the outer normal vector, $n_1$ and $n_2$ being the direction cosines of this vector with respect to the $x$ and $y$ axes, respectively. The outer normal vector for an element edge can be determined as described in Section 2.2. (See Figure 2.7). If $q(x,y) > 0$ then heat

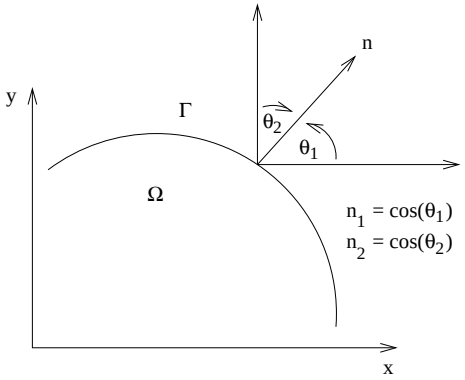



Figure 2.7: The outer normal vector on Γ.

leaves the body at point (x,y); if $q(x, y) < 0$ then heat enters the body at that point. The condition $q(x, y) = 0$ everywhere on the boundary means that the body is insulated.

$h(x, y)$ is the convective heat transfer coefficient at the boundary, and $u_0$ is the external (ambient) temperature.

Among the above boundary conditions, the Neumann condition (2.15) is special in that it must be combined with one of the others. This is because if a function $u(x, y)$ satisfies (2.11) in $\Omega$ and (2.15) on all of $\Gamma$ then so does the function $(u(x, y) + c)$, where c is any constant. That is, we have a problem with an infinite number of solutions.

Let $\Gamma$ be decomposed into two pieces, $\Gamma_1$ and $\Gamma_2$. An example of the valid use of the Neumann boundary condition is

$$\lambda_1 u_x n_1 + \lambda_2 u_y n_2 = -q, \qquad (x, y) \in \Gamma_1$$

together with

$$u = f, \qquad (x, y) \in \Gamma_2$$

or

$$\lambda_1 u_x n_1 + \lambda_2 u_y n_2 = -h(u - u_0), \qquad (x, y) \in \Gamma_2$$

Cases such as this, where boundary conditions of different types are imposed on different parts of the boundary, are called *mixed boundary conditions*.

## 2.6 The Finite Element Method

Here we show how to apply the finite element method to the problem consisting of the partial differential equation (2.11) together with the Dirichlet boundary condition (2.14). First, however, we must derive the so-called weak formulation associated with this problem (just as we had to derive (1.19) from (1.14) in Section 1.4).

Let $v(x, y)$ be an arbitrary smooth function defined on $\overline{\Omega}$ with the property that

$$v(x, y) = 0, \qquad (x, y) \in \Gamma \tag{2.17}$$

Multiplying both sides of (2.11) by $v$ and integrating over $\overline{\Omega}$, we obtain

$$\int\int_\Omega [(\lambda_1 u_x)_x + (\lambda_2 u_y)_y] v \, dx \, dy = -\int\int_\Omega \tilde{q} v \, dx \, dy$$

Integration by parts with respect to $x$ and $y$ leads to the following fundamental identities:

$$\int\int_\Omega (\lambda_1 u_x)_x v \, dx \, dy = \int_\Gamma \lambda_1 u_x v n_1 \, ds - \int\int_\Omega \lambda_1 u_x v_x \, dx \, dy$$

$$\int\int_\Omega (\lambda_2 u_y)_y v \, dx \, dy = \int_\Gamma \lambda_2 u_y v n_2 \, ds - \int\int_\Omega \lambda_2 u_y v_y \, dx \, dy$$

Here $n_1$ and $n_2$ are the direction cosines shown in Figure 2.7. Adding the above we obtain

$$\int_\Gamma (\lambda_1 u_x n_1 + \lambda_2 u_y n_2) v \, ds - \int\int_\Omega (\lambda_1 u_x v_x + \lambda_2 u_y v_y) \, dx \, dy = -\int\int_\Omega \tilde{q} v \, dx \, dy \tag{2.18}$$

Since (2.17) makes the first integral vanish, we have then

$$\int\int_\Omega (\lambda_1 u_x v_x + \lambda_2 u_y v_y) \, dx \, dy = \int\int_\Omega \tilde{q} v \, dx \, dy \tag{2.19}$$

Thus the weak formulation of problem (2.11), (2.14) is the problem of finding a function $u$ that

1. satisfies (2.19) for all sufficiently smooth functions $v$ with property (2.17),
2. satisfies (2.14) on $\Gamma$.

It is this formulation that provides the basis for applying the finite element method.

Consider now a mesh on $\overline{\Omega}$ with elements $e_n$, $n = 1, 2, ..., N$ and nodes $(x_i, y_i)$, $i = 1, 2, ..., M$. We seek an approximation $\hat{u}(x, y)$ to $u(x, y)$ of the form

$$\hat{u}(x, y) = \sum_{j=1}^{M} \hat{u}_j N_j(x, y) \tag{2.20}$$

where $N_j(x, y)$ is the typical global basis function. From (2.20) and the fact that $N_i(x_i, y_i) = 1$, $N_j(x_i, y_i) = 0$, $j \neq i$, we see that if we put

$$\hat{u}_i = f(x_i, y_i) \quad \text{for all} \quad i \quad \text{such that} \quad (x_i, y_i) \in \Gamma \tag{2.21}$$

then we will have

$$\hat{u}(x_i, y_i) = f(x_i, y_i) \quad \text{for all} \quad i \quad \text{such that} \quad (x_i, y_i) \in \Gamma$$

In other words, (2.21) enforces the Dirichlet boundary condition at the boundary nodes.

Let $M_\Omega$ and $M_\Gamma$ denote the number of nodes in $\Omega$ and $\Gamma$, respectively. We know $M_\Gamma$ of the coefficients $\hat{u}_1$,$\hat{u}_2$, ...,$\hat{u}_M$ from (2.21). To find the remaining $M_\Omega$ coefficients, we substitute (2.20) in (2.19) and let $v$ be, successively, the functions $N_i(x, y)$ for every $(x_i, y_i) \in \Omega$. (Note that these functions vanish on $\Gamma$, as required). This procedure leads to $M_\Omega$ equations of the type

$$\int\int_\Omega [\lambda_1 \, \hat{u}_x \, (N_i)_x + \lambda_2 \, \hat{u}_y \, (N_i)_y] \, dx \, dy = \int\int_\Omega \tilde{q} N_i \, dx \, dy$$

or

$$\int\int_\Omega [\lambda_1 \sum_{j=1}^{M} \hat{u}_j \, (N_j)_x (N_i)_x + \lambda_2 \sum_{j=1}^{M} \hat{u}_j \, (N_j)_y (N_i)_y] \, dx \, dy = \int\int_\Omega \tilde{q} N_i \, dx \, dy$$

or

$$\sum_{j=1}^{M} a_{i,j} \hat{u}_j = b_i \quad \text{for all} \quad i \quad \text{such that} \quad (x_i, y_i) \in \Omega \tag{2.22}$$

where

$$a_{i,j} = \int\int_\Omega [\lambda_1 \, (N_i)_x (N_j)_x + \lambda_2 \, (N_i)_y (N_j)_y] \, dx \, dy \tag{2.23}$$

$$b_i = \int\int_\Omega \tilde{q} N_i \, dx \, dy \tag{2.24}$$

## 2.7 Element matrices

The form of the global basis functions is such that $a_{i,j}$ is nonzero only if $N_i(x,y)$ and $N_j(x,y)$ overlap, and this happens only if nodes $(x_i,y_i)$ and $(x_j,y_j)$ are directly connected in the mesh. In Figure 2.8 we see, for example, that nodes 11 and 15 are directly connected. Since $N_{11}(x,y)$ is

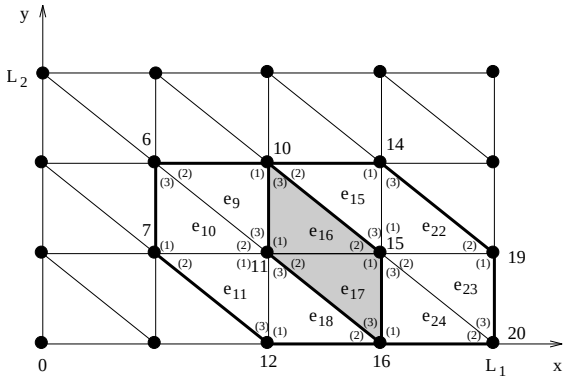



Figure 2.8: A detail of the mesh in Figure 2.5.

nonzero in elements 9, 10, 11, 16, 17 and 18, while $N_{15}(x,y)$ is nonzero in elements 15, 16, 17, 22, 23 and 24, we have

$$\begin{aligned} a_{11,15} &= \int\int_{e_{16}} [\lambda_1 (N_{11})_x (N_{15})_x + \lambda_2 (N_{11})_y (N_{15})_y]\, dx\, dy \\ &\quad + \int\int_{e_{17}} [\lambda_1 (N_{11})_x (N_{15})_x + \lambda_2 (N_{11})_y (N_{15})_y]\, dx\, dy \end{aligned}$$

In terms of the local basis functions this becomes

$$\begin{aligned} a_{11,15} &= \int\int_{e_{16}} [\lambda_1 (N_1^{(16)})_x (N_2^{(16)})_x + \lambda_2 (N_1^{(16)})_y (N_2^{(16)})_y]\, dx\, dy \\ &\quad + \int\int_{e_{17}} [\lambda_1 (N_2^{(17)})_x (N_1^{(17)})_x + \lambda_2 (N_2^{(17)})_y (N_1^{(17)})_y]\, dx\, dy \end{aligned}$$

We now introduce for each element $e_n$ the element matrix

$$\mathbf{K_n} = \begin{bmatrix} k_{1,1}^{(n)} & k_{1,2}^{(n)} & k_{1,3}^{(n)} \\ k_{2,1}^{(n)} & k_{2,2}^{(n)} & k_{2,3}^{(n)} \\ k_{3,1}^{(n)} & k_{3,2}^{(n)} & k_{3,3}^{(n)} \end{bmatrix}$$

where

$$k_{r,s}^{(n)} = \int\int_{e_n} [\lambda_1 (N_r^{(n)})_x (N_s^{(n)})_x + \lambda_2 (N_r^{(n)})_y (N_s^{(n)})_y]\, dx\, dy \tag{2.25}$$

Observe that we can express $a_{11,15}$ above as

$$a_{11,15} = k_{1,2}^{(16)} + k_{2,1}^{(17)}$$

This is typical: each nonzero $a_{i,j}$ in (2.23) is the sum of some number of element matrix entries. If $i \neq j$ then this number is precisely two. If $i = j$ then it is the number of elements to which the $i$th node belongs.

We regard now $b_i$ given by (2.24). From Figure 2.8 we see, for example, that

$$\begin{aligned} b_{11} &= \sum_{n=9,10,11,16,17,18} \int\int_{e_n} \tilde{q} N_{11}\, dx\, dy \\ &= \int\int_{e_9} \tilde{q} N_3^{(9)}\, dx\, dy + \int\int_{e_{10}} \tilde{q} N_2^{(10)}\, dx\, dy + \int\int_{e_{11}} \tilde{q} N_1^{(11)}\, dx\, dy \\ &\quad + \int\int_{e_{16}} \tilde{q} N_1^{(16)}\, dx\, dy + \int\int_{e_{17}} \tilde{q} N_2^{(17)}\, dx\, dy + \int\int_{e_{18}} \tilde{q} N_3^{(18)}\, dx\, dy \end{aligned}$$

For each element $e_n$ we define an element vector by

$$\tilde{\mathbf{q}}_\mathbf{n} = \begin{bmatrix} \tilde{q}_1^{(n)} \\ \tilde{q}_2^{(n)} \\ \tilde{q}_3^{(n)} \end{bmatrix}$$

where

$$\tilde{q}_r^{(n)} = \int\int_{e_n} \tilde{q} N_r^{(n)}\, dx\, dy \tag{2.26}$$

Every $b_i$ can be expressed as a sum of components of element vectors. For example,

$$b_{11} = \tilde{q}_3^{(9)} + \tilde{q}_2^{(10)} + \tilde{q}_1^{(11)} + \tilde{q}_1^{(16)} + \tilde{q}_2^{(17)} + \tilde{q}_3^{(18)}$$

To evaluate $k_{r,s}^{(n)}$ and $\tilde{q}_r^{(n)}$ we insert in (2.25) and (2.26), respectively, the expressions for the local basis functions given in (2.2). The result is

$$k_{r,s}^{(n)} = \frac{1}{4\Delta^2} \int\int_{e_n} (\lambda_1 \tilde{b}_r \tilde{b}_s + \lambda_2 \tilde{c}_r \tilde{c}_s)\, dx\, dy\,, \qquad r,s = 1,2,3$$

$$\tilde{q}_r^{(n)} = \frac{1}{2\Delta} \int\int_{e_n} \tilde{q}(\tilde{a}_r + \tilde{b}_r x + \tilde{c}_r y)\, dx\, dy\,, \qquad r = 1,2,3$$

In the case when $\lambda_1$, $\lambda_2$ and $\tilde{q}$ are constant with respect to $x$ and $y$, one finds that

$$k_{r,s}^{(n)} = \frac{1}{4|\Delta|}(\lambda_1 \tilde{b}_r \tilde{b}_s + \lambda_2 \tilde{c}_r \tilde{c}_s) \tag{2.27}$$

$$\tilde{q}_r^{(n)} = \frac{|\Delta|}{3} \tilde{q} \tag{2.28}$$

If any of these functions is not constant, it can be approximated in (2.27) or (2.28) either by its value at the center $(x_c\,, y_c)$ of $e_n$, defined by

$$(x_c\,, y_c) = \left( \frac{x_1 + x_2 + x_3}{3}\,,\ \frac{y_1 + y_2 + y_3}{3} \right) \tag{2.29}$$

or by the average of its values at the three nodes. (Note that in (2.29) we are using *local* node numbers).

## 2.8 The computation

We need to compute the various coefficients $a_{i,j}$ and $b_i$ in system (2.22) and then solve this system. Regarding the first task, it is convenient to divide this into two parts, the first part performing most of the work but ignoring the boundary condition and the second part imposing the Dirichlet boundary condition. This approach leads to Algorithms 4 and 6 below which are analogous to Algorithms 1 and 2, respectively, in Section 1.5.

We have seen that each nonzero $a_{i,j}$ can be expressed as the sum of entries of element matrices and that each nonzero $b_i$ can be expressed as the sum of components of element vectors. This fact makes it efficient to construct the various $a_{i,j}$ and $b_i$ by an 'elementwise' computation. More precisely, we can go through the elements one by one, computing for each element its element matrix and element vector and using this data to update the $a_{i,j}$ and $b_i$ affected by that element. This procedure is carried out by the algorithm below which makes use of an $M \times M$ array, $a$, for the $a_{i,j}$ and an $M \times 1$ array, $b$, for the $b_i$. The use of an $M \times M$ array for the $a_{i,j}$ is for clarity only. In practice a sparse data structure would be used for these values.

It should be realized that although the algorithm affects all rows of arrays $a$ and $b$, the only rows relevant for system (2.22) are those with a row number $i$ such that $(x_i, y_i) \in \Omega$. The other rows will be modified later so as to express the boundary condition.

---

**Algorithm 4:** Global assembly of coefficient matrix **A** and vector **b** (2D).

---

Allocate full arrays **A** and **b**
**for** $n := 1$ **to** $N$
    Look up the global numbers $(i, j, k)$ and $(x, y)$-coordinates of the nodes in $e_n$.
    **for** $r := 1$ **to** 3
        Compute $\tilde{q}_r^{(n)}$ from (2.28).
        $b[i] := b[i] + \tilde{q}_r^{(n)}$ (N.B. $i$ is the global number of node r).
        **for** $s := 1$ **to** 3
            Compute $k_{r,s}^{(n)}$ from (2.27).
            $a[i, j] := a[i, j] + k_{r,s}^{(n)}$ (N.B. $j$ is the global number of node $s$).

---

It will be noted that the algorithm takes no advantage of the symmetry inherent in (2.23) and (2.25), namely $a_{i,j} = a_{j,i}$ and $k_{r,s}^{(n)} = k_{s,r}^{(n)}$. These relations show that it suffices to compute $a_{i,j}$ for $j \geq i$ and $k_{r,s}^{(n)}$ for $s \geq r$. Another weakness is the inefficiency of the initialization of array **A**. We recall that the matrix entry $a_{i,j}$ is nonzero only if nodes $(x_i, y_i)$ and $(x_j, y_j)$ are directly connected in the mesh. Hence, logically, only the corresponding entries of the array need to be addressed. Algorithm 5 below eliminates both types of unnecessary work with only minor changes to Algorithm 4 and making use of sparse storage. In Matlab we can make use of the sparse commands `sparse` and `spalloc` (refer to the Matlab help info for appropriate use hereof).

---

**Algorithm 5:** Global assembly of coefficient matrix **A** and vector **b** (2D, exploit symmetry).

---

Allocate sparse **A** and **b**
**for** $n := 1$ **to** $N$
    Look up the global numbers $(i, j)$ and $(x, y)$-coordinates of the nodes in $e_n$.
    **for** $r := 1$ **to** 3
        Compute $\tilde{q}_r^{(n)}$ from (2.28).
        $b[i] := b[i] + \tilde{q}_r^{(n)}$
        **for** $s := r$ **to** 3
            Compute $k_{r,s}^{(n)}$ from (2.27).
            **if** $j \geq i$
                $a[i, j] := a[i, j] + k_{r,s}^{(n)}$
            **else**
                $a[j, i] := a[j, i] + k_{r,s}^{(n)}$

---

We consider now the handling of the Dirichlet boundary condition. First, to illustrate what we want to achieve, consider the case where $M = 4$ and make the assumption that there is only a single

node on the boundary, namely $(x_2, y_2)$. Let the system of equations as it exists after Algorithm 4 be expressed as

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

The boundary condition makes $\hat{u}_2 = f_2$. Hence the system we want to solve may be expressed as

$$\begin{bmatrix} a_{1,1} & 0 & a_{1,3} & a_{1,4} \\ 0 & 1 & 0 & 0 \\ a_{3,1} & 0 & a_{3,3} & a_{3,4} \\ a_{4,1} & 0 & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} = \begin{bmatrix} b_1 - a_{1,2} f_2 \\ f_2 \\ b_3 - a_{3,2} f_2 \\ b_4 - a_{4,2} f_2 \end{bmatrix} \tag{2.30}$$

The purpose of Algorithm 6 is to perform modifications of this type and it should be executed after Algorithm 4.

**Algorithm 6:** Imposing boundary conditions by modification of system (2D).

```
if (x_i, y_i) ∈ Γ
    a[i,i] := 1, b[i] := f_i
    for j := 1 to M   (j ≠ i)
        a[i,j] := 0
        if (x_j, y_j) ∈ Ω
            b[j] := b[j] − a[j,i] * f_i
            a[j,i] := 0
```

The weaknesses of this computation are that it fails to exploit symmetry and that much of the execution time is spent on pointless work (because most of the $a[i,j]$ and $a[j,i]$ in the innermost for-loop contain zeros from the start). Algorithm 7 below, which should be executed after Algorithm 4, both takes symmetry into account and avoids (most) pointless work. Let $S_\Gamma$ denote the set of global node numbers of nodes on $\Gamma$. For every $i \in S_\Gamma$, let $S_{\overline{\Omega}}^{(i)}$ denote the set of global node numbers of those nodes in $\overline{\Omega}$ that are directly connected to $(x_i\,, y_i)$. For the mesh in Figure 2.5, for example, we have

$$S_\Gamma = \{1,2,3,4,5,8,9,12,13,16,17,18,19,20\}, \quad S_{\overline{\Omega}}^{(5)} = \{1,6,9,10\}$$

**Algorithm 7:** Imposing boundary conditions by modification of system (2D, exploit symmetry).

```
for  i ∈ S_Γ
    a[i,i] := 1, b[i] := f_i
    for  j ∈ S_Ω̄^(i)
        if j < i
            b[j] := b[j] − a[j,i] * f_i
            a[j,i] := 0
        else
            b[j] := b[j] − a[i,j] * f_i
            a[i,j] := 0
```

After conclusion of Algorithm 7 we have an $M$'th-order system of equations to be solved that can be expressed in the compact form as a system of equations

$$\boldsymbol{A}\hat{\boldsymbol{u}} = \boldsymbol{b} \tag{2.31}$$

For the mesh shown in Figure 2.5, $\boldsymbol{A}$ and $\boldsymbol{b}$ have the following appearance:

$$\begin{bmatrix}
1 & & & & & & & & & & & & & & & & & & & \\
 & 1 & & & & & & & & & & & & & & & & & & \\
 & & 1 & & & & & & & & & & & & & & & & & \\
 & & & 1 & & & & & & & & & & & & & & & & \\
 & & & & 1 & & & & & & & & & & & & & & & \\
 & & & & & x & x & & & x & x & & & & & & & & & \\
 & & & & & x & x & & & & x & & & & & & & & & \\
 & & & & & & & 1 & & & & & & & & & & & & \\
 & & & & & & & & 1 & & & & & & & & & & & \\
 & & & & & x & & & & x & x & & & x & x & & & & & \\
 & & & & & x & x & & & x & x & & & & x & & & & & \\
 & & & & & & & & & & & 1 & & & & & & & & \\
 & & & & & & & & & & & & 1 & & & & & & & \\
 & & & & & & & & & x & & & & x & x & & & & & \\
 & & & & & & & & & x & x & & & x & x & & & & & \\
 & & & & & & & & & & & & & & & 1 & & & & \\
 & & & & & & & & & & & & & & & & 1 & & & \\
 & & & & & & & & & & & & & & & & & 1 & & \\
 & & & & & & & & & & & & & & & & & & 1 & \\
 & & & & & & & & & & & & & & & & & & & 1
\end{bmatrix}, \quad \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ x \\ x \\ f_8 \\ f_9 \\ x \\ x \\ f_{12} \\ f_{13} \\ x \\ x \\ f_{16} \\ f_{17} \\ f_{18} \\ f_{19} \\ f_{20} \end{bmatrix}$$

For every $i \in S_\Gamma$, the $i$th row of this system is of the form $\hat{u}_i = f_i$, and these rows are uncoupled from the remainder of the system with inhomogenous boundary contributions now imposed through the right hand vector **b**. This uncoupling is most obvious when the six nodes in the interior of the domain are ordered before those on the boundary. Consider, for example, the node ordering shown in Figure 2.9.

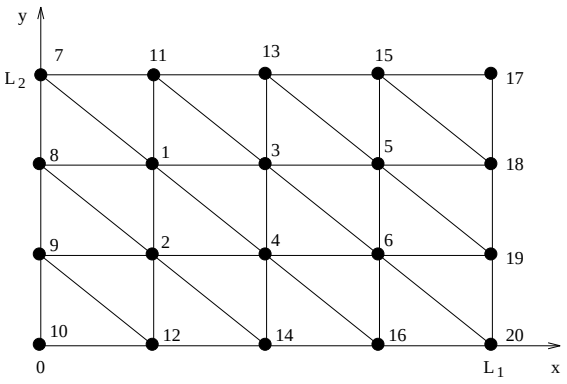



Figure 2.9: Reordering of nodes in the mesh of Figure 2.5.

Then $\boldsymbol{A}$ and $\boldsymbol{b}$ assume the form

$$\begin{bmatrix}
x & x & x & x & & & & & & & & & & & & & & & & \\
x & x & & x & & & & & & & & & & & & & & & & \\
x & & x & x & x & x & & & & & & & & & & & & & \\
x & x & x & x & & x & & & & & & & & & & & & & \\
 & & x & & x & x & & & & & & & & & & & & & \\
 & & x & x & x & x & & & & & & & & & & & & & \\
 & & & & & & 1 & & & & & & & & & & & & & \\
 & & & & & & & 1 & & & & & & & & & & & & \\
 & & & & & & & & 1 & & & & & & & & & & & \\
 & & & & & & & & & 1 & & & & & & & & & & \\
 & & & & & & & & & & 1 & & & & & & & & & \\
 & & & & & & & & & & & 1 & & & & & & & & \\
 & & & & & & & & & & & & 1 & & & & & & & \\
 & & & & & & & & & & & & & 1 & & & & & & \\
 & & & & & & & & & & & & & & 1 & & & & & \\
 & & & & & & & & & & & & & & & 1 & & & & \\
 & & & & & & & & & & & & & & & & 1 & & & \\
 & & & & & & & & & & & & & & & & & 1 & & \\
 & & & & & & & & & & & & & & & & & & 1 & \\
 & & & & & & & & & & & & & & & & & & & 1
\end{bmatrix}, \quad \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ f_7 \\ f_8 \\ f_9 \\ f_{10} \\ f_{11} \\ f_{12} \\ f_{13} \\ f_{14} \\ f_{15} \\ f_{16} \\ f_{17} \\ f_{18} \\ f_{19} \\ f_{20} \end{bmatrix}$$

The first six unknowns can be found by solving the smaller system

$$\begin{bmatrix}
x & x & x & x & & \\
x & x & & x & & \\
x & & x & x & x & x \\
x & x & x & x & & x \\
 & & x & & x & x \\
 & & x & x & x & x
\end{bmatrix}
\begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \\ \hat{u}_5 \\ \hat{u}_6 \end{bmatrix} = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \end{bmatrix}$$

In general, whenever the boundary condition is of Dirichlet type then one could work with a system of order $M_\Omega$ instead of one of order $M$. However, it is usually easiest *not* to do so, and since the difference in the sizes of the two systems loses significance as the mesh is refined (because there are $\mathcal{O}(M)$ interior nodes and only $\mathcal{O}(\sqrt{M})$ boundary nodes), there is little reason not to work with the larger system.

We consider now methods for solving (2.31) and the related issue of the choice of a practical data structure for $\boldsymbol{A}$. This matrix is symmetric positive definite. In the case of a fine mesh it is also very sparse, since the number of off-diagonal nonzeros in the $i$th row cannot exceed the number of nodes directly connected to node $(x_i, y_i)$. For a mesh of the type shown in Figure 2.5, for example, where each node is connected to at most 6 other nodes, the number of nonzero entries in a row is at most 7 (including the entry on the main diagonal). Since the total number of entries of $\boldsymbol{A}$ is $M^2$, the density of the nonzero entries in this case is bounded by $7/M$ and hence goes to zero as the mesh is refined.

An important type of sparse matrix is a *band matrix*, where the nonzero entries are more or less concentrated along the main diagonal. Let $\phi(i)$ denote the column number of the last nonzero in the $i$th row of $\boldsymbol{A}$, and let

$$h(\boldsymbol{A}) = \max_{1 \le i \le M} \{\phi(i) - i + 1\}$$

$$b(\boldsymbol{A}) = 2h(\boldsymbol{A}) - 1$$

$h(\boldsymbol{A})$ and $b(\boldsymbol{A})$ are called the *half-bandwidth* and *bandwidth* of $\boldsymbol{A}$, respectively. For example, the matrix immediately below (2.31) has $h(\boldsymbol{A}) = 6$ and $b(\boldsymbol{A}) = 11$. Since $\phi(i)$ is the largest node number of all nodes connected directly to $(x_i, y_i)$, it follows that $\boldsymbol{A}$ has a small bandwidth only

if the node ordering is such that every pair of connected nodes have node numbers close to one another. There are effective algorithms for finding node orderings that come close to minimizing the bandwidth of a matrix; see, for example, [25] and [35]. Generally speaking, the possibility of achieving a small bandwidth is greatest in the case of a long and narrow mesh, in which case the node ordering should run along the 'short' dimension. See Figure 2.10 for a simple example.

However, it turns out that the minimal bandwidth for all node orderings is no better than

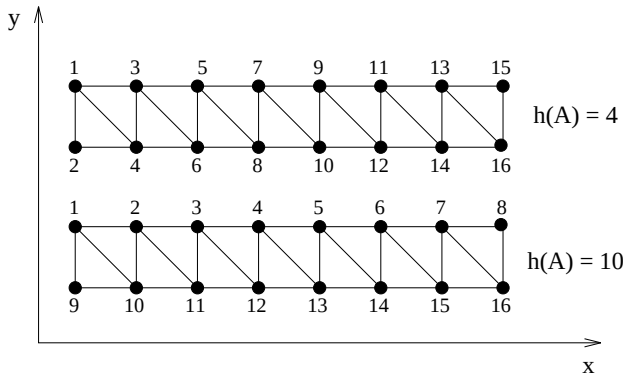



Figure 2.10: Two orderings of a mesh and the corresponding half-bandwidths. (Boundary conditions are ignored).

$b(\boldsymbol{A}) = \mathcal{O}(\sqrt{M})$. Consequently, as the mesh is refined the bandwidth increases and the density of nonzero entries in the band falls like $\mathcal{O}(1/\sqrt{M})$. This is a fundamental difference between problems in one and several space dimensions (the matrix in (1.30) has bandwidth 3 for *all* $M$) and tends to make band matrix techniques less effective in problems with more than one space dimension.

There are two general groups of methods for solving (2.31): direct and iterative. In the absence of rounding errors, direct methods can solve (2.31) exactly in a finite number of simple arithmetical operations. (Here we regard the computation of a square root as a simple arithmetical operation). Examples are Gaussian elimination and the Cholesky method. Unfortunately, in the case of sparse systems direct methods usually create fill-ins; i.e., nonzero entries in the matrix positions originally occupied by zeros. Nevertheless, there is a strong tradition for applying direct methods to finite element problems. Since these methods are easy to program when $\boldsymbol{A}$ is a band matrix, it is very common to store $\boldsymbol{A}$ in this form. For example, the diagonals in the upper half-band (including the main diagonal) can be stored as columns in an $(M \times h(\boldsymbol{A}))$ array. This is sufficient storage because fill-ins are confined to the band (and tend to fill it). When $\boldsymbol{A}$ is treated as a band matrix, it is naturally desirable that the mesh nodes have been ordered to yield as small a bandwidth as possible. In cases where $\boldsymbol{A}$ is sparse but does not have particular band structure, it is possible to employ various fill-in minimizing algorithms. For example, if the Cholesky method is employed for the solution of a sparse symmetric positive definite system the Minimum Degree (MD) algorithm can often be used to reduce the fill-in. In Matlab, this can be achieved by employing `symmmd` or the approximate multiple minimum degree function `symamd`. For a thorough treatment of direct methods for solving sparse, symmetric positive definite systems, see [25].

An iterative method for solving (2.31) produces a sequence of vectors that converges to the solution vector $\hat{\boldsymbol{u}}$. Examples are the successive overrelaxation (SOR) method and the conjugate gradient (CG) method. Iterative methods, in their simplest form, use only the original matrix entries, so there is no fill-in. The CG method, for example, requires only that the matrix-vector product $\boldsymbol{Au}$ can be computed for any given $\boldsymbol{u}$. The only $a_{i,j}$ entries that need to be stored are precisely those produced by Algorithms 4 and 7. Let the number of these be $R$. A simple storage scheme for $\boldsymbol{A}$ consists then of three single-indexed arrays of length $R$: one for the values of $i$, one for the values of $j$, and one for the values of $a_{i,j}$, where $j \geq i$.

The literature that deals with solving systems of type (2.31) is very extensive, and the above

discussion has only scratched the surface. For further reading see, for example, [3], [4], [10], [25], [31], [32], [35] and [49].

## 2.9 The Neumann boundary condition

We consider now the procedure for solving differential equation (2.11) when the boundary condition on a part of $\Gamma$ is of Neumann type (2.15). As stated earlier, this boundary condition cannot be applied to the whole of $\Gamma$ because the solution of the problem is then not unique. We therefore suppose that the boundary is divided into two parts,

$$\Gamma = \Gamma_1 \bigcup \Gamma_2$$

and that $u(x,y)$ is required to satisfy

$$\lambda_1(x,y)\, u_x\, n_1(x,y) + \lambda_2(x,y)\, u_y\, n_2(x,y) = -q(x,y)\,, \qquad (x,y) \in \Gamma_1 \tag{2.32}$$

$$u = f(x,y)\,, \qquad (x,y) \in \Gamma_2 \tag{2.33}$$

To derive the weak formulation of this problem, we return to (2.18) and require now that $v$ be zero only on $\Gamma_2$. Using (2.32), we can rewrite (2.18) as

$$\int\int_{\Omega} (\lambda_1 u_x v_x + \lambda_2 u_y v_y)\, dx\, dy = \int\int_{\Omega} \tilde{q} v\, dx\, dy - \int_{\Gamma_1} qv\, ds \tag{2.34}$$

The problem of finding a function $u$ that satisfies (2.11) in $\Omega$ and boundary conditions (2.32), (2.33) is equivalent to finding a function $u$ that

1. satisfies (2.34) for all sufficiently smooth $v$ such that $v = 0$ on $\Gamma_2$,
2. satisfies (2.33).

Let there be given a mesh on $\overline{\Omega}$ with $M$ nodes. We recall that $M_\Omega$ and $M_\Gamma$ denote, respectively, the number of nodes in $\Omega$ and the number of nodes on $\Gamma$. We define further:

| | |
|---|---|
| $M_{\Gamma_1}$: | The number of nodes on $\Gamma_1$ |
| $M_{\Gamma_2}$: | The number of nodes on $\Gamma_2$ |
| $S_\Omega$ | $i \in S_\Omega \;\Leftrightarrow\; (x_i, y_i) \in \Omega$ |
| $S_{\Gamma_1}$: | $i \in S_{\Gamma_1} \;\Leftrightarrow\; (x_i, y_i) \in \Gamma_1$ |
| $S_{\Gamma_2}$: | $i \in S_{\Gamma_1} \;\Leftrightarrow\; (x_i, y_i) \in \Gamma_2$ |

We assume that nodes are placed at the points where $\Gamma_1$ and $\Gamma_2$ meet and that these nodes are assigned to $\Gamma_2$.

To solve the above problem by the FEM, we again approximate $u(x,y)$ by a function $\hat{u}(x,y)$ of type (2.20). Because of the Dirichlet boundary condition on $\Gamma_2$, we put

$$\hat{u}_i = f_i\,, \quad i \in S_{\Gamma_2} \tag{2.35}$$

which gives us $M_{\Gamma_2}$ equations. To derive the remaining $(M - M_{\Gamma_2})$ equations, we substitute (2.20) for $u$ in (2.34) and let $v$ be, successively, all of the functions $N_i(x,y)$ such that $(x_i, y_i) \in \Omega \bigcup \Gamma_1$; i.e., such that $i \in S_\Omega \bigcup S_{\Gamma_1}$. The number of these functions is precisely $(M - M_{\Gamma_2})$, and they all vanish on $\Gamma_2$ as required. This procedure yields the system

$$\sum_{j=1}^{M} a_{i,j} \hat{u}_j = b_i\,, \quad i \in S_\Omega \bigcup S_{\Gamma_1} \tag{2.36}$$

where

$$a_{i,j} = \int\int_{\Omega} [\lambda_1 (N_i)_x (N_j)_x + \lambda_2 (N_i)_y (N_j)_y] \, dx \, dy \tag{2.37}$$

$$b_i = \int\int_{\Omega} \tilde{q} N_i \, dx \, dy - \int_{\Gamma_1} q N_i \, ds \tag{2.38}$$

Comparing with (2.23) and (2.24), we see that the only complication caused by the Neumann boundary condition is that we must now deal with the *boundary integrals*

$$\int_{\Gamma_1} q N_i \, ds \,, \quad i \in S_\Omega \bigcup S_{\Gamma_1} \tag{2.39}$$

However, since the form of $N_i(x, y)$ implies that (2.39) can be nonzero only if $(x_i, y_i) \in \Gamma_1$ , the problem reduces to that of computing

$$\int_{\Gamma_1} q N_i \, ds \,, \quad i \in S_{\Gamma_1} \tag{2.40}$$

Consider, as an example, the case $i = 17$ in Figure 2.11. Since $N_{17}(x, y)$ is zero everywhere

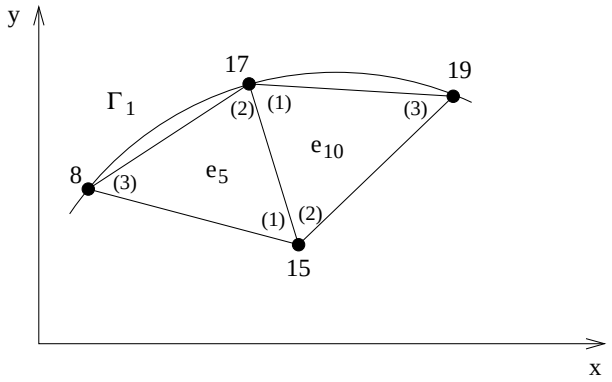



Figure 2.11: Two element edges on $\Gamma_1$.

except in elements $e_5$ and $e_{10}$ , we have

$$\begin{aligned} \int_{\Gamma_1} q N_{17} \, ds &= \int_{(x_{19}, y_{19})}^{(x_8, y_8)} q N_{17} \, ds \\ &= \int_{(x_{19}, y_{19})}^{(x_{17}, y_{17})} q N_{17} \, ds + \int_{(x_{17}, y_{17})}^{(x_8, y_8)} q N_{17} \, ds \\ &= \int_{(x_{19}, y_{19})}^{(x_{17}, y_{17})} q N_1^{(10)} \, ds + \int_{(x_{17}, y_{17})}^{(x_8, y_8)} q N_2^{(5)} \, ds \end{aligned}$$

This example is typical: We can always express (2.40) as the sum of two integrals, each defined on a single element edge on $\Gamma_1$. We call an element edge on the boundary a *boundary edge*. Note that a boundary edge on $\Gamma_1$ affects only two of the entries $b_i$ in (2.38). From Figure 2.11, for example, we see that the boundary edge from node 17 to node 8 affects only $b_{17}$ and $b_8$.

The phrase 'from node 17 to node 8' stresses that we have assigned to the boundary edge a *first* node (node 17) and a *second* node (node 8). We call node 17 the first node because it is the first node we reach when making a counterclockwise movement along the boundary of $e_5$ , starting at the node *not* belonging to the boundary edge (node 15). We can construct a table of boundary edges in which each edge on $\Gamma_1$ is represented by two integers: $n$, the number of the element to which the edge belongs, and $r$, the local node number of the first node on the edge. Assuming

that the local ordering of nodes in every element is counterclockwise, it is easy to determine $s$, the local node number of the second node on the edge.

Let $E_1$ denote the number of boundary edges on $\Gamma_1$, and let these edges be ordered in some way. We define

$$\boldsymbol{q}_p = \begin{bmatrix} q_1^{(p)} \\ q_2^{(p)} \end{bmatrix}, \quad p = 1, 2, \ldots, E_1$$

where

$$q_1^{(p)} = \int_{(x_i,y_i)}^{(x_j,y_j)} qN_r^{(n)}\, ds\,, \quad q_2^{(p)} = \int_{(x_i,y_i)}^{(x_j,y_j)} qN_s^{(n)}\, ds$$

To sum up the notation: $p$ is the edge number, $n$ is the element number, $r$ and $s$ are the local numbers of the first and second nodes on the edge, respectively, and $i$ and $j$ are the corresponding global node numbers. When $q(x, y)$ is constant then

$$q_1^{(p)} = q_2^{(p)} = \frac{q}{2}\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \tag{2.41}$$

Otherwise, one can approximate $q$ in (2.41) by either its value at the edge midpoint

$$(x_c, y_c) = \left(\frac{x_i + x_j}{2}\,,\ \frac{y_i + y_j}{2}\right) \tag{2.42}$$

or by the average of its values at the end nodes.

Algorithm 8 performs the updating required by the boundary integrals in (2.38).

**Algorithm 8:** Imposing Neumann boundary conditions by modification of system (2D).

**for** $p := 1$ **to** $E_1$
Look up $n$ and $r$ for the $p$th edge.
Determine $s$ and look up $i$ and $j$ and the (x,y)-coordinates of the end nodes.
Compute $q_1^{(p)}$ and $q_2^{(p)}$ from (2.41).
$b[i] := b[i] - q_1^{(p)}$
$b[j] := b[j] - q_2^{(p)}$

### Conclusion

When the problem to be solved consists of differential equation (2.11) and boundary conditions (2.32) and (2.33), then the computational procedure for constructing the algebraic system of equations $\boldsymbol{A\hat{u}} = \boldsymbol{b}$ without exploiting symmetry of the coefficient matrix $A$ is as follows:

1. Global assembly, Algorithm 4,
2. Modification for imposing Neumman boundary conditions, Algorithm 8,
3. Modification for imposing Dirichlet boundary conditions, Algorithm 6 (with $S_\Gamma$ replaced by $S_{\Gamma_2}$).

The order of steps 2 and 3 must not be reversed, since this can lead to incorrect values of $b_i$ for values of $i$ such that $(x_i, y_i)$ is a node where $\Gamma_1$ and $\Gamma_2$ meet.

## 2.10 The Robin boundary condition

We consider now the case when the boundary condition is of the Robin type (2.16) on all of $\Gamma$. Returning to (2.18), we must now remove the requirement that $v$ vanish on $\Gamma$. Noting that the

integrand in the boundary integral appears in (2.16), we see that the weak formulation of problem (2.11), (2.16) is that of determining the function $u$ that satisfies

$$\int\int_\Omega (\lambda_1 u_x v_x + \lambda_2 u_y v_y)\, dx\, dy + \int_\Gamma huv\, ds = \int\int_\Omega \tilde{q}v\, dx\, dy + \int_\Gamma hu_0\, v\, ds \tag{2.43}$$

for all sufficiently smooth $v$. Substituting (2.20) for $u$ in (2.43) and letting $v$ be, successively, the $M$ global basis functions, we obtain the system of equations

$$\sum_{j=1}^{M} a_{i,j}\hat{u}_j = b_i\,, \qquad i \;=\; 1, 2, ..., M \tag{2.44}$$

where

$$a_{i,j} = \int\int_\Omega [\lambda_1 (N_i)_x (N_j)_x + \lambda_2 (N_i)_y (N_j)_y]\, dx\, dy + \int_\Gamma hN_i N_j\, ds \tag{2.45}$$

$$b_i = \int\int_\Omega \tilde{q} N_i\, dx\, dy + \int_\Gamma hu_0\, N_i\, ds \tag{2.46}$$

Comparing with the case of the Neumann boundary condition, we see that the only really new feature here is the boundary integral in (2.45). Let $E$ denote the number of boundary edges on $\Gamma$, and let these edges be ordered in some way. Let

$$\boldsymbol{H}_p = \begin{bmatrix} h_{1,1}^{(p)} & h_{1,2}^{(p)} \\ \\ h_{2,1}^{(p)} & h_{2,2}^{(p)} \end{bmatrix}, \quad p = 1, 2, \ldots, E$$

where

$$h_{1,1}^{(p)} = \int_{(x_i,y_i)}^{(x_j,y_j)} h\left(N_r^{(n)}\right)^2 ds\,, \qquad h_{1,2}^{(p)} = \int_{(x_i,y_i)}^{(x_j,y_j)} hN_r^{(n)} N_s^{(n)}\, ds$$

$$h_{2,1}^{(p)} = \int_{(x_i,y_i)}^{(x_j,y_j)} hN_s^{(n)} N_r^{(n)}\, ds\,, \qquad h_{2,2}^{(p)} = \int_{(x_i,y_i)}^{(x_j,y_j)} h\left(N_s^{(n)}\right)^2 ds$$

When $h(x, y)$ is constant one finds that

$$h_{1,1}^{(n)} = h_{2,2}^{(n)} = \frac{h}{3}\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \tag{2.47}$$

$$h_{1,2}^{(n)} = h_{2,1}^{(n)} = \frac{h}{6}\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \tag{2.48}$$

Algorithm 9 below performs the updating required by the boundary integrals in (2.45).

**Algorithm 9:** Imposing Robin boundary conditions by modification of system (2D).

**for** $p := 1$ **to** $E$
  Look up $n$ and $r$ for the $p$'th edge.
  Determine $s$ and look up $i$, $j$ and the (x,y)-coordinates of the end nodes.
  Compute $h_{1,1}^{(p)}, h_{1,2}^{(p)}$ and $h_{2,2}^{(p)}$ from (2.47) and (2.48).
  $a[i,i] := a[i,i] + h_{1,1}^{(n)}$
  $a[j,j] := a[j,j] + h_{2,2}^{(n)}$
  **if** $j > i$
    $a[i,j] := a[i,j] + h_{1,2}^{(n)}$
  **else**
    $a[j,i] := a[j,i] + h_{1,2}^{(n)}$

## Conclusion

When the problem to be solved consists of differential equation (2.11) and the Robin boundary condition (2.16), then the computational procedure for constructing the algebraic system of equations $\boldsymbol{A}\hat{\boldsymbol{u}} = \boldsymbol{b}$ without exploiting symmetry of the coefficient matrix $\boldsymbol{A}$ is as follows:

1. Global assembly, Algorithm 4,
2. Modification for imposing Neumman boundary conditions, Algorithm 8, with $E_1$ replaced by $E$ and $q$ replaced by $(-hu_0)$ ),
3. Modification for imposing Robin boundary conditions, Algorithm 9.

## Exercises

The exercises in this chapter are concerned with boundary value problems defined on a rectangular domain

$$\Omega = \{(x, y) \in R^2 | x_0 \leq x \leq x_0 + L_1\,, \quad y_0 \leq y \leq y_0 + L_2\} \tag{2.49}$$

which is illustrated in Figure 2.5 for the case $(x_0, y_0) = (0, 0)$. The finite element meshes to be used are of the type shown in the same Figure. Note that the mesh in the figure may be viewed as a $4 \times 3$ array of smaller rectangles, each consisting of two triangular elements.

### Matlab arguments

| | |
|---|---|
| `x0`, `y0`: | The coordinates $(x_0, y_0)$ of the lower left point of the domain. |
| `L1`: | Length of the domain $(L_1)$. Interval $x \in [x_0, x0 + L_1]$. |
| `L2`: | Height of the domain $(L_2)$. Interval $y \in [y_0, y_0 + L_2]$ |
| `noelms1`: | Number of rectangles in the $x$-direction. (`noelms1` $= 4$ in Figure 2.5). |
| `noelms2`: | Number of rectangles in the $y$-direction. (`noelms2` $= 3$ in Figure 2.5). |
| `noelms`: | Number of elements. (`noelms` $= 24$ in Figure 2.5). |
| `nonodes`: | Number of nodes. (`nonodes` $= 20$ in Figure 2.5). |
| `VX(1:nonodes)`: | The $x$-coordinates of the nodes. |
| `VY(1:nonodes)`: | The $y$-coordinates of the nodes. |
| `EToV(1:noelms,1:3)`: | The Element To Vertice connectivity table. (See Exercise 2.1). |
| `delta`: | The area of an element. |
| `abc(1:3,1:3)`: | Coefficients of the local basis functions. (See Exercise 2.2). |
| `lam1,lam2`: | Constant values of the functions $\lambda_1(x, y)$ and $\lambda_2(x, y)$. |
| `qt(1:nonodes)`: | Values of $\tilde{q}(x, y)$ at the nodes. |
| `A(1:nonodes,1:nonodes)`: | The global matrix, in Matlab's sparse format. |
| `b(1:nonodes)`: | The global right-hand-side vector. |
| `nobnodes`: | The number of nodes on $\Gamma_2$. (See Exercise 2.4). |
| `bnodes(1:nobnodes)`: | The global node numbers of nodes on $\Gamma_2$ . |
| `f(1:nobnodes)`: | The values of $f(x, y)$ at the nodes on $\Gamma_2$ . |
| `nobeds`: | The number of element edges on $\Gamma_1$ . (See Exercise 2.6). |
| `beds(1:nobeds,1:2)`: | The element edges on $\Gamma_1$ . |
| `q(1:nobeds)`: | The values of $q(x, y)$ at the midpoints of the element edges on $\Gamma_1$ . |

**N.B.** In all test cases that require values of `lam1` and `lam2`, put `lam1`=`lam2`=1.

## Exercise 2.1

Let a finite element mesh be imposed on domain (2.49) of the type shown in Figure 2.5.

a) Write a Matlab routine that computes the arrays `VX(1:nonodes)` and `VY(1:nonodes)`, the $x$ and $y$ coordinates of the mesh nodes. As illustrated in Figure 2.5, the ordering of the nodes should be column-wise, beginning at the left boundary and running downward in each column.

**Head:**
`function [VX,VY] = xy(x0,y0,L1,L2,noelms1,noelms2)`

**Test case:**
$(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = 4`, `noelms2 = 3`.

b) Write a Matlab routine that constructs an Element-To-Vertice array `EToV(1:noelms,1:3)` defining how mesh nodes are connected to define the element of the triangular mesh. `EToV(n,r)` is defined to be the global node number of the node in element $e_n$ with local number $r$. For the mesh in Figure 2.5, for example, the 4'th row of `EToV` should consist of the values `EToV(4,1)` = 3, `EToV(4,2)` = 7 and `EToV(4,3)` = 2.

**Head:**
`function EToV = conelmtab(noelms1,noelms2)`

**Test case:**
`noelms1 = 4`, `noelms2 = 3` (See Figure 2.5).

## Exercise 2.2

For a given finite element mesh we need to be able to compute the geometric information from the mesh data tables when generating the system of equations defining our discrete finite element method.

a) Write a Matlab routine that computes, for a given element $e_n$, the quantities $\Delta$ and $\tilde{a}_i$ , $\tilde{b}_i$ , $\tilde{c}_i$ , $i = 1, 2, 3$ defined by (2.1) and (2.3), respectively. Test output using $n = 9$.

The contents of array `abc` should be arranged as follows:

$$\texttt{abc}(1,1) = \tilde{a}_1\,, \quad \texttt{abc}(1,2) = \tilde{b}_1\,, \quad \texttt{abc}(1,3) = \tilde{c}_1$$

$$\texttt{abc}(2,1) = \tilde{a}_2\,, \quad \texttt{abc}(2,2) = \tilde{b}_2\,, \quad \texttt{abc}(2,3) = \tilde{c}_2$$

$$\texttt{abc}(3,1) = \tilde{a}_3\,, \quad \texttt{abc}(3,2) = \tilde{b}_3\,, \quad \texttt{abc}(3,3) = \tilde{c}_3$$

**Head:**
`function [delta,abc] = basfun(n,VX,VY,EToV)`

b) Write a Matlab routine that computes, for a given element $e_n$ and edge number $k$, the outer normal vector components $n_1$ and $n_2$ as defined by (2.6). The edge number is defined from the `EToV` table, e.g. edge 1 is defined for element $n$ by the vertices `EToV`($n$,1:2), edge two by `EToV`($n$,2:3) and edge three by `EToV`($n$,[3 1]).Test output using $n = 9$.

**Head:**
`function [n1,n2] = outernormal(n,k,VX,VY,EToV)`

**Test case:**
$(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = 4`, `noelms2 = 3`.

## Exercise 2.3

a) Write a Matlab routine that implements the assembly process described by Algorithm 4 but stores *all* nonzeros of the global matrix $A$, not just those in the upper triangle. (This makes it possible to use later Matlab's very efficient equation solver command `u = A\b`). Assume that $\lambda_1(x, y)$ and $\lambda_2(x, y)$ are constants (`lam1` and `lam2`) and that $\tilde{q}(x, y)$ is defined at the mesh nodes by array `qt(1:nonodes)`. Take the value of $\tilde{q}$ in (2.28) to be the average of $\tilde{q}(x, y)$ at the three nodes of the element.

b) For each test case use the Matlab command `[B,d] = spdiags(A)` to print `A`. In case 2 print also `b`.

**Head:**
`function [A,b] = assembly(VX,VY,EToV,lam1,lam2,qt)`

Array `A(1:nonodes,1:nonodes)` should contain the assembled matrix in Matlab's sparse format, and array `b(1:nonodes)` should contain the assembled right-hand-side vector.

**Test cases:**

1. $(x_0, y_0) = (0, 0)$, $L_1 = 1$, $L_2 = 1$, `noelms1` $= 4$, `noelms2` $= 3$, $\tilde{q}(x, y) = 0$.
2. $(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = 4`, `noelms2 = 3`, $\tilde{q}(x, y) = -6x + 2y - 2$.

## Exercise 2.4

a) Write a Matlab routine based on Algorithm 6, i.e. in a version that does *not* exploit symmetry, that updates arrays $A$ and $b$ so as to impose the Dirichlet boundary condition

$$u = f(x, y)\,, \qquad (x, y) \in \Gamma_2$$

where $\Gamma_2$ is part, or all, of the boundary $\Gamma$. (The case where $\Gamma_2$ is only part of $\Gamma$ arises in problem (2.32), (2.33)). The number of nodes on $\Gamma_2$ is `nobnodes`. The global node numbers of the nodes on $\Gamma_2$ are given by array `bnodes(1:nobnodes)`, and the values of $f(x, y)$ at these nodes are given by array `f(1:nobnodes)`.

b) The test cases in this exercise continue those of Exercise 2.3. Call `dirbc` after `assembly`. Use command `[B,d] = spdiags(A)` to print `A`, and print also `b`.

**Head:**
`function [A,b] = dirbc(bnodes,f,A,b)`

**Test cases:** Let $\Gamma_2 = \Gamma$.

1. $(x_0, y_0) = (0, 0)$, $L_1 = 1$, $L_2 = 1$, `noelms1` $=$ `noelms2` $= 4$,
   $\tilde{q}(x, y) = 0$, $f(x, y) = 1$.
2. $(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = 4`, `noelms2 = 3`,
   $\tilde{q}(x, y) = -6x + 2y - 2$, $f(x, y) = x^3 - x^2y + y^2 - 1$.

## Exercise 2.5

a) Use the Matlab routines of the previous exercises to write a program that solves the boundary value problem

$$u_{xx} + u_{yy} = -\tilde{q}(x, y), \qquad (x, y) \in \Omega \tag{2.50}$$

$$u = f(x, y), \qquad (x, y) \in \Gamma \tag{2.51}$$

where $\Omega$ is the rectangular domain given by (2.49) and $\Gamma$ is its boundary.

An easy but effective way to test the program is the following:

- Choose a simple function $u(x, y)$.
- Determine analytically the functions $\tilde{q}(x, y)$ and $f(x, y)$ that make $u(x, y)$ the solution of problem (2.50), (2.51); i.e., define

$$\tilde{q}(x, y) = -(u_{xx} + u_{yy}), \quad (x, y) \in \overline{\Omega}, \qquad f(x, y) = u(x, y), \quad (x, y) \in \Gamma$$

- Solve problem (2.50), (2.51) using the program.
- Compute the error in the finite element solution. A suitable measure of the error is

$$E = \max_{1 \le j \le M} |\hat{u}_j - u(x_j, y_j)| \tag{2.52}$$

  where $\hat{u}_j$ is the finite element solution at node $(x_j, y_j)$ and $M$ is the number of nodes ($M =$ `nonodes`).
- Produce a *convergence plot* where measured errors versus mesh size are plotted. Make sure to employ sufficiently small mesh sizes to clearly illustrate the *asymptotic* behavior of the error. It is a good idea to include a line in the plot which clearly shows the *expected* (from theory) behavior of the method tested.

**Test cases:**

1. $u(x, y) = x^3 - x^2y + y^2 - 1$,
   $(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = 4`, `noelms2 = 3`.

   a) Print the solution values $\hat{u}_j$ and the error $E$.
2. $u(x, y) = x^2y^2$,

   $(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$,

   `noelms1` $=$ `noelms2` $= 2^p$, $p = 1, 2, 3, \ldots$

   a) Compute $E$ for each value of p.

   b) $E$ is a function of $h$, where $h$ is the largest element edge length in the mesh.
   On the basis of the results from a), describe this function for small values of h.

## Exercise 2.6

a) Write a Matlab routine that constructs an array for holding information about boundary edges (`beds`) of the triangular mesh. (HINT: the use of a distance function `fd` can make the job of finding nodes that belong to a boundary implicitly defined by the distance function very easy. Below it is included in the argument list as an optional argument. `top` is a tolerance to be used with the signed distance function to tune the selection of nodes.)

**Head:**
```
function [beds] = ConstructBeds(VX,VY,EToV,tol[,fd])
```

b) Write a Matlab routine that updates the global right-hand-side vector stored in array `b` so as to impose the Neumann boundary condition

$$\lambda_1 u_x n_1 + \lambda_2 u_y n_2 = -q(x,y)\,, \qquad (x,y) \in \Gamma_1$$

The computation required is given in Algorithm 8.

**Head:**
```
function b = neubc(VX,VY,EToV,beds,q,b)
```

`VX`, `VY` and `EToV` are the same as before. Introduce `nobeds` as the number of element edges on $\Gamma_1$. Then an array `beds(1:nobeds,1:2)` is a list of these element edges. Each row ($p$) of `beds` identifies an element edge as follows: $n$ = `beds(p,1)` is the number of the element that contains the edge. $r$ = `beds(p,2)` is the local node number of the first node on the edge. (See Section 2.9 for the definition of "first node").

The computation requires also the second node on the edge. If the local number of the second node is denoted $s$, then

$$r{=}1 \implies s{=}2\,, \quad r{=}2 \implies s{=}3\,, \quad r{=}3 \implies s{=}1$$

assuming the three nodes are ordered counterclockwise.

Array `q(1:nobeds)` contains the values of the function $q(x,y)$ at the midpoints of the element edges.

## Exercise 2.7

Let $\Gamma_1$ denote the left and bottom edges of the rectangular domain (2.49), and let $\Gamma_2$ denote the right and top edges.

a) Use the Matlab routines of the previous exercises to write a program that solves the boundary value problem

$$u_{xx} + u_{yy} = -\tilde{q}(x,y)\,, \quad (x,y) \in \Omega$$

$$u_n = -q(x,y)\,, \quad (x,y) \in \Gamma_1\,, \qquad u = f(x,y)\,, \quad (x,y) \in \Gamma_2$$

where $u_n$ is the outward-directed normal derivative of $u$. Since we have here $\lambda_1(x,y) = \lambda_2(x,y) = 1$, the boundary condition on $\Gamma_1$ is the Neumann condition (2.32). Note that $u_n = -u_x$ on the left edge of the boundary and $u_n = -u_y$ on the bottom edge.

**Test cases**
These test cases are based on the testing procedure described in Exercise 2.5. The functions to be derived analytically from the given solution $u$ are now $\tilde{q}$, $q$ and $f$. The error $E$ is defined in (A.40).

1. $u(x,y) = 3x + 5y - 7$,

   $(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = 4`, `noelms2 = 3`.

   a) Print the solution values $\hat{u}_j$ in the 2-D array format, where the solution array is allocated as `u = zeros(noelms2+1,noelms1+1)`. Print also the error $E$.

2. $u(x,y) = \sin(x)\sin(y)$,

   $(x_0, y_0) = (-2.5, -4.8)$, $L_1 = 7.6$, $L_2 = 5.9$, `noelms1 = noelms2 = 32`.

   a) Print $E$.

## Exercise 2.8

The Neumann boundary condition is often used to exploit symmetry in the solution of a boundary value problem. Consider, for example, the problem

$$u_{xx} + u_{yy} = -2\pi^2 \cos(\pi x)\cos(\pi y)\,, \quad -1 \le x, y \le 1 \tag{2.53}$$

$$u(x,y) = \cos(\pi x)\cos(\pi y)\,, \quad (x,y) \in \Gamma \tag{2.54}$$

where $\Gamma$ is the boundary of the domain in (2.53). The domain as well as the function on the right-hand side of (2.53) are symmetric about both the $x$ and $y$ axes. The solution $u(x,y)$ inherits these symmetry propeties, so it is sufficient to determine $u(x,y)$ on a quarter of the domain.

a) Formulate a boundary value problem for $u(x,y)$ on the subdomain $0 \le x, y \le 1$.

b) Use your Matlab software to solve this problem. Let `noelms1` = `noelms2` = 3 and print the solution in a 2-D format.

c) Use your Matlab software to solve the original problem, (2.53), (2.54). Let `noelms1` = `noelms2` = 6, and print the solution in a 2-D format.

d) Both programs compute an approximation to the value $u(0,0)$. Compute these approximations for the following meshes:

   Program b): `noelms1` = `noelms2` = $2^p$, $p = 1, 2, \ldots, 6$.

   Program c): `noelms1` = `noelms2` = $2^{p+1}$, $p = 1, 2, \ldots, 6$.

   Note that for the same value of $p$, the element dimensions in the two cases are identical; i.e., the two meshes have the same fineness.

# Chapter 3

# Spectral/$hp$-FEM in one space dimension

In the foregoing chapters the classical FEM has been used to understand the basic ideas of the formulation and numerical implementation with the purpose of solving partial differential equations. The aim of this chapter is to take the next step, where we will generalize the basic ideas such that we can choose both the number of elements and the order of the polynomial approximations over the elements. This will turn the FEM method into an arbitrarily high-order accurate Spectral/$hp$-FEM method, also referred to as a Spectral Element Method (SEM) in the original work [43]. In this process, we will also aim for a generic implementation of the methodology where the discretization parameters are not hard-coded (i.e. with fixed tabulated parameters) anywhere in the algorithms.

The spectral element method received its name because theoretical analysis shows that when the element interpolation nodes are distributed over an element to achieve the highest interpolation accuracy, then the interior nodes have to be distributed at positions corresponding to the zeros of certain families of orthogonal polynomials. Therefore the position and number of the local interpolation nodes defines the spectral element expansion basis and the properties of the associated spectral element method.

An important property of the Spectral/$hp$-FEM is that there are two basic paths to convergence, namely, $h$- and $p$-adaptivity. In $h$-adaptivity the mesh is adjusted while the types of all elements are kept fixed, i.e. on each local element the polynomial order is constant. In $p$-adaptivity the number of elements is kept fixed while convergence is achieved by increasing the polynomial order on each element. An optimal convergence strategy can be a combination of both basic types of adaptivity and is called $hp$-adaptivity. The word "spectral" used in connection with the Spectral/$hp$-FEM implies that with the proper choice of basis functions it is possible to reduce the numerical errors exponentially fast, i.e. faster than any fixed algebraic convergence rate. This high-order property of the Spectral/$hp$-FEM can be exploited for doing efficient computations and enhancing the accuracy of a solution on a given mesh.

For more in-depth theory the reader is referred to [36, 45] together with [34] which covers implementation details and fundamental analysis of the spectral element methods and one of it's cousins, namely, the nodal Discontinuous Galerkin Method (DG-FEM).

## 3.1 The Spectral/$hp$-Finite Element mesh

Consider some closed spatial domain of interest, which is represented by some discrete domain $\bar{\Omega}$. As usual, the starting point is to cover this domain with $N$ non-overlapping elements $e_n$,

$n = 1, 2, ..., N$. Each element can then be covered by a set of $M_P = P + 1$ nodes making it possible to represent the global solution $u(x)$ by some approximate solution $\hat{u}(x)$ constructed from polynomials of order at most $P$ over each element. The total number of unique points in the computational domain $\bar{\Omega}$ is denoted by $M$. An example of a one-dimensional mesh is illustrated in Figure 3.1. The choice of node position turns out to be important for numerical stability and accuracy and we shall return to this subject in Section 3.2.

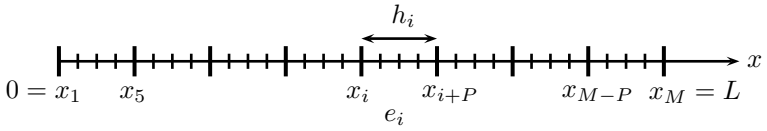



Figure 3.1: Sketch and notation for a one-dimension mesh consisting of elements with a local fourth order ($P = 4$) polynomial representing the solution on each element.

## 3.2 Spectral/*hp*-Finite Element Basis functions

A function $u(x)$ is represented by piece-wise polynomial functions of the form

$$u(x) \cong \hat{u}(x) = \sum_{j=1}^{M} \hat{u}_j N_j(x), \quad x \in \bar{\Omega} \tag{3.1}$$

where $N_i(x)$ is a global finite element basis function defined such that it possess the Cardinal property (see Section 1.2) for the $i$'the node and vanishes for all other mesh nodes

$$N_i(x_j) = \delta_{i,j} \tag{3.2}$$

Each of the global basis functions is defined on the entire domain of interest, but have only local support over the element. This implies that in one space dimension each function $N_j(x)$ is nonzero on at most two adjacent elements.

As described in Section 1.2, it is possible to represent each of the global basis functions $N_i(x)$ in terms of local basis functions defined on the elements only in the form

$$N_i(x) = \bigoplus_{n=1}^{N} \sum_{j=0}^{P} N_j^{(n)}(x_i) N_j^{(n)}(x) \tag{3.3}$$

where $N_j^{(n)}(x)$ is a local basis function (e.g. in the form of a Lagrange polynomial) belonging to the $n$'th element and defined from the set of nodes that have been associated with it. The direct sum operator $\oplus$ has been introduced and is a mathematical symbol for the summation of disjoint intervals, which can be thought of as patching the different elements together. For a general mesh consisting of $N$ elements, the global basis functions are defined as

$$N_i(x) = \bigoplus_{n=1}^{N} \sum_{j=0}^{P} N_j^{(n)}(x_i) N_j^{(n)}(x) = \begin{cases} \sum_{j=0}^{P} N_j^{(1)}(x_i) N_j^{(1)}(x) & , x \in e_1 \\ \sum_{j=0}^{P} N_j^{(2)}(x_i) N_j^{(2)}(x) & , x \in e_2 \\ \quad \vdots & \\ \sum_{j=0}^{P} N_j^{(n)}(x_i) N_j^{(n)}(x) & , x \in e_N \end{cases} \tag{3.4}$$

Based on the equivalent expressions (3.3) and (3.4) it should be clear that over each element we can represent the solution using arbitrary order polynomials. Examples of global basis functions are illustrated in Figure 3.2. To represent the solution over an element, the idea is to use a $P$'th

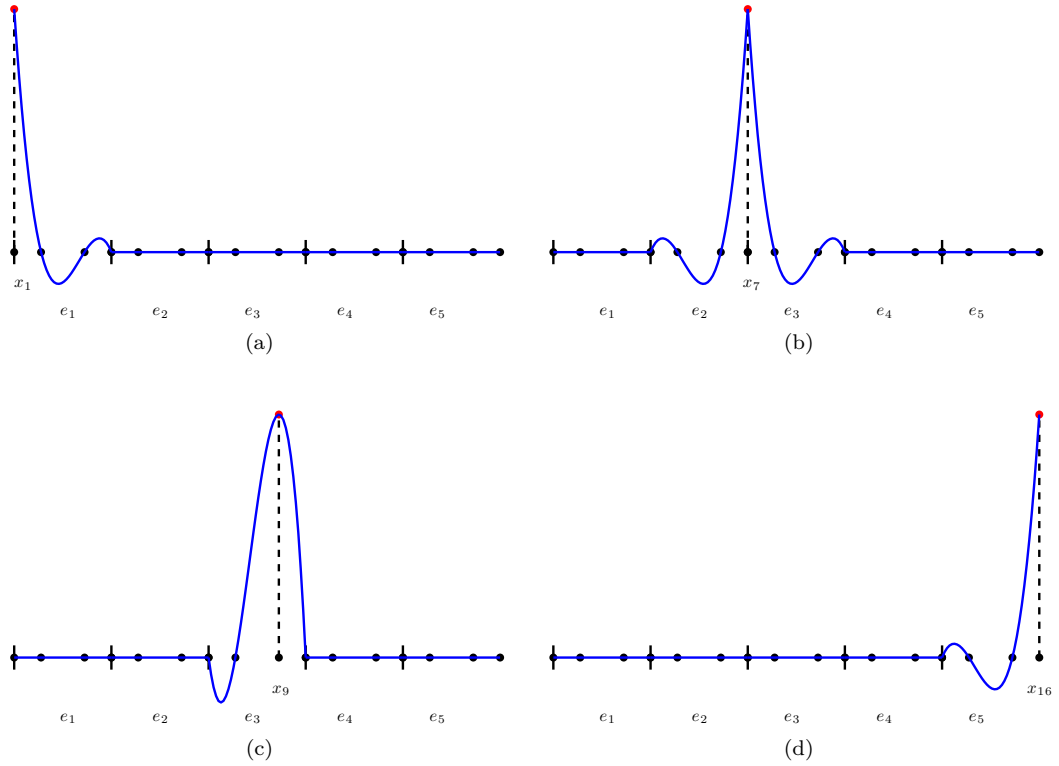



Figure 3.2: Different global nodal basisfunctions with Cardinal property at nodes a) $x_1$, b) $x_7$, c) $x_9$ and d) $x_{16}$ on a mesh consisting of 5 elements with a total of 16 global nodes. Local polynomial order on each element is four. The nodal basis functions in a), b) and c) are also referred to as vertex nodal basis functions. The basis function in c) is also referred to as a bubble (interior) basis function.

order polynomial. Such a polynomial is defined from $P+1$ distinct interpolation nodes, which leaves $P-1$ interior interpolation nodes at each element. For the Spectral/*hp*-element method we need to include nodes on the element boundaries to ensure at least $C^0$-continuity for the global interpolating functions $N_i(x)$, $i = 1, .., M$.

It is convenient computationally, to map each of the $N$ element subdomains to the standard element defined on the interval $r \in [-1, 1]$ using an affine map

$$x(r) = \tfrac{1}{2}(b+a) + \tfrac{1}{2}(b-a)r, \quad x \in [a, b] \tag{3.5}$$

We will see that this makes it possible to only store a single set of local operators that can be reused on all (or most) elements. The obvious advantage is that this reduce the necessary storage requirements for the algorithm.

The $n$'th interpolating Lagrange polynomial $N_n(r)$ on the standard element is the unique polynomial of order $P$ with the Cardinal property

$$N_n(r_i) = \delta_{n,i} \tag{3.6}$$

Thus, introducing the interpolation operator $\mathcal{I}_P$ which interpolates a function $f(r)$ at the collocation nodes, we can form an interpolating polynomial of order $P$

$$\mathcal{I}_P f(r) = \sum_{i=0}^{P} f(r_i) N_i(r) \tag{3.7}$$

The uniqueness of each of the Lagrange polynomials $N_i(r)$ follows by choosing a set of $P+1$ distinct nodes on the interval of the domain.

The $P$'th degree interpolating Lagrange polynomial's is defined from

$$N_n(r) = \frac{\prod_{i=0,i\neq n}^{P}(r-r_i)}{\prod_{j=0,j\neq i}^{P}(r_i-r_j)}, \quad n=0,...,P \tag{3.8}$$

An equivalent representation is

$$N_n(r) = \frac{\Phi_P(r)}{(r-r_n)\Phi'_P(r_n)} \tag{3.9}$$

where the Lagrange generating polynomial $\Phi_P(x)$ is defined in terms of all $P+1$ interpolation nodes as

$$\Phi_P(r) \equiv \prod_{i=0}^{P}(r-r_i) \tag{3.10}$$

Having introduced the interpolating polynomial (3.7) it is possible to determine the $q$'th derivative of the interpolating polynomial by direct differentiation

$$\frac{d^q}{dr^q}\mathcal{I}_P f(r) = \sum_{i=0}^{P} f(r_i)\frac{d^q N_i(r)}{dr^q} \tag{3.11}$$

Thus, at each of the interpolation nodes, it is possible to determine the discrete set of first derivatives as a matrix-vector product of the form

$$\mathbf{u}'_P = \mathcal{D}\mathbf{u}_P \tag{3.12}$$

where the vectors holds respectively the values of the function and its derivatives at the $P+1$ collocation points

$$\mathbf{u}_P = [u_0 u_1 \cdots u_P]^T, \quad \mathbf{u}'_P = [u'_0 u'_1 \cdots u'_P]^T \tag{3.13}$$

and we have introduced a $(P+1)\times(P+1)$ nodal differentiation matrix for the first derivative defined as

$$\mathcal{D}_{i,j} \equiv \left(\frac{dN_i}{dr}\right)_{r=r_j} \tag{3.14}$$

where the derivatives of the Lagrange polynomials are found by differentiation of (3.9) such that

$$\frac{dN_i(r)}{dr} = \frac{\Phi'_P(r)(r-r_i)-\Phi_P(r)}{(r-r_i)^2\Phi'_P(r_i)} \tag{3.15}$$

Thus, we can show that the nodal differentiation matrix is defined from

$$\mathcal{D}_{i,j} \equiv \left(\frac{dN_i}{dr}\right)_{r=r_j} = \begin{cases} \frac{\Phi'_P(r_j)}{(r_j-r_i)\Phi'_P(r_i)} & , \quad i\neq j \\ \frac{\Phi''_P(r_i)}{2\Phi'_P(r_i)} & , \quad i=j \end{cases} \tag{3.16}$$

where the last expression is the limiting expression for $r_i \to r_j$ found using l'Hôpital's rule. It is possible to construct this matrix directly once the distribution of distinct nodes associated with the standard element has been chosen although it is not computationally the most efficient way. We shall see, there is more convenient and generic way of constructing this matrix exactly.

## Nodal or modal representation

For representing the local solution on the elements, we can make a choice, namely, whether the representation is nodal or modal. We have already introduced a nodal polynomial (3.7) which interpolates the function values on the mesh nodes directly. However, in close resemblance with the large class of spectral methods, we can choose to represent our local solution in the form

$$u(r) = \sum_{n=0}^{P} \hat{u}_n \psi_n(r) \tag{3.17}$$

where $\psi_n$, $n = 0, ..., P$ is a set of basis functions. The type of these basis functions can be either a Lagrange polynomial or some other basis on the mesh. If the basis functions are Lagrange polynomials then the coefficients $\hat{u}_n$ corresponds to the interpolated values of the represented solution at the collocation nodes as required by (3.6). This type of basis is referred to as a nodal basis. It is customary to choose either the Lagrange polynomials (nodal) or some family of orthogonal polynomials for representing the solution (modal) similar to spectral methods. Possible choices of basis functions in one space dimension is illustrated in Figure 3.3.

There exists a unique similarity transformation between the nodal and modal representations, and this can be expressed as

$$\mathcal{V}\hat{\mathbf{u}} = \mathbf{u} \tag{3.18}$$

where $\mathcal{V}$ is a generalized Vandermonde matrix defined as

$$\mathcal{V}_{i,j} \equiv \psi_{j-1}(\xi_i), \quad i, j = 1, ..., M_P \tag{3.19}$$

where $M_P$ is the total number of points for an element. The Vandermonde matrix can be used for implementing the Spectral/*hp*-FEM method. However, the choice of basis functions and nodes is important for the accuracy and numerical stability of the method. For optimal approximation properties it is important to identify those points that minimize the Lebesque constant (see Section E), since selecting non-optimal points can result in ill-conditioning of the Vandermonde matrix and impact the accuracy of the computed solutions.

It can be shown that the optimal distribution of points in one space dimensions for optimal approximation properties are the so-called $P+1$ Legendre-Gauss-Lobatto (LGL) points which can be determined in Matlab using the command

```
>> r  = JacobiGL(0,0,P);
```

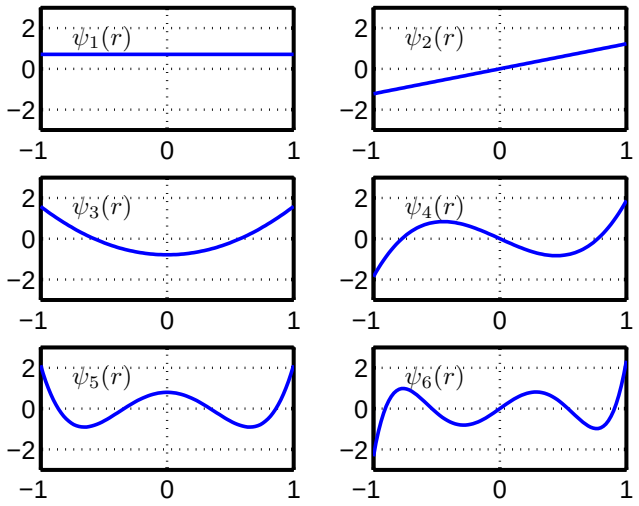
This set of points has end nodes at the interval $r \in [-1, 1]$ and has the interior points clustered quadratically near the end nodes for large $P$.

Furthermore, the generalized Vandermonde matrix corresponding to the choice of reference basis and collocation nodes $r_i$, $i = 0, ..., P$, can be computed using the Matlab command

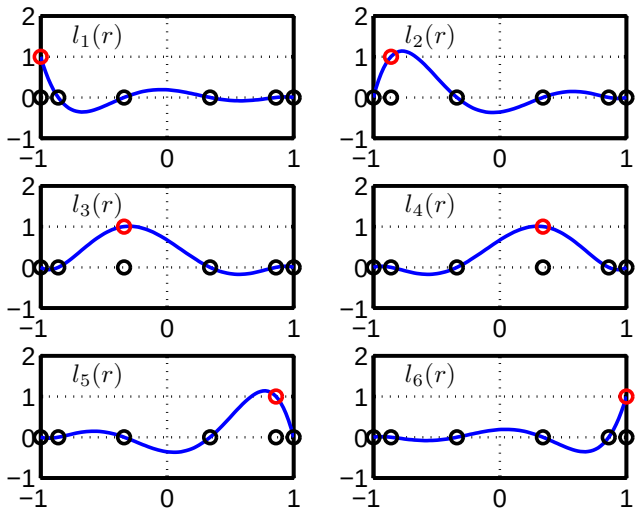
```
>> V = Vandermonde1D(P,r);
```

The `Vandermonde1D` routine computes the generalized Vandermonde matrix based on the Legendre polynomials, which belong to the family of Jacobi polynomials (some details on Jacobi polynomials can be found in Appendix C). The Legendre polynomials form an orthogonal basis in $L^2([-1, 1])$ and can be constructed from the monomial polynomials through a Gram-Schmidt orthogonalization process. The class of orthogonal Jacobi polynomials can be generated explicitly through recurrence relations up to very high orders and form complete sets, which make them highly suitable for use in finite element software as the local element basis.

The Vandermonde matrix can also be used for interpolation to other node distribution sets in the following way




(a)




(b)

Figure 3.3: Illustration of the a) modal Legendre polynomials and b) nodal Lagrange polynomials up to order $P = 6$ on the interval $r \in [-1, 1]$.

```
>> Vnew = Vandermonde1D(P,rnew);
>> unew = Vnew*(V \ u);
```

and thereby makes it possible to represent the solution at any set of new points (`rnew`) in the reference interval.

The auxiliary routines described in this section can be found in Appendix F.

## 3.3 Interpolation

The Spectral/*hp*-finite element basis functions can be used for interpolation. For a function $u(x)$ defined on the interval $0 \leq x \leq L$, let us construct an interpolating function using the global finite element basis functions $N_i(x)$ as

$$u_I(x) = \sum_{i=1}^{M} u(x_i) N_i(x), \quad 0 \leq x \leq L \tag{3.20}$$

where $N_i(x)$ are defined in (3.2) for the given mesh consisting of a total of $M$ interpolation nodes. Since each $N_i(x)$ is constructed piecewise using arbitrary order polynomials with the requirement that the basis functions are globally continuous, i.e. $N_i(x) \in C^0$, so is the interpolating function $u_I(x)$. Further, it follows from the Cardinal property (3.2) that

$$u_I(x) = u(x_i), \quad i = 1, 2, ..., M \tag{3.21}$$

which ensures that the interpolating function $u_I(x)$ interpolated $u(x)$ at each of the mesh nodes. Thus, the interpolating function can be used to represent the discrete function everywhere in the region defined by the mesh.

Regarding the interpolation error $\epsilon(x) = u_I(x) - u(x)$, it can be shown [20] that if $u(x)$ is sufficiently differentiable, i.e. $u(x) \in C^{P+1}$, then on each of the $N$ elements $i = 1, 2, ..., N$ the error can be bounded from above as

$$\max_{x \in e_i} |\epsilon(x)| \leq C_i h_i^{P+1} \max_{x \in e_i} |u^{(P+1)}(x)| \tag{3.22}$$

where $u^{(P+1)}(x)$ denotes the $(P+1)$'th derivative of $u(x)$. These are "local" error bounds with $C_i$ being some constants dependent on the location of the interpolation nodes in the interior of the elements. It can be shown that if the positions of the interpolation nodes are chosen to be equidistant on each element, then the use of high-order polynomials for the interpolation can lead to *Runge's phenomenon* where the interpolating polynomials exhibit dramatic oscillations in between the interpolation nodes. Thus, to avoid potential problems of this kind, we as a general rule of thumb need to use low orders of polynomials if we employ equi-distant nodes and otherwise use nodes that are clustered near the ends of the elements, e.g. a LGL node distribution.

The local bounds (3.22) lead directly to the "global" bound for the interpolation error

$$\max_{0 \leq x \leq L} |\epsilon(x)| \leq C h^{P+1} \max_{0 \leq x \leq L} |u^{(P+1)}(x)| \tag{3.23}$$

where the largest element length $h$ is determined from

$$h = \max_i h_i \tag{3.24}$$

and the constant from

$$C = \max_i C_i \tag{3.25}$$

Sometimes there is given a set of values $w_i$, $i = 1, 2, ..., M$, and one want to construct a continuous function $w(x)$, $x \in [0, L]$ such that it interpolates the values exactly

$$w(x_i) = w_i, \quad i = 1, 2, ..., M. \tag{3.26}$$

This can be achieved by taking $w(x)$ as the continuous, piecewise-polynomial function

$$w(x) = \sum_{i=1}^{M} w_i N_i(x), \quad 0 \leq x \leq L \tag{3.27}$$

## 3.4 The Spectral/*hp*-Finite Element Method

To formulate a Spectral/*hp*-finite element method we follow the same procedure laid out in Section 1.4 for the classical FEM. The main difference is a change in the representation of the global solution, since for the Spectral/*hp*-finite element method we should be able to choose arbitrarily high-order polynomials as the local basis over each element.

We will illustrate the method for solving the boundary value problem for a second-order differential equation for one space variable (1.14) with boundary conditions (1.15) as stated in Section 1.4.

The solution $\hat{u}(x)$ approximating the unknown solution $u(x)$ will be represented by a function which is globally continuous but has first-order derivatives with jump discontinuities at the nodes. Therefore, it is necessary to reduce the continuity requirement by removing the second-order derivative from the formulation by suitably rewriting the problem into a so-called weak formulation. This is done by multiplying both sides of the differential equation by a function $v(x)$ which satisfies the homogenous Dirichlet boundary conditions at the domain boundaries

$$v(0) = v(L) = 0 \tag{3.28}$$

and then integrate over the interval $0 \leq x \leq L$ and apply integration by parts. Thus, we find using (3.28) the weak formulation

$$\int_0^L (u'v' + uv)dx = 0 \tag{3.29}$$

To apply the finite element method to this weak formulation we choose to represent the global approximate solution in the form (3.1). Now, we need a way to determine the unknown coefficients $u_i$. Two of these coefficients are determined using the boundary conditions as

$$\hat{u}_1 = c, \quad \hat{u}_M = d \tag{3.30}$$

Thus, it remains to determine the $M-2$ unknown coefficients to determine an approximate solution to the differential equation. This is done using the weak formulation (3.29) by substituting $\hat{u}(x)$ for $u(x)$ and then putting the test function $v(x)$ equal to each of the global basis functions $N_i(x)$, $i = 2, ..., M-1$. This makes it possible to generate sufficient number of equations to be able to determine the unknown coefficients. Note that each of these global basis functions vanishes at the endpoints of the interval as required to be able to use the weak formulation (3.29).

By inserting (3.1) into (3.29) we find

$$\int_0^L \left( \sum_{i=1}^{M} \hat{u}_i \frac{dN_i}{dx}\frac{dN_j}{dx} + \sum_{i=1}^{M} \hat{u}_i N_i N_j \right) dx = 0, \quad j = 2, ..., M-1 \tag{3.31}$$

or equivalently

$$\sum_{i=1}^{M} \hat{u}_i \int_0^L \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx = 0, \quad j = 2, ..., M-1 \tag{3.32}$$

Thus, these equations can be collected to produce a linear system of the form

$$\mathbf{A}\hat{\mathbf{u}} = \mathbf{b} \tag{3.33}$$

where the elements of the coefficient matrix $\mathbf{A}$ is defined from

$$a_{i,j} = \int_0^L \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx, \quad i = 2, ..., M-1 \tag{3.34}$$

for the interior coefficients. Due to the boundary conditions (3.30) the remaining coefficients are

$$a_{1,j} = \delta_{1,j}, \quad a_{M,j} = \delta_{M,j} \tag{3.35}$$

Since each of the global basis functions has local support over at most two adjacent elements it is possible to reduce the integration limits of the integrals in (3.34). For global basis functions with support only in the same element, e.g. the $n$'th element, we find

$$a_{i,j} = k_{i,j}^{(n)}, \quad k_{i,j}^{(n)} \equiv \int_{e_n} \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx \tag{3.36}$$

and for global basis functions with nonzero support at the boundary of two neighboring elements, e.g. the $k$'th and $q$'th elements, we find

$$a_{i,j} = k_{i,j}^{(k)} + k_{i,j}^{(q)} = \int_{e_k} \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx + \int_{e_q} \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx \tag{3.37}$$

consisting of two integral contributions.

The corresponding right hand side vector $\mathbf{b} = (b_1, b_2, ..., b_M)^T$ for the linear system (3.33) is defined from

$$\begin{aligned} b_1 &= c, \quad b_M = d, \\ b_i &= 0, \quad i = 2, ..., M-1 \end{aligned} \tag{3.38}$$

Thus, we have defined a system of $M$ linear equations in $M-2$ unknowns.

Note how the procedure described earlier in Section 1.4 for the classical FEM method has now been generalized. The main difference is in the way the global basis functions are defined. Ultimately this affects the way we determine the integrals in the coefficients (3.34) as defined in (3.36) and (3.37).

## 3.5 Element matrices

It is possible to express the global solution as a direct sum of the local solutions only defined over the elements as

$$u(x) \cong \hat{u}(x) = \sum_{i=1}^{M} \bigoplus_{n=1}^{N} \sum_{j=0}^{P} \hat{u}_i N_j^{(n)}(x_i) N_j^{(n)}(x), \quad x \in \bar{\Omega} \tag{3.39}$$

which is found by combining (3.1) and (3.3).

The expression for the global basis functions (3.3) defines the connection between the local and the global basis functions. This will be exploited in the following for determining the element contributions to the operators of the linear system (3.33).

We also need to define the connection between the first derivative of the global basis functions and the local basis functions. This can be found by differentiation of (3.3) and we find

$$\frac{dN_i(x)}{dx} = \bigoplus_{n=1}^{N} \sum_{j=0}^{P} N_j^{(n)}(x_i) \frac{dN_j^{(n)}(x)}{dx} \tag{3.40}$$

If we insert (3.3) and (3.40) into (3.34) we find

$$a_{i,j} = \bigoplus_{n=1}^{N} \int_{e_n} \left[ \left( \sum_{m=0}^{P} N_m^{(n)}(x_i) \frac{dN_m^{(n)}(x)}{dx} \right) \left( \sum_{m=0}^{P} N_m^{(n)}(x_j) \frac{dN_m^{(n)}(x)}{dx} \right) + \right. \tag{3.41}$$

$$\left. \left( \sum_{m=0}^{P} N_m^{(n)}(x_i) N_m^{(n)}(x) \right) \left( \sum_{m=0}^{P} N_m^{(n)}(x_j) N_m^{(n)}(x) \right) \right] dx \tag{3.42}$$

We then have three different cases

1. $x_i$ and $x_j$ belongs to two different elements which are not neighbours.
2. $x_i = x_j$ and they both belong to the interior of the same element.
3. $x_i$ and $x_j$ belongs to the boundary of the same element.

In the first case there will be no contributions to the integral since the integrands of each element will be zero, since at least one of the local interpolating polynomials will be zero in any elements.

In the second case, there will be one contribution from a single element, e.g. the $n$'th element, and the coefficient (3.44) will simplify to

$$a_{i,j} = \int_{e_n} \left[ \left( \sum_{m=0}^{P} N_m^{(n)}(x_i) \frac{dN_m^{(n)}(x)}{dx} \right) \left( \sum_{m=0}^{P} N_m^{(n)}(x_j) \frac{dN_m^{(n)}(x)}{dx} \right) + \left( \sum_{m=0}^{P} N_m^{(n)}(x_i) N_m^{(n)}(x) \right) \left( \sum_{m=0}^{P} N_m^{(n)}(x_j) N_m^{(n)}(x) \right) \right] dx \tag{3.43}$$

In the third case there will be two contributions to the $n$'th element, and (3.44) will simplify to

$$a_{i,j} = k_{i,j}^{(k)} + k_{i,j}^{(q)}, \quad x_i, x_j \in e_k \cap e_q, \tag{3.44}$$

for any pair of global basis functions which are nonzero at the same boundary edge of the $n$'th element, having assumed that this edge is shared between two neighboring elements $e_p$ and $e_q$.

Each of the elements of the coefficient matrix $\mathbf{A}$ is seen to have local contributions of the form

$$\mathcal{M}_{i,j}^{(n)} = \int_{e_n} N_i^{(n)}(x) N_j^{(n)}(x) dx, \quad \mathcal{K}_{i,j}^{(n)} = \int_{e_n} \frac{dN_i^{(n)}(x)}{dx} \frac{dN_j^{(n)}(x)}{dx} dx \tag{3.45}$$

where $\mathcal{M}_{i,j}^{(n)}$ is referred to as a *element mass matrix* and $\mathcal{K}_{i,j}^{(n)}$ a *element stiffness* (or *diffussion*) *matrix* for the $n$'th element. The sum of these two matrices

$$k_{i,j}^{(n)} = \mathcal{M}_{i,j}^{(n)} + \mathcal{K}_{i,j}^{(n)} \tag{3.46}$$

is equivalent to the expression (1.27) for one element defined in terms of a local basis with polynomial order corresponding to $P = 1$.

These contribution can be collected in standard element matrices for the elements and it will be shown in the following how we can compute these with exact integration without sorting to a numerical quadrature rule (cf. Appendix C). Instead we will exploit that we can represent the local polynomials in both modal and nodal forms and then use the analytical properties of a set of orthogonal modal basis functions to compute the standard element matrices.

The element mass matrix can be determined from the generalized Vandermonde matrix. The $i$'th local basis functions represented in terms of a Lagrange polynomial can be expressed as

$$N_i(r) = \sum_{n=1}^{M_P} \left(\mathcal{V}^T\right)_{i,n}^{-1} \psi_n(r) \tag{3.47}$$

Insert into an element of the *standard element mass matrix*

$$\begin{aligned} \mathcal{M}_{i,j} &= \int_{-1}^{1} \sum_{n=1}^{M_P} \left(\mathcal{V}^T\right)_{i,n}^{-1} \psi_n(r) \sum_{m=1}^{M_P} \left(\mathcal{V}^T\right)_{j,m}^{-1} \psi_m(r) dr \\ &= \sum_{n=1}^{M_P} \sum_{m=1}^{M_P} \left(\mathcal{V}^T\right)_{i,n}^{-1} \left(\mathcal{V}^T\right)_{j,m}^{-1} (\psi_n, \psi_m)_I \\ &= \sum_{n=1}^{M_P} \left(\mathcal{V}^T\right)_{i,n}^{-1} \left(\mathcal{V}^T\right)_{j,n}^{-1} \end{aligned} \tag{3.48}$$

which is equivalent to

$$\mathcal{M} = \left(\mathcal{V}\mathcal{V}^T\right)^{-1} \tag{3.49}$$

For a general straight-sided element in one spatial dimension this enables us to form the mass matrix for the $n$'th element as

$$\mathcal{M}^{(n)} = \frac{h_n}{2}\mathcal{M} \tag{3.50}$$

Where the factor $h_n/2$ is the Jacobian of the transformation of the integral over the element space to the reference space (this type of transformation is defined in (C.2)).

To determine the *element differentiation matrix*, we choose to express the derivative of the $j$'th Lagrange polynomial as

$$\frac{dN_j(r)}{dr} = \sum_{n=1}^{M_P} \frac{dN_j(r)}{dr}\Big|_{r_n} N_n(r) \tag{3.51}$$

Now, consider an element of the stiffness matrix (note this matrix is different from the stiffness matrix $\mathcal{K}$ in (3.45))

$$\begin{aligned}
\mathcal{S}_{i,j} &= \int_{-1}^{1} N_i(r)\frac{dN_j(r)}{dr}dr \\
&= \int_{-1}^{1} N_i(r)\sum_{n=1}^{M_P} \frac{dN_j(r)}{dr}\Big|_{r_n} N_n(r)dr \\
&= \sum_{n=1}^{M_P} \frac{dN_j(r)}{dr}\Big|_{r_n} \int_{-1}^{1} N_i(r)N_n(r)dr
\end{aligned} \tag{3.52}$$

Introduce the differentiation matrix

$$\mathcal{D}_{r,(i,j)} = \frac{dN_j}{dr}\Big|_{r_i} \tag{3.53}$$

and we find that

$$\mathcal{S} = \mathcal{M}\mathcal{D}_r \tag{3.54}$$

The entries of the differentiation matrix is found directly from

$$\mathcal{V}^T\mathcal{D}_r^T = (\mathcal{V}_r)^T, \quad \mathcal{V}_{r,(i,j)} = \frac{d\psi_j}{dr}\Big|_{r_i} \tag{3.55}$$

Thus, we can compute the differentiation matrix from

$$\mathcal{D}_r = \mathcal{V}_r\mathcal{V}^{-1} \tag{3.56}$$

To determine the element stiffness (diffusion) matrix

$$\mathcal{K}^{(n)} = \int_{e_n} \frac{dN_i(r)}{dr}\frac{dN_j(r)}{dr}dr \tag{3.57}$$

we choose to express the derivative of the polynomials in the Lagrange basis as

$$\frac{d}{dr}N_j(r) = \sum_{n=1}^{M_P} \frac{dN_j(r)}{dr}\Big|_{r_n} N_n(r) \tag{3.58}$$

and use this to determine the element diffusion matrix in the compact form

$$\mathcal{K}^{(n)} = (\mathcal{D}_r^{(n)})^T\mathcal{M}^{(n)}\mathcal{D}_r^{(n)} = \frac{2}{h_n}\mathcal{D}_r^T\mathcal{M}\mathcal{D}_r \tag{3.59}$$

This expression can be used for computing the element diffusion matrix exactly using Vandermonde-type matrices and only requires that we can represent the local solution using a modal basis which is orthogonal.

## 3.6 The computation

To do a global assembly of the linear system based on a Spectral/$hp$-FEM discretization we need to make a connection between the local element nodes and the set of unique global nodes. This can be achieved by introducing a connectivity matrix defined such that

$c_{n,1}$ = `gidx` is the global number of the first node of the $n$'th element
$c_{n,2}$ = `gidx`+1 is the global number of the second node of the $n$'th element
$\vdots$
$c_{n,P+1}$ = `gidx`+$P$ is the global number of the $P+1$ node of the $n$'th element

If there are only two nodes per element, i.e. the nodes correspond to the vertex nodes of the elements, then the connectivity matrix is equivalent to the Element-To-Vertex connectivity table `EToV`. For higher order discretizations where the local polynomial order can be arbitrarily high, the number of columns is larger than two, since then the connectivity matrix **c** tracks also the local nodes of each element by their global node numbers. The connectivity matrix **c** relating the index

| C | | | | | |
|---|---|---|---|---|---|
| $n$ | 1 | 2 | 3 | $\cdots$ | $N$ |
| 1 | 1 | 2 | 3 | $\cdots$ | $P$+1 |
| 2 | $P$+1 | $P$+2 | $P$+3 | $\cdots$ | 2$P$+1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| N | $(N-1)P$+1 | $(N-1)P$+2 | $(N-1)P$+3 | $\cdots$ | $NP$+1 |

Table 3.1: Element table for the mesh in Figure 3.1.

of the local nodes to each element to the unique global nodes can be generated using Algorithm 10 for one spatial dimension and is illustrated in Table 3.1 for the mesh in Figure 3.1 where all nodes are ordered and numbered from left to right.

**Algorithm 10:** Determine connectivity table for local-to-global mappings (1D).

```
Allocate storage for C
gidx = 2
for n := 1 to N
    gidx = gidx - 1
    for i := 1 to M_p
        C[n, i] := gidx
        gidx := gidx + 1
```

This resulting connectivity table **C** can then be used in the global assembly process and Algorithm 11 can then replace Algorithm 1.

**Algorithm 11:** Global assembly of coefficient matrix **A** (1D, high-order).

Allocate storage for **A** and **b**
Compute standard operators, cf. (3.49) and (3.56)
**for** $n := 1$ **to** $N$
    Compute $k_{i,j}^{(n)}$ from (3.46).
    **for** $j := 1$ **to** $M_P$
        **for** $i := 1$ **to** $j$
            $a[C[n,i], C[n,j]] := a[C[n,i], C[n,j]] + k_{i,j}^{(n)}$

Similar to Algorithm 1, the new Algorithm 11 also take into account symmetry of the coefficient matrix **A**. Again, corrections to the right hand side vector needs to be done to take into account the boundary conditions. This can be achieved with Algorithm 12 after the the global assembly of the coefficient matrix and right hand side vector of the linear system has been built.

For the modification of the coefficient matrix **A** using Algorithm 12 it is assumed that the matrix is symmetric and that the connectivity table holds information about the first and last element in, respectively, the first and last row of the array. Note how the first element of the right hand side vector is updated twice in the algorithm to ensure that this element is not left overwritten after the modification procedure in the case where there is only one element.

**Algorithm 12:** Imposing boundary conditions by modification of system (1D, high-order).

$b[1] := c$
$a[1,1] := 1$
**for** $i := 2$ **to** $M_p$
    $b[i] := b[i] - a[1,i] * c$
    $a[1,i] := 0$
$b[C[N, M_P]] := d$
**for** $i := 1$ **to** $M_P - 1$
    $b[C[N,i]] := b[C[N,i]] - a[C[N,i], C[N, M_P]] * d$
    $a[C[N,i], C[N, M_P]] := 0$
$a[C[N, M_P], C[N, M_P]] := 1$
$b[1] := c$

## 3.7 The Neumann boundary conditions

In Section 2.9 it was described how to impose Neumann boundary conditions in the classical FEM discretization. To extend this to the Spectral/*hp*-FEM implementation, we reconsider the procedure for solving the differential equation (1.14) when the boundary condition on a part of the boundary is of Neumann type.

As stated in Section 2.5, the boundary conditions of Neumann type cannot be applied to all of the boundaries, since then the solution of the problem is not unique. Therefore to illustrate the changes in the derivation procedure, we make the assumption that on the boundaries of the interval $0 \leq x \leq L$ in one space dimension, we require $u(x)$ to satisfy the Neumann boundary condition

$$u'(x) = q, \quad x = 0 \tag{3.60}$$

and the Dirichlet boundary condition

$$u = d, \quad x = L \tag{3.61}$$

To derive the weak formulation of this problem, we return to (1.18) and require now that $v$ be zero only on $x = L$ where the Dirichlet boundary conditions needs to be fulfilled. Using (3.60), we can rewrite (1.18) as

$$\int_0^L (u'v' + uv)dx = -u'(0)v(0) \tag{3.62}$$

The problem of finding a function $u(x)$ that satisfies (3.62) in $0 \leq x \leq L$ and boundary conditions (3.60), (3.61) is equivalent to finding a function $u(x)$ that

1. satisfies (3.62) for all sufficiently smooth $v(x)$ such that $v = 0$ on $x = L$,
2. satisfies (3.61).

Let there be given a mesh with $M$ nodes. To solve the problem using a Spectral/$hp$-FEM, we approximate $u(x)$ by a function $\hat{u(x)}$ of type (3.1). Because of the Dirichlet boundary conditions on $x = L$, we put

$$\hat{u}_M = d, \quad x_M = L \tag{3.63}$$

To derive the remaining $(M - 1)$ equations, we substitute (3.1) for $u(x)$ in (3.62) and let $v$ be, successively, all of the functions $N_i(x)$, $i = 1, 2, ..., M - 1$. Note that all of these functions vanish on the boundary $x = L$ as required. From these steps we find

$$\sum_{i=1}^{M} \hat{u}_i \int_0^L \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx = -u'(0)v(0)\delta_{1,j}, \quad j = 1, ..., M - 1 \tag{3.64}$$

Thus, these equations can be collected to produce a linear system of the form

$$\mathbf{A}\hat{\mathbf{u}} = \mathbf{b} \tag{3.65}$$

where the elements of the coefficient matrix $\mathbf{A}$ is defined from

$$a_{i,j} = \int_0^L \left( \frac{dN_i}{dx}\frac{dN_j}{dx} + N_i N_j \right) dx, \quad i = 1, ..., M - 1 \tag{3.66}$$

for the interior coefficients and the first coefficient. Due to the boundary condition (3.61) the remaining coefficients are

$$a_{M,j} = \delta_{M,j} \tag{3.67}$$

The corresponding right hand side vector $\mathbf{b}$ for the linear system is defined as

$$\begin{aligned} b_1 &= -q \\ b_i &= 0, \quad i = 2, ..., M \end{aligned} \tag{3.68}$$

Thus, we have defined a system of $M$ linear equations in $M - 1$ unknowns.

Note, how the only changes to the procedure described in Section 3.4 where two Dirichlet boundary conditions where applied at the ends of the domain, involves modifying the first row of the system matrix $\mathbf{A}$ together with updating the first element of $\mathbf{b}$.

To do a global assembly of the linear system we can use Algorithm 11 which sets up the system without any boundary conditions imposed. To do the necessary modifications to take into account the boundary conditions (a Neumann condition at the left boundary node and a Dirichlet condition at the right boundary node), we need to apply Algorithm 13.

**Algorithm 13:** Imposing Neumann boundary conditions by modification of system (1D, high-order).

$b[C[N, M_P]] := d$
**for** $i := 1$ **to** $M_P - 1$
    $b[C[N, i]] := b[C[N, i]] - a[C[N, i], C[N, M_P]] * d$
    $a[C[N, i], C[N, M_P]] := 0$
$a[C[N, M_P], C[N, M_P]] := 1$
$b[1] := b[1] - q$

## 3.8 Error bounds

It is difficult to prove general error bounds for finite element solutions of differential equations (e.g. see review [6]) and the discussion here is far from exhaustive. However, we present a few results that can indicate the properties of the Spectral/*hp*-FEM when used for the approximation of continuous solutions. Fortunately, for practical purposes it is straightforward to do experimental tests numerically to produce estimates for a particular problem solved approximately by the use of Spectral/*hp*-FEM. If estimates are available it is useful to compare the numerical results with the analytical and expected results for validation purposes.

An error bound for the approximation error of the *p*-version of the FEM for the Helmholtz model problem

$$\begin{aligned} u''(x) - u(x) &= f(x), \quad x \in \Omega, \\ u(x) &= 0, \quad x \in \Gamma(\Omega) \end{aligned} \tag{3.69}$$

with $f(x)$ a square-integrable function on the domain $\Omega$, is expressed in the energy norm [8] as

$$||u(x) - \hat{u}(x)||_E \leq C_1 P^{-(k-1)} ||u||_{k,\Omega} \tag{3.70}$$

where $C_1$ is independent of the solution $u(x)$ and polynomial order $P$. The results have been expressed in terms of a Sobolov norm defined as

$$||u||^2_{k,\Omega} = \sum_{|\alpha|=0}^{k} ||D^\alpha u||^2_{0,\Omega}, \quad D^\alpha = \frac{d^\alpha}{dx^\alpha}, \quad ||u||^2_{0,\Omega} = \int_\Omega u^2 dx \tag{3.71}$$

which can be interpreted as a simple measure of the magnitude and smoothness of a function. The appearance of the Sobolov norm of $u$ in the estimate (3.70) indicates that the magnitude of the errors is dependent on the exact solution to the problem. For the *h*-version of FEM the following estimate is found

$$||u(x) - \hat{u}(x)||_E \leq C_2 h^\mu ||u||_{k,\Omega} \tag{3.72}$$

where $\mu = \min(k-1, P)$ and $h$ is a measure of the largest element size on a quasi-uniform mesh. The magnitude of $\mu$ determines the rate of convergence when *h*-refinement is employed with polynomial order fixed. Combining the two estimates (3.70) and (3.72) we find

$$||u(x) - \hat{u}(x)||_E \leq C_3 h^\mu P^{-(k-1)} ||u||_{k,\Omega} \tag{3.73}$$

where $C_3$ is dependent on $k$. Thus, for smooth solutions we can in generally expect order $P$ convergence under *h*-refinement. Furthermore, from the upper bounds we recognize the there are two paths to convergence as expected, respectively *h*- and *p*-type convergence. Furthermore, for the *h*-version the convergence rate is limited by the order of the polynomials used for the approximation on the elements. For the *p*-version the rate of convergence will be exponential for smooth solutions because then for all $k$ the Sobolev norm $||u||_{k,\Omega}$ is finite.

The results for various $h$-convergence tests for different polynomial orders for problem (1.14) and (1.15) is shown in Figure 3.4. Clearly, the magnitude of the errors can be reduced much faster by using the higher order formulations (until the level where roundoff-errors in the computations becomes significant).

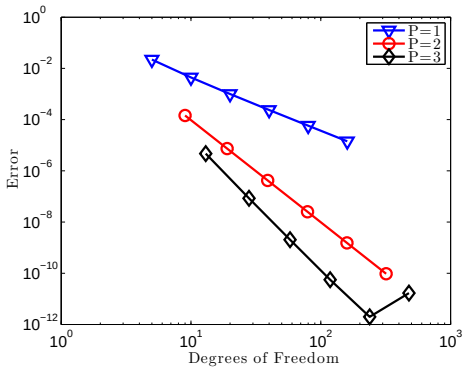



Figure 3.4: $h$-convergence tests for various polynomial orders for problem (1.14) and (1.15).

# Exercises

## Exercise 3.1

a) Algorithm 10 can only be used in cases where both the mesh nodes and elements are both ordered and connected from left to right. Write a Matlab routine that can transform an arbitrary element connectivity table and corresponding set of element nodes to have the node ordering and element numbering assumed in Algorithm 10. (HINT: consult Appendix D).

**Head:**
```
function [x,EToV] = reordermesh1D(x,EToV)
```

**Test Case:**
```
x = [1.0 2/3 0.0 1/3]; EToV = [3 4; 4 2; 2 1];
```

b) Write a Matlab routine for Algorithm 10.

**Head:**
```
function [C] = ConstructConnectivity1D(EToV,P)
```

**Test Case:**
```
EToV = [3 4; 4 2; 2 1];
```

c) From the modified connectivity table `EToV` and coordinate table `x` create a mesh coordinate array `x1d(1:M)` consisting of coordinate positions for a mesh with uniform node distributions on the elements and a total of `M` nodes. The uniform distribution is specified in the vector `r(1:P+1)` which should contain $P+1$ nodes on the reference interval $r \in [-1,1]$ including the endpoint as specified in the argument list.

d) Repeat c) but with Legendre-Gauss-Lobatto node distributions on the elements.

**Head:**
```
function [x1d] = mesh1D(x,EToV,r)
```

**Test Case:**
```
r = linspace(-1,1,P+1);
```

## Exercise 3.2

a) Determine an interpolating function $u_I(x)$ to $u(x) = \exp(\cos(\pi x))$ on the interval $x \in [-1,1]$ using 4 elements with 5 local nodes uniformly distributed on each element. Then evaluate the interpolating function $u_I(x)$ using the relationship (3.18) on the Legendre-Gauss-Lobatto node distribution on the local elements. (HINT: map global nodes to local nodes. Then employ a Vandermonde-type matrix for the determining the coefficients of the expansion followed by evaluation to the new set of points using a second Vandermonde-type matrix. Map the transformed solution and new set of nodes back to a global solution vector and corresponding nodes).

b) Consider the interpolation of the function $u(x) = \sin(\pi x)$ on the interval $x \in [-1, 1]$. Carry out a numerical $h$-convergence test where the convergence estimate given in (3.23) is confirmed in the asymptotic limit where $h \to 0$ for different orders of the local polynomial expansions, e.g. $P = 1, 2, 3, 4$.

## Exercise 3.3

a) Show that the spectral element stiffness (diffusion) matrix $\mathcal{K}^{(n)}$ can be computed by matrix-matrix products of the form given in (3.59).

b) Write a Matlab script to compute the spectral element matrices involved in computing the spectral element stiffness (diffusion) matrix $\mathcal{K}^{(n)}$ in a). The supplied auxiliary routines can be used (see Appendix F).

**Head:**
```
function [Kn] = stiffnessmatrix1D(hn,P,r)
```

**Test Case:**
```
P = 5, [r] = JacobiGL(0,0,P), hn = 1.0.
```

## Exercise 3.4

a) Repeat Exercise 1.2 b) and write a Spectral/*hp*-FEM Matlab code which creates all local elemental matrices which in any case are computed for the chosen parameters. $P$ is the order of the local polynomial basis functions to be used. `M` is the total number of vertex nodes in the mesh and for $P > 1$ every local element will contain interior nodes.

b) Do a convergence study where $h$- and $p-$type convergence are compared. What is the rate of convergence in each case?

**Head:**
```
function [u] = BVP1Dhp(L,c,d,M,P)
```

## Exercise 3.5

a) Derive a weak formulation for the one-dimensional Poisson equation

$$u''(x) = f(x)$$

b) Write a Spectral/*hp*-FEM Matlab code which determines an approximate solution to the problem given in a) for $x \in [-1, 1]$ with right hand side function and boundary conditions

$$f(x) = \exp(4x), \quad u(\pm 1) = 0.$$

c) To validate your code carry out both $h$- and $p$-convergence tests and compare with the exact solution $u(x) = (\exp(4x) - x\sinh(4) - \cosh(4))/16$.

## Exercise 3.6

a) Derive a weak formulation for the one-dimensional Poisson equation

$$u''(x) = f(x)$$

b) Write a Spectral/*hp*-FEM Matlab code which determines an approximate solution to the problem given in a) for $x \in [0, L]$ with right hand side function and boundary conditions

$$f(x) = -\pi^2 \cos(\pi x), \quad u(0) = 1, \quad u'(L) = -\pi \sin(\pi L).$$

c) To validate your code carry out both *h*- and *p*-convergence tests and compare with the exact solution $u(x) = \cos(\pi x)$ for the case where $L = 1.6$.

# Chapter 4

# Spectral/$hp$-FEM in two space dimensions

The extension of the classical ideas of FEM into generalised Spectral/$hp$-FEM for solving differential equations in two space dimensions is conceptually trivial, but introduces new complexity in terms of bookkeeping. Thus, we will need to revisit the same steps as we have done for the classical FEM in Chapter 2 and the high-order extensions will be similar to those introduced for the Spectral/$hp$-FEM in one space dimension in Chapter 3.

## 4.1 The Spectral/$hp$-Finite Element Mesh

To consider problems in two space dimensions, the two-dimensional domain $\bar{\Omega}$ with interior $\Omega$ and boundary $\Gamma$ needs to be represented in the form of a finite element mesh consisting of triangular elements defined by a set of vertex nodes. We restrict ourselves to only consider straight-sided triangles since the purpose of these lecture notes is to guide the reader through the basic steps of implementing a high-order spectral element method for use with unstructured triangular meshes.

The goal is to be able to handle general meshes. Thus, for a generic implementation we are faced with the following issues

- How to generate the mesh?
- How to automatically compute required mesh information such as transformation weights and geometric information?
- How to semi-automatically handle different types of boundary conditions?

It is possible to use any suitable mesh generator (cf. Appendix D) to generate the required mesh data tables, respectively a coordinate table `p=[VX(:) VY(:)]` and a connectivity table `EToV`. These mesh data tables holds information about the vertex nodes and uniquely defines the straight-sided elements. For reference in the following, we again make use of the mesh for the rectangular domain illustrated in Figure 2.5 with connections defined in Table 2.1 in the form of an Element-To-Vertex table `EToV`.

Each of the elements of the mesh needs to be populated with a set of nodes. The local set of nodes determines what we call the local degrees of freedom and is measured as the number of local nodes $M_P$. The positions of the local nodes can be shown to determine the numerical stability and interpolation accuracy of the implemented method. When the set of local nodes includes more nodes than the three vertex nodes defining each triangle, we need to be able to uniquely

| EToE | | | |
|---|---|---|---|
| *n* | *1* | *2* | *3* |
| 1 | 1 | 2 | 8 |
| 2 | 3 | 1 | 2 |
| 3 | 2 | 4 | 10 |
| 4 | 5 | 3 | 4 |
| . | . | . | . |
| 24 | 24 | 23 | 17 |

(a)

| EToF | | | |
|---|---|---|---|
| *n* | *1* | *2* | *3* |
| 1 | 1 | 2 | 3 |
| 2 | 1 | 2 | 3 |
| 3 | 1 | 2 | 3 |
| 4 | 1 | 2 | 3 |
| . | . | . | . |
| 24 | 1 | 2 | 3 |

(b)

Table 4.1: a) `EToE` and b) `EToF` tables for the mesh in Figure 2.5.

identify those nodes and keep track of them in the Spectral/*hp*-FEM setup. Furthermore, for the Spectral/*hp*-FEM implementation we need automatic procedures which are accurate and robust for computing geometric information related to the mesh setup, local element matrices for the Spectral/*hp*FEM discretization and a convenient way to handle the book-keeping for relating the local and the global information. In the following, we employ procedures and scripts described in [34]. The necessary auxiliary scripts are contained in Appendix F.

First, create two more mesh data tables storing the Element-To-Element connectivity `EToE` and the Element-To-Face connectivity `EToF` using

```
>> [EToE,EToF]= tiConnect2D(EToV,Nfaces);
```

where the number of elements is specified as $N_{faces} = 3$ for triangles. The two additional mesh data tables are useful to have at hand for the global assembly procedure described in Section 4.5. The `EToE` table describes for each face of an element which element is the neighboring element to the face. The `EToF` table describes for each face which local face number of the neighboring element a given element connects to. For the reference mesh shown in Figure 2.5 the two additional mesh data tables are illustrated in Table 4.1. Remark that for boundary edges and elements where there are no connections self-referencing have been employed. For example, the first element ($n = 1$) in the `EToE` table connects to it self, because the first face is a boundary edge.

The local coordinates can be determined on the reference triangle space $(r, s)$ using the commands

```
>> [x,y] = Nodes2D(P); % reference element nodes
>> [r,s] = xytors(x,y);
```

where `Nodes2D.m` and `xytors.m` are routines that explicitly computes a near-optimal set of interpolation nodes on the triangle using a Warp & Blend procedure described in [34, 59]. Another near-optimal set of nodes are the Fekete node distribution which can be computed, e.g. using the algorithm presented in [55]. The Fekete points are defined as the set of points which maximize the determinant of the Vandermonde matrix. The determination of the Fekete nodes is complicated due to the lack of a closed form solution and therefore has to be computed through an iterative process for a given polynomial order. Solutions to the Fekete problem of determining unknown nodes for different polynomial orders of the local basis can be tabulated for reuse. However, fast and explicit computations circumvent the need for tabulated data and is preferred in this context.

To build the coordinates of all nodes we use a Gordon-Hall transfinite interpolation procedure [28] to map the local coordinates to the physical coordinates $(x, y)$ using the transformation

$$\mathbf{x} = \frac{-(\mathbf{r}+\mathbf{s})}{\mathbf{2}}\mathbf{v_1} + \frac{(\mathbf{1}+\mathbf{r})}{\mathbf{2}}\mathbf{v_2} + \frac{(\mathbf{1}+\mathbf{s})}{\mathbf{2}}\mathbf{v_3} \tag{4.1}$$

where the coordinate pairs $v_i = (x_i, y_i)^T$ for the vertex nodes define the domains for the elements in physical space $(x, y)$. In Matlab the coordinates of all nodes can be build with the following set of commands

```
>> v1 = EToV(:,1)'; v2 = EToV(:,2)'; v3 = EToV(:,3)';
>> VX = p(:,1)'; VY = p(:,2)';
>> x = 0.5*(-(r+s)*VX(v1)+(1+r)*VX(v2)+(1+s)*VX(v3));
>> y = 0.5*(-(r+s)*VY(v1)+(1+r)*VY(v2)+(1+s)*VY(v3));
```

Having determined all nodes in two data arrays `x` and `y` (size $M_P \times N$) we are faced with the task of relating the local to the global nodes and vice versa. This is a bookkeeping problem which is fundamental for the efficient implementation of Spectral/*hp*-FEM. With the shown computations we have guaranteed that all elements have the same ordering of the nodes. However, it is necessary to decide on the ordering to make the implementations straightforward. Here we make the choice that the nodes should be ordered as vertex nodes first, then edge nodes and last all interior nodes. Ensuring a proper order of nodes is needed for the local-to-global mappings that are needed when the global contributions needs to be determined, e.g. in the global assembly procedure.

```
% Compute index maps for node positions
>> tol = 0.2/P^2;
>> Mp = length(s);
% Faces
>> fid1 = find( abs(s+1) < tol)';
>> fid2 = find( abs(r+s) < tol)';
>> fid3 = find( abs(r+1) < tol)';
>> Fids = [fid1;fid2;fid3]';
% Interior
>> fint = setdiff(1:Mp,[fid1 fid2 fid3]);

% Reorder local nodes to match the chosen ordering in terms of vertex nodes,
% edge nodes and interior nodes.
>> Mpf = P+1;
>> LocalReorder = [1 Mpf Mp fid1(2:Mpf-1) fid2(2:Mpf-1) fid3(Mpf-1:-1:2) fint];
>> x = x(LocalReorder,:);
>> y = y(LocalReorder,:);
>> r = r(LocalReorder);
>> s = s(LocalReorder);
```

From the mesh data tables, we need to be able to compute the Jacobians of the transformation accurately and consistently together with the geometric information about the outer normal vector components. This can be computed using simple local finite element operations as follows

```
% reference element matrices
>> [V2D] = Vandermonde2D(P, r, s);
>> [Dr,Ds] = Dmatrices2D(P, r, s, V2D);
% calculate geometric factors
>> [rx,sx,ry,sy,J] = GeometricFactors2D(x,y,Dr,Ds);
% calculate geometric normals and edge lengths
>> [nx, ny, sJ] = Normals2DFEM(x,y,Dr,Ds,Fids,Npf,N);
>> Fscale = sJ./(J(Fids,:));
```

## 4.2 Spectral/*hp*-Finite Element Basis Functions

A function $u(x)$ is represented by piece-wise polynomial functions of the form

$$\hat{u}(x,y) = \sum_{i=1}^{M} \hat{u}_i N_i(x,y) \tag{4.2}$$

where $N_i(x,y)$ is a global finite element basis function defined such that it possess the Cardinal property for the $i$'th node and vanishes for all other nodes

$$N_i(x_i,y_i) = \delta_{i,j} \tag{4.3}$$

Each of these global basis functions are defined on the entire domain of interest, but have local support only over at most a few neighboring elements.

Similar to the one space dimension formulation, it is possible to represent each of the global basis functions $N_i(x,y)$ in terms of local basis functions defined on the elements only in the compact form

$$N_i(x,y) = \bigoplus_{n=1}^{N} \sum_{j=1}^{M_P} N_j^{(n)}(x_i,y_i) N_j^{(n)}(x,y) \tag{4.4}$$

where the local basis function $N_j^{(n)}(x,y)$ is a multivariate Lagrange polynomial belonging to the $n$'th element and defined from the set of nodes that have been associated with the element.

Thus, over the element we have a local representation of an arbitrary function of the form

$$\hat{u}(x,y) = \sum_{j=1}^{M_P} \hat{u}_j N_j^{(n)}(x_i,y_i) N_j^{(n)}(x,y) \tag{4.5}$$

However, as discussed in Section 3.2 it is possible to make a choice in representation, namely, whether the local representation should be modal or nodal. The choice reflects the choice of basis used for representation, since the function in question should be uniquely defined. Thus, the local solution can be represented in the general form

$$u(r,s) = \sum_{i=1}^{M_P} \hat{u}_i \psi(r,s) \tag{4.6}$$

where $\psi(r,s)$ are the basis functions and $\hat{u}_n$ are the coefficients or weights of the influence of the basis functions. The possibility to represent the local solution in both modal and nodal forms can similar to one space dimension also be exploited in two space dimensions for the computation of the local elemental matrices needed for the global assembly. This can be exploited in any number of dimensions.

### Local basis in 2D

For 2D discretizations we require that our local basis functions span a 2D polynomial space of order $P$ (cf. [30, 36, 45]), i.e.

$$\mathcal{P}^P = \text{SPAN}\{x^\alpha y^\beta\}, \quad \alpha,\beta \geq 0, \alpha+\beta \leq P \tag{4.7}$$

A complete polynomial space of order $P$ will then have dimension $\frac{1}{2}(P+1)(P+2)$. A basis can therefore be ordered as illustrated in Table 4.2. It is noted, that the necessary polynomial orders of the basis polynomials derived from mixed products can be deduced with help from Pascal's triangle of order $P$, which expresses the coefficients of the terms in the binomial expansion $(x+y)^m$, where

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | 1 | | | |
| | | x | | y | | |
| | $x^2$ | | xy | | $y^2$ | |
| $x^3$ | | $x^2y$ | | $xy^2$ | | $y^3$ |
| ... | ... | | ... | | ... | ... |

Table 4.2: Determining a complete polynomial basis in 2D of order $P$ using Pascal's triangle.

$m \geq 0$ is an integer. By selecting a complete polynomial space for a given order $P$ the variation of a function $f(\xi, \eta)$ along any straight coordinate line will match the polynomial order of the complete expansion basis and thereby exhibit the property known as *geometric isotropy*. However, it is also possible to choose either an incomplete (e.g. see discussion in [37]) or an extended basis. Any choice of polynomial approximation space will impact the accuracy of the approximations and may affect the aliasing properties of the scheme, e.g. quadrilaterals vs. triangles.

## Basis functions for 2D triangulations

The unstructured orthogonal basis functions proposed by Proriol [46] and later Dubiner [19] in two horizontal dimensions has been adopted throughout the work. This set of basis functions provides the basis for spectral accuracy. The triangular expansion is defined on the reference triangle

$$\mathcal{I} = \{(r, s) | -1 \leq r, s; r + s \leq 0\}. \tag{4.8}$$

By introducing the so-called *collapsed coordinate system* as described in [36], the reference triangle is uniquely mapped to a reference quadrilateral by the following mapping

$$a = 2\frac{1+r}{1-s} - 1, \quad b = s, \tag{4.9}$$

such that the reference triangle can be defined in terms of the collapsed coordinates on a reference quadrilateral as

$$\mathcal{I} = \{(a, b) | -1 \leq a, b \leq 1\}, \tag{4.10}$$

which has independent limits, and thus is suitable for using one-dimensional basis functions to construct a multi-dimensional basis on the triangle (similar to a tensorial basis on a structured grid, e.g. the Legendre-Legendre tensor basis in $L^2$).

In the collapsed coordinates the *orthogonal principal basis functions* are defined as

$$\phi_{pq}(r, s) = \tilde{\psi}_p^a(a)\tilde{\psi}_q^b(b), \tag{4.11}$$

using the one-dimensional basis functions

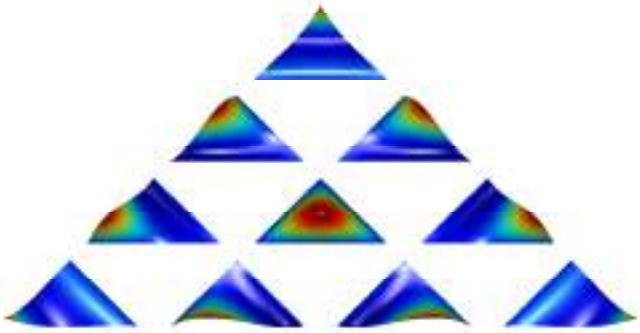
$$\tilde{\psi}_p^a(r) = P_p^{0,0}(r), \quad \tilde{\psi}_p^b(s) = \left(\frac{1-s}{2}\right)^p P_q^{2p+1,0}(s), \tag{4.12}$$

where $P_p^{\alpha,\beta}(z)$ is the $p$th order Jacobi polynomial, which belongs to the family of orthogonal polynomial solutions to the singular Sturm-Liouville problem. The resulting modal and nodal shape functions are depicted in Figure 4.1.

Using the auxiliary routines included in Appendix F we can evaluate the Proriol basis on the triangle using the commands

```
>> [a,b] = rstoab(r,s);
>> [Psi] = Simplex2DP(a,b,i,j);
```

(a)



(b)

Figure 4.1: Shape of the chosen basis functions on the standard triangle; a) modal and equivalent b) nodal polynomial basis functions.

where the reference coordinates have first been mapped to the collapsed coordinates $(a, b)$. The spatial derivatives of the triangular basis functions can be determined using

```
>> [dPsidr, dPsids] = GradSimplex2DP(a,b,i,j);
```

The generalized Vandermonde matrix for a $P$'th order complete basis can be computed with the command

```
>> [V] = Vandermonde2D(P, r, s);
```

and the Vandemonde matrices holding the gradients of the basis functions as

```
>> [Vr,Vs] = GradVandermonde2D(P,r,s);
```

## 4.3 Spectral/*hp*-Finite Element Method

Following conceptually the same procedure as described in Section 2.6, we show how to apply the Spectral/*hp*-FEM to problem (2.11) taking into account the Dirichlet boundary condition (2.14).

The usual starting point is the weak formulation associated with this problem.

Let $v(x,y)$ be an arbitrary smooth function defined on $\bar{\Omega}$ with the property that

$$v(x,y) = 0, \quad (x,y) \in \Gamma \tag{4.13}$$

Multiplying both sides of (2.11) by $v$ and integrating over $\bar{\Omega}$ we obtain

$$\int\int_{\Omega} \left[(\lambda_1 u_x)_x + (\lambda_2 u_y)_y\right] v dxdy = -\int\int_{\Omega} \tilde{q} v dxdy \tag{4.14}$$

Integration by parts with respect to $x$ and $y$ leads to the following fundamental identities

$$\int\int_{\Omega} (\lambda_1 u_x)_x v dxdy = \int_{\Gamma} \lambda_1 u_x v n_x ds - \int\int_{\Omega} \lambda_1 u_x v_x dxdy$$
$$\int\int_{\Omega} (\lambda_2 u_y)_y v dxdy = \int_{\Gamma} \lambda_2 u_y v n_y ds - \int\int_{\Omega} \lambda_2 u_y v_y dxdy$$

where the outward point normal vector to the domain boundary is defined as $\mathbf{n} = (n_x, n_y)^T$. By summation of these identities we can form

$$\int_{\Gamma} \left[\lambda_1 u_x v n_x + \lambda_2 u_y v n_y\right] ds - \int\int_{\Omega} \left[\lambda_1 u_x v_x + \lambda_2 u_y v_y\right] dxdy = -\int\int_{\Omega} \tilde{q} dxdy \tag{4.15}$$

The first integral vanishes due to (4.13) and this equation reduces to

$$\int\int_{\Omega} \left[\lambda_1 u_x v_x + \lambda_2 u_y v_y\right] dxdy = \int\int_{\Omega} v\tilde{q} dxdy \tag{4.16}$$

This is the weak formulation of the problem.

We then seek an approximation $\hat{u}(x,y)$ to $u(x,y)$ of the form (4.2).

## 4.4 Element Matrices

We are concerned with how to compute the local contributions on an element-to-element basis for the discretization of Spectral/*hp*-FEM formulated in terms of a weak formulation. For example, we could be concerned with the computation of integrals of the form

$$\int\int_{\Omega} \frac{\partial}{\partial x} u(x,y) v(x,y) dxdy \tag{4.17}$$

where $u(x,y)$ is an exact solution and $v(x,y)$ is a test function. The exact solution will be approximated by a FEM function of the form

$$\hat{u}(x,y) = \sum_{j=1}^{M} \hat{u}_j N_j(x,y) \tag{4.18}$$

where $N_j(x,y)$ is the $j$'th global basis function defined over the domain $\Omega$.

Now, in the Galerkin formulation of a FEM problem, we replace the exact solution with an approximate solution and assume that the test functions is taken as $v(x,y) = N_i(x,y)$ for all $i = 1, ..., M$, and we can then write the initial integral term as

$$\int\int_{\Omega} \sum_{j=1}^{M} \hat{u}_j \frac{\partial}{\partial x} N_j(x,y) N_i(x,y) dxdy \tag{4.19}$$

which can be rewritten into

$$\sum_{j=1}^{M} \hat{u}_j \int\int_{\Omega} \frac{\partial}{\partial x} N_j(x,y) N_i(x,y) dxdy \tag{4.20}$$

The integral contributions over the global domain in this expression can be expressed in terms of the local finite element contributions using that

$$\int\int_{\Omega} \frac{\partial}{\partial x} N_j(x,y) N_i(x,y) dxdy = \sum_{n=1}^{N} \int\int_{e_n} \frac{\partial}{\partial x} N_j(x,y) N_i(x,y) dxdy \tag{4.21}$$

Thus, our concern will then be to determine how we can compute the local integrals of the form

$$\int\int_{e_n} \frac{\partial}{\partial x} N_j(x,y) N_i(x,y) dxdy \tag{4.22}$$

for an arbitrary element $e_n$, $n = 1, 2, ..., M$.

On each element, we can represent the local solution using local polynomials of the form

$$\hat{u}^{(n)}(x,y) = \sum_{j=1}^{M} \hat{u}_j^{(n)} N_j^{(n)}(x,y) \tag{4.23}$$

Thus, on the element, we can express the global basis functions in terms of the local basis functions and thus the last integral can be written as

$$\int\int_{e_n} \frac{\partial}{\partial x} N_j^{(n)}(x,y) N_i^{(n)}(x,y) dxdy \tag{4.24}$$

Thus, by taking $i, j = 1, ..., M$ we find all possible combinations of the local basis functions appearing in the integrand and this defines an element matrix $\mathcal{K}^{(n)}$ with elements defined as

$$\mathcal{K}_{i,j}^{(n)} \equiv \int\int_{e_n} \frac{\partial}{\partial x} N_j^{(n)}(x,y) N_i^{(n)}(x,y) dxdy \tag{4.25}$$

So, how can we compute such elements? It turns out to be convenient to make use of the reference domain $\mathcal{I}$ we introduced for a standard triangle in the $(r,s)$-coordinate system (computational domain). Thus, we map the element domain to the reference domain using a Gordon-Hall transfinite interpolation procedure. By the chain rule, and the use of this mapping we can transform the local integral as

$$\begin{aligned} &\int\int_{e_n} \frac{\partial}{\partial x} N_j^{(n)}(x,y) N_i^{(n)}(x,y) dxdy = \\ &\mathcal{J}^{(n)} \int\int_{I} \left( r_x \frac{\partial}{\partial r} N_j(r,s) + s_x \frac{\partial}{\partial s} N_j(r,s) \right) N_i(r,s) drds \end{aligned} \tag{4.26}$$

where $s_x$ and $s_y$ are geometric weights and $\mathcal{J}^{(n)}$ is the Jacobian of the transformation between the two domains. It is seen that, such local integrals involves partial contributions of the form

$$(\mathcal{S}_r)_{i,j} \equiv \int\int_{I} \frac{\partial}{\partial r} N_j(r,s) N_i(r,s) drds \tag{4.27}$$

$$(\mathcal{S}_s)_{i,j} \equiv \int\int_{I} \frac{\partial}{\partial s} N_j(r,s) N_i(r,s) drds \tag{4.28}$$

The computation of these exploits that we can represent the local nodal polynomials equivalently as local modal polynomials picked from an orthonormal family. One such choice is the Proriol basis on the triangle we introduced earlier.

To determine the integral contributions in $(\mathcal{S}_r)_{i,j}$ we make the convenient choice of expressing the derivative of the $j$'th nodal (Lagrange) polynomial as

$$\frac{\partial}{\partial r}N_j = \sum_{m=1}^{M} \frac{\partial}{\partial r}N_j(r,s)\Big|_{(r_m,s_m)} N_m(r,s) \tag{4.29}$$

We insert this expression and find

$$(\mathcal{S}_r)_{i,j} \equiv \int\int_I \left( \sum_{m=1}^{M} \frac{\partial}{\partial r}N_j(r,s)\Big|_{(r_m,s_m)} N_m(r,s)N_i(r,s) \right) drds \tag{4.30}$$

or by moving the integral into the sum as

$$(\mathcal{S}_r)_{i,j} \equiv \left( \sum_{m=1}^{M} \frac{\partial}{\partial r}N_j(r,s)\Big|_{(r_m,s_m)} \int\int_I N_m(r,s)N_i(r,s) \right) drds \tag{4.31}$$

Each element in $\mathcal{S}_r$ is thus an inner product between the column of a differentiation matrix and the row of a mass matrix (defined next).

By introducing a differentiation matrix of the form

$$\mathcal{D}_{r,(i,j)} \equiv \frac{\partial N_j(r,s)}{\partial r}\Big|_{(r_i,s_i)} \tag{4.32}$$

and a mass matrix

$$\mathcal{M}_{i,j} = \int\int_I N_i(r,s)N_j(r,s)drds \tag{4.33}$$

The mass matrix can be computed from the vandermonde matrix as

$$\mathcal{M} = (\mathcal{V}\mathcal{V}^T)^{-1} \tag{4.34}$$

The mass matrix for the reference domain relates to the $n$'th element through the Jacobian as

$$\mathcal{M}^{(n)} = \mathcal{J}^{(n)}\mathcal{M} \tag{4.35}$$

We can now find the relationship

$$\mathcal{S}_r = \mathcal{M}\mathcal{D}_r \tag{4.36}$$

We can compute $\mathcal{S}_s$ in a similar way as

$$\mathcal{S}_s = \mathcal{M}\mathcal{D}_s \tag{4.37}$$

where $\mathcal{D}_s$ is defined in the obvious way (see definition of $\mathcal{D}_r$ above).

Note that the following matrices can be computed directly from the chosen modal basis; $\mathcal{M}, \mathcal{D}_r, \mathcal{D}_s$. These matrices constitutes *fundamental building stones* for computing various integrals and is conceptually constructed in same way independent of the dimension of the element basis (i.e. 1D/2D/3D).

We are now in a position to compute the element matrix we needed

$$\mathcal{K}^{(n)} = \mathcal{M}^{(n)}\mathcal{D}_x \tag{4.38}$$

with the differentiation matrix defined by the chain rule

$$\mathcal{D}_x = r_x\mathcal{D}_r + s_x\mathcal{D}_s \tag{4.39}$$

and with the geometric weights defined by the property that the mapping between the reference domain and the physical domain is bijective.

The weak formulation (4.16) expressed on the global domain can be expressed in terms of the local contributions by splitting up the integral such that

$$\sum_{n=1}^{N} \int\int_{e_n} \left[\lambda_1 u_x v_x + \lambda_2 u_y v_y\right] dxdy = \sum_{n=1}^{N} \int\int_{e_n} \tilde{q} dxdy \tag{4.40}$$

Each of the local integrals can be expressed in terms of the local basis. For example, if we assume that $\lambda_1$ is constant then we need to understand how to compute integrals of the form

$$\int\int_{e_n} u_x v_x dxdy \tag{4.41}$$

If we express the integrands as a product between two interpolating polynomials we find

$$\int\int_{e_n} u_x v_x dxdy = \mathcal{J}^{(n)} \mathcal{D}_x^T \mathcal{M} \mathcal{D}_x \tag{4.42}$$

where $\mathcal{J}^{(n)}$ is the jacobian of the transformation for the $n$'th element and the differentiation operator is determined using the chain rule

$$\mathcal{D}_x = r_x \mathcal{D}_r + s_x \mathcal{D}_s \tag{4.43}$$

and similar for the $y$-derivative

$$\mathcal{D}_y = r_y \mathcal{D}_r + s_y \mathcal{D}_s \tag{4.44}$$

The element mass matrix is determined as

$$\mathcal{M}^{(n)} = \mathcal{J}^{(n)} \int\int_{\mathcal{I}} N_i^{(n)}(r,s) N_j^{(n)}(r,s) drds = \mathcal{J}^{(n)} \left(\mathcal{V}\mathcal{V}^{\mathcal{T}}\right)^{-1}. \tag{4.45}$$

Thus, the elements of the $\mathbf{A}$ matrix can be determined as

$$a^{(n)} = \lambda_1 \mathcal{J}^{(n)} \mathcal{D}_x^T \mathcal{M} \mathcal{D}_x + \lambda_2 \mathcal{J}^{(n)} \mathcal{D}_y^T \mathcal{M} \mathcal{D}_y \tag{4.46}$$

when $\lambda_1$ and $\lambda_2$ are both constant. Similarly, for the elements of $\mathbf{b}$ we can compute the local elements contributions as

$$b^{(n)} = \int\int_{e_n} v\tilde{q} dxdy = \mathcal{M}^{(n)} \tilde{q}^{(n)} \tag{4.47}$$

## 4.5 Global assembly

The global assembly procedure in two space dimensions are slightly more complicated than in one space dimensions. In the following we will require $C^0$ continuity across all element edges for the global basis functions.

To do this, we need to establish a relationship between the local node and global node numberings. There are several ways of doing this, however, we make the choice that the node numbering is with vertex nodes ordered first, then edge nodes and last the interior nodes. Using Algorithm 14 it is possible to construct a local-to-global array table which holds the global node numbers for each local node and can be used in the global assembly process.

**Algorithm 14:** Determine connectivity table for local-to-global mappings (2D, high-order).

Allocate storage for **c**
gidx = $N_v$ (N.B. $N_v$ is the total number of unique vertex node in the mesh)
$M_P$ = (P+1)(P+2)/2
Mpf = $P+1$ (N.B. Mpf is the number of points on each element face)
**for** $n := 1$ **to** $N$
  **for** $i := 1$ **to** $N_{faces}$
    C[n,i] = EToV[n,i]
    **if** EToE[n,i]≥n
      lidx = $N_{faces}$+(i-1)*(Mpf-2)+(1:Mpf-2)
      C[n,lidx] = gidx+1:gidx+Mpf-2
      gidx = gidx + Mpf - 2;
    **else**
      kc = EToE[n,,i]
      ic = EToF[n,i]
      lidx = $N_{faces}$+(i-1)*(Mpf-2)+(1:Mpf-2)
      C[n,lidx] = C[kc,$N_{faces}$+(ic-1)*(Mpf-2)+(Mpf-2:-1:1)]
    **end**
  lidx = $N_{faces}$+$N_{faces}$(Mpf-2)+1:Mp
  C[n,lidx] = gidx + 1 : gidx + $M_P$ - $N_{faces}$ - $N_{faces}$·(Mpf-2)
  gidx = gidx +$M_P$ - $N_{faces}$ - $N_{faces}$·(Mpf-2)

For the global assembly in two space dimensions we can use Algorithm 15 which takes into account the symmetry of the discrete operator. The local-to-global connectivity table `c` generated using Algorithm 14 can be used in a global assembly process similar to Algorithm 7, however, with the modification that we need to check the indexes to only put elements into the upper triangular part of A for exploiting the symmetry. This is done in Algorithm 15, which can replace Algorithm 7.

**Algorithm 15:** Global assembly of coefficient matrix **A** (2D, high-order, exploit symmetry).

Allocate storage for **A** and **b**
**for** $n := 1$ **to** $N$
  Compute $k_{i,j}^{(n)}$ from (4.46).
  **for** $j := 1$ **to** $M_P$
    **for** $i := 1$ **to** $M_P$
      **if** $C[n,j] \geq C[n,i]$
        $a[C[n,i], C[n,j]] := a[C[n,i], C[n,j]] + k_{i,j}^{(n)}$

In the construction of the right hand side vector b we need to include the mass matrix contribution and this can be achieved using Algorithm 16.

**Algorithm 16:** Global assembly of system right hand side vector **b** (2D, high-order).

**for** $n := 1$ **to** $N$
  Compute $m_{i,j}^{(n)}$ from (4.45).
  **for** $j := 1$ **to** $M_P$
    $jj = C[n, j]$
    $x_j = x[jj]; y_j = x[jj]$
    **for** $i := 1$ **to** $M_P$
      $ii = C[n, i]$
      $b[ii] = b[ii] + m_{i,j}^{(n)} * f[x_j, y_j]$

Once the linear system coefficient matrix and right hand side vector has been built, we need to modify some coefficients to impose the Dirichlet boundary conditions. This can be achieved with Algorithm 17.

**Algorithm 17:** Imposing boundary conditions by modification of system (2D, high-order).

Look up global node numbers for Dirichlet boundary conditions, $i \in S_D$
**for** $i \in S_D$
  Determine global number $i$ for node in question
  Compute right hand side function $f(x_i, y_i)$
  $b = b - f * a[:, i]$
  $a[i, :] = 0; a[:, i] = 0$
  $a[i, i] = 1$
**for** $i \in S_D$
  Compute exact solution $u(x_i, y_i)$ for the boundary node $i$
  $b[i] = u(x(i), y(i))$

# Exercises

## Exercise 4.1

a) Explain all steps in Algorithm 14. Describe what assumptions have been made.

b) Write a Matlab code that implements Algorithm 14 and output the connectivity table `c` for the parameters given for the test case.

**Head:**
```
function [c] = ConstructConnectivityTable2D(x,y,EToV,P)
```

**Test case:**
Create a rectangular mesh using `conelmtab` from Exercise 2.1. Set `noelms1 = 4`, `noelms2 = 3`. $P = 5$.

## Exercise 4.2

a) Update the routine determined in Exercise 2.2 or write a new Matlab routine that implements the assembly process described by Algorithm 15 and 16. *All* nonzeros of the global matrix $A$ should be stored, not just those in the upper triangle. Assume that $\lambda_1(x, y)$ and $\lambda_2(x, y)$ are constants (`lam1` and `lam2`) and that $\tilde{q}(x, y)$ is defined at the mesh nodes by array `qt(1:nonodes)`. Compute the contribution of of $\tilde{q}$ to the right hand side vector $\mathbf{b}$ in an element-wise fashion using discrete integration based on using the local element mass matrices (see 4.47).

b) Repeat tests in Exercise 2.2 for the given test cases.

## Exercise 4.3

a) Update the routine determined in Exercise 2.3 or write a new Matlab routine that implements the updating of the assembled system of equations as described by Algorithm 17.

b) Repeat tests in Exercise 2.3 for the given test cases.

## Exercise 4.4

a) Use the Matlab routines of the previous exercises to write a program that solves the boundary value problem of Exercise 2.4.

b) Repeat tests in Exercise 2.4 for the given test cases for different polynomial orders $P = 1, 2, 3, 4$. Note that for $P = 1$ we should obtain the same results as obtained in Exercise 2.4. Make a plot of the results by plotting the error in the finite element solution versus $\sqrt{\text{nonodes}}$ (for uniform distribution of mesh elements this measure is proportional to the element edge lengths).

## Exercise 4.5

A cylinder with radius R is positioned within a uniform horizontal fluid flow with velocity $U_\infty = 1$. If the fluid flow is potential the velocity field must be divergence free and irrotational. It can then be shown that an analytical solution describing the flow about a very long cylinder can be stated in semi-polar coordinates $(r, \theta)$ as

$$u_r = U_\infty \left[1 - \left(\frac{R}{r}\right)^2\right] \cos\theta, \quad u_\theta = -U_\infty \left[1 + \left(\frac{R}{r}\right)^2\right] \sin\theta$$

In the case of potential flow, the velocity field can be defined in terms of a scalar velocity potential function $\phi$ as

$$\mathbf{u} = \nabla\phi$$

under the additional assumption that the flow is divergence free (mass conservation)

$$\nabla \cdot \mathbf{u} = \nabla^2\phi = 0$$

Therefore, in two space dimensions, the scalar velocity potential function must satsify the famous Laplace equation everywhere

$$\partial_{xx}\phi + \partial_{yy}\phi = 0$$

At the cylinder boundary we can assume that the flow is tangential to the surface, which can be stated as

$$\partial_n\phi = \partial_x\phi n_x + \partial_y\phi n_y = 0, \quad (x, y) \in \partial\Omega_{cyl}$$

where the outward pointing normal vector (to the domain) has been introduced as $\mathbf{n} = (n_x, n_y)$. The analytical solution can be used to define necessary domain boundary conditions.

a) Show that the analytical solution for potential flow about a cylinder is divergence free.

b) Write a Matlab solver that can solve the Laplace equation and visualize the results (e.g. show streamlines and contour plot).

c) Generate a mesh for the problem. (HINT: local refinement near the cylinder can be advantageous to reduce overall errors.)

d) Carry out a $h$- and $p-$convergence study and discuss the results. Relate the discussion to your expectations.

# Chapter 5

# Time-dependent problems

In science and engineering we often encounter problems which are time-dependent. For these types of problems the strategy is to employ a Method of Lines discretization where the spatial terms are discretized independently of the temporal terms. With this basic approach it is standard to employ a FEM-type discretization for the spatial terms and then a finite difference approximation for the temporal term. The basic idea is to turn the PDEs into semi-discrete equations to which a suitable Ordinary Differential Equation (ODE) solver can be applied.

## 5.1 The mathematical formulation

We recall from Section 2.4 that stationary heat conduction in a two-dimensional body is described by the partial differential equation

$$(\lambda_1(x,y)u_x)_x + (\lambda_2(x,y)u_y)_y = -\tilde{q}(x,y)$$

where $\lambda_1(x,y)$ and $\lambda_2(x,y)$ are the heat conductivities with respect to the $x$ and $y$ directions, respectively, and $\tilde{q}(x,y)$, depending on its sign, is a heat source or sink. Let $c(x,y)$ denote the specific heat of the body. Then a simple model of time-dependent (instationary) heat conduction is given by the equation

$$c(x,y)\,u_t = (\lambda_1(x,y)u_x)_x + (\lambda_2(x,y)u_y)_y + \tilde{q}(x,y,t) \tag{5.1}$$

The unknown of the equation is the time-dependent temperature $u(x,y,t)$.

To obtain a well-formulated problem for (5.1) we must supplement this equation with an initial condition and a boundary condition. The initial condition is of the form

$$u(x,y,0) = g(x,y)\,, \quad (x,y) \in \Omega \tag{5.2}$$

The boundary condition may be of any of the three types given in Section 2.5. For simplicity, we will assume that only the inhomogeneous terms in these boundary conditions may be time-dependent. Hence the following possibilities:

*Dirichlet boundary condition:*

$$u = f(x,y,t) \tag{5.3}$$

*Neumann boundary condition:*

$$\lambda_1(x,y)\,u_x\,n_1(x,y) + \lambda_2(x,y)\,u_y\,n_2(x,y) = -q(x,y,t) \tag{5.4}$$

*Robin boundary condition:*

$$\lambda_1(x,y)\,u_x\,n_1(x,y) + \lambda_2(x,y)\,u_y\,n_2(x,y) = -h(x,y)[u - u_0(x,y,t)] \tag{5.5}$$

As before, different boundary conditions may be specified on different parts of the boundary, a frequent occurrence in practice. It will be recalled that in stationary heat conduction the Neumann boundary condition must be combined with one of the other boundary conditions since otherwise the solution is unique only within an additive constant. In the time-dependent case this requirement is dropped because the initial condition (5.2) removes the ambiguity. Thus (5.4) may now be specified on all of $\Gamma$.

## 5.2 The Finite Element Method

Here we will show how the finite element method can be used to solve problem (5.1), (5.2), (5.3), where we assume that (5.3) applies to the whole of $\Gamma$. First we derive the weak formulation of the problem following the approach taken in Section 2.6. Multiplying both sides of (5.1) by a function $v(x,y)$ that vanishes on $\Gamma$ and integrating over $\Omega$ we obtain

$$\int\int_\Omega c\,u_t\,v\,dx\,dy = \int\int_\Omega [(\lambda_1 u_x)_x + (\lambda_2 u_y)_y + \tilde{q}]v\,dx\,dy$$

Using the integral identities from the derivation of (2.19) and the fact that $v$ vanishes on $\Gamma$ we find that

$$\int\int_\Omega c\,u_t\,v\,dx\,dy = -\int\int_\Omega (\lambda_1 u_x v_x + \lambda_2 u_y v_y)\,dx\,dy + \int\int_\Omega \tilde{q}v\,dx\,dy \tag{5.6}$$

The weak formulation of problem (5.1), (5.2), (5.3) is thus the problem of finding a function $u(x,y,t)$ that

1. satisfies (5.6) for all sufficiently smooth $v(x,y)$ such that $v(x,y) = 0$ on $\Gamma$,
2. satisfies (5.2),
3. satisfies (5.3).

In applying the finite element method to this time-dependent problem we seek to approximate the exact solution $u(x,y,t)$ by an approximate solution $\hat{u}(x,y,t)$ of the form

$$\hat{u}(x,y,t) = \sum_{j=1}^{M} \hat{u}_j(t) N_j(x,y) \tag{5.7}$$

The problem hereby becomes that of finding the functions $\hat{u}_i(t)\,,\ t \geq 0\,,\ \ i = 1,2,\ldots,M$. Some of these are determined by the Dirichlet boundary condition (5.3). More precisely, recall that $S_\Omega$ and $S_\Gamma$ are the sets of integers defined by

$$i \in S_\Omega \ \Leftrightarrow\ (x_i,y_i) \in \Omega\,, \qquad i \in S_\Gamma \ \Leftrightarrow\ (x_i,y_i) \in \Gamma\,,$$

and let

$$f_i(t) = f(x_i,y_i,t)\,,\ t \geq 0\,,\quad i \in S_\Gamma \tag{5.8}$$

If we define

$$\hat{u}_i(t) = f_i(t)\,,\quad i \in S_\Gamma \tag{5.9}$$

then it follows from (5.7) and the nature of the shape functions that

$$\hat{u}(x_i,y_i,t) = f(x_i,y_i,t)\,,\ t \geq 0\,,\quad i \in S_\Gamma \tag{5.10}$$

That is, (5.9) imposes the Dirichlet boundary condition on $\hat{u}$ at the boundary nodes.

It remains to determine the functions $u_i(t)\,,\ \ i \in S_\Omega$. To this end we substitute (5.7) for $u$ in (5.6) and let $v = N_i(x,y)$ for all $i \in S_\Omega$. The result, after some straightforward algebra, is

$$\sum_{j=1}^{M} c_{i,j}\frac{d}{dt}\hat{u}_j = -\sum_{j=1}^{M} a_{i,j}\hat{u}_j + b_i(t)\,,\quad t \geq 0\,,\quad i \in S_\Omega \tag{5.11}$$

where

$$c_{i,j} = \int\int_\Omega cN_iN_j\ dx\,dy$$

$$a_{i,j} = \int\int_\Omega [\lambda_1\,(N_i)_x(N_j)_x + \lambda_2\,(N_i)_y(N_j)_y]\,dx\,dy$$

$$b_i(t) = \int\int_\Omega \tilde{q}(x,y,t)N_i\,dx\,dy$$

(5.11) is a system of $M_\Omega$ coupled first-order ordinary differential equations for the determination of the $M_\Omega$ unknown functions $\hat{u}_i(t)\,,\ \ i \in S_\Omega$. We need initial conditions for this system. From the relation

$$\hat{u}(x,y,0) = \sum_{j=1}^{M} \hat{u}_j(0)N_j(x,y)$$

(see (5.7)) and the initial condition (5.2) we see that we must require

$$\hat{u}_i(0) = g(x_i,y_i)\,,\quad i \in S_\Omega \tag{5.12}$$

(5.11) and (5.12) constitute a well-formulated initial value problem for (5.11).

In analogy with our treatment of stationary problems, it turns out to be computationally convenient to expand the ODE system to one having $M$ equations by adding an independent differential equation for every $\hat{u}_i(t)\,,\ i \in S_\Gamma$. Thus on the basis of (5.9) we will supplement (5.11) with the equations

$$\frac{d}{dt}\hat{u}_i = \frac{d}{dt}f_i(t)\,,\quad t \geq 0\,,\quad i \in S_\Gamma \tag{5.13}$$

the corresponding initial conditions being

$$\hat{u}_i(0) = f_i(0)\quad i \in S_\Gamma \tag{5.14}$$

The expanded system is illustrated by the following time-dependent version of example (2.30), where we have $S_\Omega = \{1,3,4\}$ and $S_\Gamma = \{2\}$ :

$$\begin{bmatrix} c_{1,1} & 0 & c_{1,3} & c_{1,4} \\ 0 & 1 & 0 & 0 \\ c_{3,1} & 0 & c_{3,3} & c_{3,4} \\ c_{4,1} & 0 & c_{4,3} & c_{4,4} \end{bmatrix} \begin{bmatrix} \frac{d}{dt}\hat{u}_1 \\ \frac{d}{dt}\hat{u}_2 \\ \frac{d}{dt}\hat{u}_3 \\ \frac{d}{dt}\hat{u}_4 \end{bmatrix} = - \begin{bmatrix} a_{1,1} & 0 & a_{1,3} & a_{1,4} \\ 0 & 0 & 0 & 0 \\ a_{3,1} & 0 & a_{3,3} & a_{3,4} \\ a_{4,1} & 0 & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} + \begin{bmatrix} b_1(t) - a_{1,2}f_2(t) - c_{1,2}\frac{d}{dt}f_2(t) \\ \frac{d}{dt}f_2(t) \\ b_3(t) - a_{3,2}f_2(t) - c_{3,2}\frac{d}{dt}f_2(t) \\ b_4(t) - a_{4,2}f_2(t) - c_{4,2}\frac{d}{dt}f_2(t) \end{bmatrix} \tag{5.15}$$

Defining

$$\hat{\boldsymbol{u}}(t) = [\hat{u}_1(t), \hat{u}_2(t), \ldots, \hat{u}_M(t)]^T$$

we can express this system in the general case by

$$\boldsymbol{C}\frac{d}{dt}\hat{\boldsymbol{u}} = -\boldsymbol{A}\hat{\boldsymbol{u}} + \boldsymbol{b}(t)\,,\quad t \geq 0 \tag{5.16}$$

$\boldsymbol{C}$ is symmetric positive definite (and hence nonsingular), while $\boldsymbol{A}$ is symmetric positive semidefinite and singular. Both matrices are constant and have roughly the same distribution of nonzero entries.

It is also convenient to write (5.16) in the standard form for first-order ODE systems

$$\frac{d}{dt}\hat{\boldsymbol{u}} = \boldsymbol{F}(t, \hat{\boldsymbol{u}}), \quad t \geq 0 \tag{5.17}$$

where, for our problem

$$\boldsymbol{F}(t, \hat{\boldsymbol{u}}) = \boldsymbol{C}^{-1}[-\boldsymbol{A}\hat{\boldsymbol{u}} + \boldsymbol{b}(t)] \tag{5.18}$$

In actual implementations we should avoid computing $\boldsymbol{C}^{-1}$ as it is computationally expensive. Instead, when $\boldsymbol{C}$ is time-constant it is in general better to compute the factorization once and reuse this in subsequent steps.

The initial condition for the expanded ODE system is

$$\hat{\boldsymbol{u}}(0) = \hat{\boldsymbol{u}}_0 \tag{5.19}$$

where we define

$$(\hat{\boldsymbol{u}}_0)_i = \begin{cases} g(x_i, y_i), & i \in S_\Omega \\ f_i(0), & i \in S_\Gamma \end{cases}$$

## 5.3 A time-stepping procedure

Since many problems in science and technology (and a number of other areas, as well) give rise to initial value problems having the general form of (5.17), (5.19), much effort has been expended in developing numerical methods that solve problems of this type. For information about these the reader is urged to consult, for example, [40] and [42]. Here we restrict attention to the simple one-parameter family of methods defined by

$$\frac{1}{\Delta t}(\hat{\boldsymbol{u}}_{n+1} - \hat{\boldsymbol{u}}_n) = \theta \boldsymbol{F}(t_{n+1}, \hat{\boldsymbol{u}}_{n+1}) + (1-\theta)\boldsymbol{F}(t_n, \hat{\boldsymbol{u}}_n), \quad n = 0, 1, 2, \ldots \tag{5.20}$$

where the parameter $\theta$ is restricted to the interval $0 \leq \theta \leq 1$. We assume for simplicity that the time step $\Delta t$ is constant. The time points are thus $t_n = n\,\Delta t$, $n = 0, 1, 2, \ldots$, and the vector $\hat{\boldsymbol{u}}_{n+1}$ produced by (5.20) is an approximation of $\hat{\boldsymbol{u}}(t_{n+1})$. This method has several names, depending on the value of $\theta$:

| | |
|---|---|
| $\theta = 0$; | *The forward Euler method* |
| $\theta = 1/2$; | *The trapez method* (or *Crank-Nicolson method*) |
| $\theta = 1$; | *The backward Euler method* |

To apply (5.20) to our problem we use the expression for $\boldsymbol{F}$ in (5.18), obtaining

$$\frac{1}{\Delta t}(\hat{\boldsymbol{u}}_{n+1} - \hat{\boldsymbol{u}}_n) = \theta \boldsymbol{C}^{-1}[-\boldsymbol{A}\hat{\boldsymbol{u}}_{n+1} + \boldsymbol{b}(t_{n+1})] + (1-\theta)\boldsymbol{C}^{-1}[-\boldsymbol{A}\hat{\boldsymbol{u}}_n + \boldsymbol{b}(t_n)]$$

Multiplying both sides by $\Delta t\, \boldsymbol{C}$ yields then

$$(\boldsymbol{C} + \Delta t\, \theta \boldsymbol{A})\hat{\boldsymbol{u}}_{n+1} = [\boldsymbol{C} - \Delta t\,(1-\theta)\boldsymbol{A}]\hat{\boldsymbol{u}}_n + \Delta t\, \theta\, \boldsymbol{b}(t_{n+1}) + \Delta t\,(1-\theta)\, \boldsymbol{b}(t_n) \tag{5.21}$$

This is a linear algebraic system of equations that must be solved for $n = 0, 1, 2, \ldots$. The coefficient matrix $(\boldsymbol{C} + \Delta t\, \theta \boldsymbol{A})$ is symmetric positive definite and constant with respect to time.

Before examining the computational details of (5.21), which is the topic of the next section, we make a theoretical analysis of this method when it is applied to a model problem. Thus suppose

that $\tilde{q}(x,y,t) = 0$ in (5.1) and $f(x,y,t) = 0$ in (5.3). The initial value problem (5.16), (5.19) then becomes

$$\frac{d}{dt}\hat{\boldsymbol{u}} = -\boldsymbol{D}\hat{\boldsymbol{u}}\,, \quad t \geq 0\,, \qquad \hat{\boldsymbol{u}}(0) = \hat{\boldsymbol{u}}_0 \tag{5.22}$$

where $\boldsymbol{D} = \boldsymbol{C}^{-1}\boldsymbol{A}$. Further, (5.21) is equivalent to

$$\hat{\boldsymbol{u}}_{n+1} = \boldsymbol{E}\hat{\boldsymbol{u}}_n\,, \quad n = 0, 1, 2, \ldots \tag{5.23}$$

where

$$\boldsymbol{E} = (\boldsymbol{I} + \Delta t\,\theta\boldsymbol{D})^{-1}\,[\boldsymbol{I} - \Delta t\,(1-\theta)\boldsymbol{D}]$$

We seek first the analytical solution of (5.22) and introduce for this purpose the eigenvalues and eigenvectors of $\boldsymbol{D}$:

$$\boldsymbol{D}\boldsymbol{v}_i = \xi_i\boldsymbol{v}_i\,, \quad i = 1, 2, \ldots, M$$

A study of $\boldsymbol{D}$, which we omit, shows that the eigenvalues are real and non-negative and can be ordered by

$$0 = \xi_1 = \xi_2 = \ldots = \xi_{M_\Gamma} < \xi_{M_\Gamma+1} \leq \xi_{M_\Gamma+2} \cdots \leq \xi_M$$

(The presence of the zero eigenvalue with multiplicity $M_\Gamma$ is the result of including (5.13) in the ODE system). Further, we can choose to expand the initial vector $\hat{\boldsymbol{u}}_0$ in terms of the eigenvectors of $\boldsymbol{D}$ in the form

$$\hat{\boldsymbol{u}}_0 = \sum_{i=M_\Gamma+1}^{M} \alpha_i\boldsymbol{v}_i \tag{5.24}$$

If $\boldsymbol{D}$ is a diagonalizable matrix, we can decompose in terms of matrix holding the eigenvectors in columns $\mathbf{V}$ and a diagonal matrix $\boldsymbol{\Lambda}$ holding eigenvalues as

$$\boldsymbol{D} = \mathbf{V}^{-1}\boldsymbol{\Lambda}\mathbf{V} \tag{5.25}$$

which can be inserted in (5.22) and by a change of basis and introducing $z = \mathbf{V}\hat{u}$ we find

$$\frac{d}{dt}\mathbf{z} = \boldsymbol{\Lambda}\mathbf{z}, \quad t \geq 0, \mathbf{z}(0) = z_0 = \mathbf{V}\hat{u}_0 \tag{5.26}$$

The exact solution to each of these individual equations are by the separation of variables method found to be

$$z_i(t) = z_i(0)e^{-\xi_i}, \quad i = 1, 2, ..., M \tag{5.27}$$

By transformation back to the original basis we find that we can express the solution as

$$\hat{\boldsymbol{u}}(t) = \sum_{i=M_\Gamma+1}^{M} \alpha_i e^{-\xi_i t}\boldsymbol{v}_i\,, \quad t \geq 0 \tag{5.28}$$

An important property of this solution is that, because of the positive signs of the $\xi_i$, each term expresses smooth exponential decay with respect to time and

$$\lim_{t\to\infty} \hat{\boldsymbol{u}}(t) = \mathbf{0}$$

Physically, since there is no heat source and the temperature is held at zero at the boundary of the body, all heat eventually escapes. We note that for $t = t_n = n\,\Delta t$ the solution may be written as

$$\hat{\boldsymbol{u}}(t_n) = \sum_{i=M_\Gamma+1}^{M} \alpha_i\eta_i^n\boldsymbol{v}_i\,, \quad n = 0, 1, 2, \ldots \tag{5.29}$$

where

$$\eta_i = e^{-\xi_i\,\Delta t}\,, \quad i = M_\Gamma{+}1, \ldots, M \tag{5.30}$$

Having found the analytical solution of (5.22), we now seek that of (5.23). A basic matrix theorem allows us to relate the eigenvalues of $\boldsymbol{E}$ to those of $\boldsymbol{D}$ as follows:

$$\boldsymbol{E}\boldsymbol{v}_i = \tilde{\eta}_i \boldsymbol{v}_i\,, \quad i = 1, 2, \ldots, M$$

where

$$\tilde{\eta}_i = \frac{1 - (1-\theta)\xi_i \Delta t}{1 + \theta \xi_i \Delta t} \tag{5.31}$$

Then, by (5.24),

$$\hat{\boldsymbol{u}}_n = \boldsymbol{E}^n\, \hat{\boldsymbol{u}}_0 = \sum_{i=M_\Gamma+1}^{M} \alpha_i \boldsymbol{E}^n\, \boldsymbol{v}_i$$

and, since $\boldsymbol{E}^n\, \boldsymbol{v}_i = \tilde{\eta}_i^n\, \boldsymbol{v}_i$ , we have as our result

$$\hat{\boldsymbol{u}}_n = \sum_{i=M_\Gamma+1}^{M} \alpha_i \tilde{\eta}_i^n\, \boldsymbol{v}_i\,, \quad n = 0, 1, 2, \ldots \tag{5.32}$$

The $i$th term of this expansion decays with time if and only if $|\tilde{\eta}_i| < 1$ , so assuming all $\alpha_i$ are nonzero we have

$$\lim_{n\to} \hat{\boldsymbol{u}}_n = \boldsymbol{0} \quad \Leftrightarrow \quad |\tilde{\eta}_i| < 1\,, \quad i = M_\Gamma{+}1, \ldots, M$$

The $i$th term can decay ›th decay, similar to that described b 0 then it describes 'nonphysical' osc

Comparing (5.32) with effectively approximating $\eta_i$ by $\tilde{\eta}_i$ . $\tilde{\eta}_i)\}$ , $\theta =$ $0, 1/2, 1$ , where $x_i = \xi_i \Delta$ ıe typical

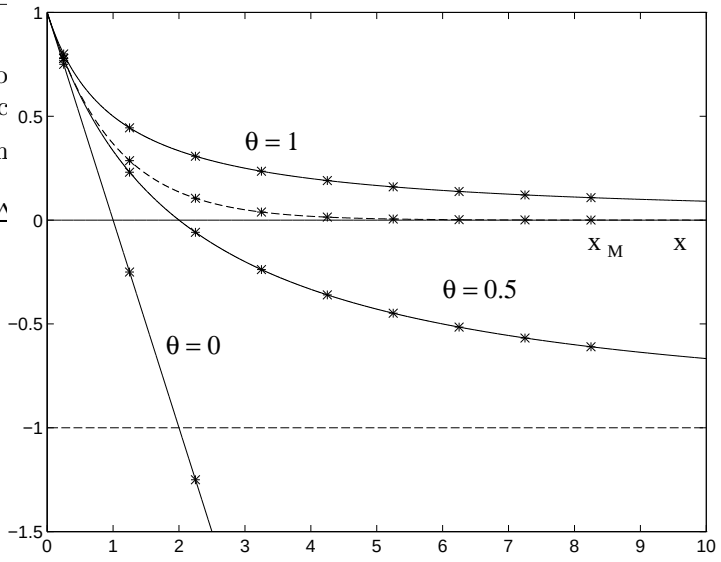



Figure 5.1: The point sets $\{x_i, \eta_i\}$ (on the dashed line) and $\{x_i, \tilde{\eta}_i\}$ for $\theta = 0, 0.5, 1$ .

situation in practice is

$$M_\Omega = \mathcal{O}(h^{-2})\,, \quad \xi_{M_\Gamma+1} = \mathcal{O}(h^0)\,, \quad \xi_M = \mathcal{O}(h^{-2}) >> 1 \tag{5.33}$$

where $h$ is the average element edge length. Thus there can be many values of $x_i$, and, depending on $\Delta t$, they can extend across a large interval. Note, however, that reducing $\Delta t$ has the effect of sliding all points leftward along their respective curves toward the point (0,1). Obviously, for any value of $\theta$ we can make all of the differences $|\tilde{\eta}_i - \eta_i|$ as small as we please simply by choosing a sufficiently small value of $\Delta t$. However, "sufficiently small" may turn out to be too small for practical computation. Fortunately, it is usually sufficient to well approximate only the *leading* values of $\{\eta_i\}$. This is because that when the function $g(x,y)$ in (5.2) is reasonably smooth then the trailing end of the sequence $\{\alpha_i\}$ consists of very small values. (Recall that the $\alpha_i$ come from (5.24), which may be viewed as a discrete Fourier-type expansion of $g(x,y)$ ). The trailing end of the expansion in (5.32) will then have little effect on $\hat{\boldsymbol{u}}_n$ provided we make sure that $|\tilde{\eta}_i| \leq 1$ for all $i$.

We now consider the three special cases of $\theta$ in greater detail:

**The case $\theta = 0$**
a) *Regarding $\tilde{\eta}_i$ for small values of $\xi_i \Delta t$:*
Expanding (5.31) and (5.30) in power series yields the result

$$\tilde{\eta}_i - \eta_i = -\frac{1}{2}(\xi_i \Delta t)^2 + \mathcal{O}((\xi_i \Delta t)^3)$$

b) *Regarding $\tilde{\eta}_i$ for large values of $\xi_i \Delta t$:*
We see from Figure 5.1 that if $\xi_M \Delta t > 2$ then $\tilde{\eta}_M < -1$ and $|\tilde{\eta}_M| \geq |\tilde{\eta}_i|$, $i \neq M$. Supposing for simplicity that $\xi_M$ is a simple eigenvalue of $\boldsymbol{D}$, $\tilde{\eta}_M$ is then the unique dominant eigenvalue of $\boldsymbol{E}$ and

$$\hat{\boldsymbol{u}}_n \approx \alpha_M \tilde{\eta}_M^n \boldsymbol{v}_M, \quad n >> 0$$

This $\hat{\boldsymbol{u}}_n$ exhibits oscillations whose amplitude *grows without bound.* To avoid this disaster we must require that

$$\Delta t \leq 2/\xi_M \tag{5.34}$$

Unfortunately, according to (5.33) this makes $\Delta t = \mathcal{O}(h^2)$, which is often too small for practical computation.

c) *Accuracy:* $\| \hat{\boldsymbol{u}}_{t/\Delta t} - \hat{\boldsymbol{u}}(t) \|_\infty = \mathcal{O}(\Delta t)$

d) *Usefulness:* Limited.

**The case $\theta = 1/2$**
a) *Regarding $\tilde{\eta}_i$ for small values of $\xi_i \Delta t$:*

$$\tilde{\eta}_i - \eta_i = -\frac{1}{12}(\xi_i \Delta t)^3 + \mathcal{O}((\xi_i \Delta t)^4)$$

b) *Regarding $\tilde{\eta}_i$ for large values of $\xi_i \Delta t$:*
We see from Figure 5.1 that $|\tilde{\eta}_i| < 1$ for all $i$ so the above-mentioned disaster cannot occur. However, one should be aware that if $\Delta t > 2/\sqrt{\xi_{M_\Gamma+1}\xi_M}$ then $\tilde{\eta}_M$ is the eigenvalue of maximum absolute value, and we again have

$$\hat{\boldsymbol{u}}_n \approx \alpha_M \tilde{\eta}_M^n \boldsymbol{v}_M, \quad n >> 0$$

This relation now describes oscillations that decay, *but perhaps only very slowly.*

c) *Accuracy:* $\| \hat{\boldsymbol{u}}_{t/\Delta t} - \hat{\boldsymbol{u}}(t) \|_\infty = \mathcal{O}((\Delta t)^2)$

d) *Usefulness:* Because of its second-order accuracy, this method is usually the most effective of the three under consideration when relatively high accuracy is required. However, there is the possibility of persistent low-level oscillations.

**The case** $\theta = 1$
a) *Regarding* $\tilde{\eta}_i$ *for small values of* $\xi_i\,\Delta t$:

$$\tilde{\eta}_i - \eta_i = \frac{1}{2}(\xi_i\,\Delta t)^2 + \mathcal{O}((\xi_i\,\Delta t)^3)$$

b) *Regarding* $\tilde{\eta}_i$ *for large values of* $\xi_i\,\Delta t$:
Since $0 < \tilde{\eta}_i < 1$ for all $i$, the approximate solution $\hat{\boldsymbol{u}}_n$ will always exhibit smooth decay (even when $g(x, y)$ is unsmooth).

c) *Accuracy:* $\| \hat{\boldsymbol{u}}_{t/\Delta t} - \hat{\boldsymbol{u}}(t) \|_\infty = \mathcal{O}(\Delta t)$

d) *Usefulness:* Because of its 'robustness', this method is often attractive if only modest accuracy is needed.

## 5.4 The computation

**The case** $f = f(x, y)$.

When the function $f$ in the Dirichlet boundary condition is independent of time, the ODE system (5.16) is illustrated by

$$\begin{bmatrix} c_{1,1} & 0 & c_{1,3} & c_{1,4} \\ 0 & 1 & 0 & 0 \\ c_{3,1} & 0 & c_{3,3} & c_{3,4} \\ c_{4,1} & 0 & c_{4,3} & c_{4,4} \end{bmatrix} \begin{bmatrix} \frac{d}{dt}\hat{u}_1 \\ \frac{d}{dt}\hat{u}_2 \\ \frac{d}{dt}\hat{u}_3 \\ \frac{d}{dt}\hat{u}_4 \end{bmatrix} = - \begin{bmatrix} a_{1,1} & 0 & a_{1,3} & a_{1,4} \\ 0 & 0 & 0 & 0 \\ a_{3,1} & 0 & a_{3,3} & a_{3,4} \\ a_{4,1} & 0 & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} + \begin{bmatrix} b_1(t) - a_{1,2} f_2 \\ 0 \\ b_3(t) - a_{3,2} f_2 \\ b_4(t) - a_{4,2} f_2 \end{bmatrix} \tag{5.35}$$

where $f_2$ is a constant. (See (5.15)). Associated with the second equation in this system is the uncoupled initial value problem

$$\frac{d}{dt}\hat{u}_2 = 0\,,\ t \geq 0\,, \quad \hat{u}_2(0) = f_2$$

which has the constant solution $\hat{u}_2(t) = f_2\,,\ t \geq 0\,.$

The computational problem that must be dealt with is that of constructing and solving the linear systems in (5.21). These can be expressed as

$$\boldsymbol{R}\,\hat{\boldsymbol{u}}_{n+1} = \boldsymbol{e}_{n+1}\,, \quad n = 0, 1, 2, \ldots \tag{5.36}$$

where

$$\boldsymbol{e}_{n+1} = \boldsymbol{S}\,\hat{\boldsymbol{u}}_n + d_1\,\boldsymbol{b}(t_{n+1}) + d_2\,\boldsymbol{b}(t_n) \tag{5.37}$$

$$\boldsymbol{R} = \boldsymbol{C} + d_1\boldsymbol{A}\,, \quad \boldsymbol{S} = \boldsymbol{C} - d_2\boldsymbol{A}\,, \quad d_1 = \Delta t\,\theta\,, \quad d_2 = \Delta t\,(1-\theta) \tag{5.38}$$

We note for future reference the relation

$$\boldsymbol{R} = \boldsymbol{S} + \Delta t\,\boldsymbol{A} \tag{5.39}$$

The computation can be broken down into a sequence of steps which are formulated below in terms of a pseudo-programming language as illustrated in the following. Use is made of the following arrays:

| Name | Dimensions | Contents |
|---|---|---|
| $r$ | $M \times M$ | $\boldsymbol{A}$ and $\boldsymbol{R}$ |
| $s$ | $M \times M$ | $\boldsymbol{C}$ and $\boldsymbol{S}$ |
| $b$ | $M \times 1$ | $\boldsymbol{b}(t_n)\,,\ n = 0, 1, \ldots$ |
| $d$ | $M \times 1$ | The time-independent part of $\boldsymbol{b}(t_n)$ |
| $e$ | $M \times 1$ | $\boldsymbol{e}_{n+1}\,,\ n = 0, 1, \ldots$ |

An $M \times 1$ array is also needed for the solution vector $\hat{\boldsymbol{u}}_{n+1}$. It will be seen that the code exploits the symmetry of the problem; the entries addressed in $r$ and $s$ are all on or above the main diagonals of these arrays.

The use of $M \times M$ arrays has the advantage of making it easier to show what needs to be computed. It should be emphasized, however, that a practical implementation of the method would make use of data structures for symmetric sparse matrices.

The computation begins with the following initialization steps:

**Step 1:**

**Algorithm 18:** Step 1. Global assembly of system matrices (2D, time-dependent).

Allocate storage for $r$, $s$ and $\mathbf{b}$
**for** $nn := 1$ **to** $N$
  Look up global index numbers and coordinates of nodes in element $e_{nn}$.
  **for** $rr := 1$ **to** 3
    Compute $\tilde{q}_{rr}^{(nn)}(0)$ from (2.28).
    $b[i] := b[i] + \tilde{q}_{rr}^{(nn)}(0)$
    **for** $ss := rr$ **to** 3
      Compute $k_{rr,ss}^{(nn)}$ and $c_{rr,ss}^{(nn)}$ from (2.27) and (5.40), respectively.
      **if** $j \geq i$
        $r[i,j] := r[i,j] + k_{rr,ss}^{(nn)}$, $s[i,j] := s[i,j] + c_{rr,ss}^{(nn)}$
      **else**
        $r[j,i] := r[j,i] + k_{rr,ss}^{(nn)}$, $s[j,i] := s[j,i] + c_{rr,ss}^{(nn)}$

Step 1, which is very similar to Algorithm 4 in Section 2.8, assembles matrix $\boldsymbol{A}$ in array $r$ and matrix $\boldsymbol{C}$ in array $s$. It also assembles in array $b$ that part of the vector $\boldsymbol{b}(0)$ which is due to the source function $\tilde{q}(x, y, 0)$. In this step the Dirichlet boundary condition is ignored.

**Step 2:**

**Algorithm 19:** Step 2. Global assembly of system matrices (2D, time-dependent).

Allocate storage for $\mathbf{d}$
**for** $i \in S_\Gamma$
  $r[i,i] := 0$, $s[i,i] := 1$, $b[i] := 0$, $d[i] := 0$
  **for** $j \in S_\Omega^{(i)}$
    **if** $j < i$
      $temp = r[j,i] * f_i$
      $b[j] := b[j] - temp$, $d[j] := d[j] + temp$
      $r[j,i] := 0$, $s[j,i] := 0$
    **else if** $j > i$
      $temp = r[i,j] * f_i$
      $b[j] := b[j] - temp$, $d[j] := d[j] + temp$
      $r[i,j] := 0$, $s[i,j] := 0$

Step 2 modifies $\boldsymbol{A}$, $\boldsymbol{C}$ and $\boldsymbol{b}(0)$ on the basis of the Dirichlet boundary condition.

**Step 3:** $s := s - d_2 * r$, $r := s + \Delta t * r$

Step 3 puts matrix $\boldsymbol{S}$ in array $s$, overwriting $\boldsymbol{C}$, and matrix $\boldsymbol{R}$ in array $r$, overwriting $\boldsymbol{A}$. (See (5.38) and (5.39)).

**Step 4:** Factor $\boldsymbol{R}$ and overwrite $\boldsymbol{R}$ with this factor.

This is relevant only if a direct method (Gaussian elimination, for example) is to be used to solve the systems in (5.36). (See below).

Step 4 concludes the initialization part of the computation. The steps that follow are performed for $n = 0, 1, 2 \ldots$:

**Step 5:** $e := \boldsymbol{S}\,\hat{\boldsymbol{u}}_n$

This computes the first term of vector $\boldsymbol{e}_{n+1}$ (see (5.37)) and stores it in array $e$.

**Step 6:** $e := e + d_2 * b$

This adds the third term, $d_2\,\boldsymbol{b}(t_n)$. Still missing is the second term, $d_1\,\boldsymbol{b}(t_{n+1})$.

**Step 7:**

---

**Algorithm 20:** Step 7. Modification of system vector (2D, time-dependent).

---

Allocate storage for **b**.
**for** $nn := 1$ **to** $N$
  Look up global numbers and coordinates of nodes in element $e_{nn}$.
  **for** $rr := 1$ **to** 3
    Compute $\tilde{q}_{rr}^{(nn)}(t_{n+1})$ from (2.28).
    $b[i] := b[i] + \tilde{q}_{rr}^{(nn)}(t_{n+1})$

---

Step 7 begins the computation of $\boldsymbol{b}(t_{n+1})$ in array $b$, overwriting $\boldsymbol{b}(t_n)$. It ignores the boundary condition.

**Step 8**

---

**Algorithm 21:** Step 8. Modification of system vector (2D, time-dependent).

---

**for** $i \in S_\Gamma$
  $b[i] := 0$
$b = b - d$

---

Step 8 ends the computation of $\boldsymbol{b}(t_{n+1})$ by incorporating the boundary condition.

**Step 9:** $e := e + d_1 * b$

Step 9 ends the computation of $\boldsymbol{e}_{n+1}$.

**Step 10:** Solve (5.36).

Regarding the assembly of matrix $\boldsymbol{C}$ in Step 1, let us recall that the typical entry of this matrix is

$$c_{i,j} = \int\int_\Omega cN_iN_j\,dx\,dy$$

To assemble $\boldsymbol{C}$ we introduce for every element $e_n$ an element matrix with entries

$$c_{r,s}^{(n)} = \int\int_{e_n} cN_r^{(n)}N_s^{(n)}\,dx\,dy\,, \quad r,s = 1,2,3$$

An analysis, which we omit, shows that when the specific heat $c$ is constant then

$$\begin{bmatrix} c_{1,1}^{(n)} & c_{1,2}^{(n)} & c_{1,3}^{(n)} \\ c_{2,1}^{(n)} & c_{2,2}^{(n)} & c_{2,3}^{(n)} \\ c_{3,1}^{(n)} & c_{3,2}^{(n)} & c_{3,3}^{(n)} \end{bmatrix} = \frac{c\,|\Delta|}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \tag{5.40}$$

where $\Delta$ is given by (2.1). If $c$ is not constant it can be approximated by its value at the center of the triangle or by the average of its values at the three nodes.

The symmetric positive definite systems in (5.36) may be solved by either a direct or iterative method, and the discussion at the end of Section 2.8 is applicable also here. However, for each of these types of methods there are special considerations for time-dependent problems, where not just one but usually many systems must be solved. Regarding first direct methods, these effectively decompose the coefficient matrix $\boldsymbol{R}$ into factors, and it is important to exploit these. In the case of symmetric Gaussian elimination, for example, the factorization is

$$\boldsymbol{R} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$$

where $\boldsymbol{L}$ and $\boldsymbol{D}$ have lower triangular and diagonal form, respectively. Thus the typical system in (5.36) can be expressed as

$$\boldsymbol{L}(\boldsymbol{D}\boldsymbol{L}^T\hat{\boldsymbol{u}}_{n+1}) = \boldsymbol{e}_{n+1} \tag{5.41}$$

and this can be solved by solving, successively, the lower triangular system $\boldsymbol{L}\tilde{\boldsymbol{e}}_{n+1} = \boldsymbol{e}_{n+1}$ and the upper triangular system $\boldsymbol{L}^T\hat{\boldsymbol{u}}_{n+1} = \boldsymbol{D}^{-1}\tilde{\boldsymbol{e}}_{n+1}$. Since this procedure is usually much faster than the computation of $\boldsymbol{L}$ and $\boldsymbol{D}$, it is important to keep these matrices. (A re-factorization of $\boldsymbol{R}$ is necessary, however, if the time step $\Delta t$ is changed during the computation.)

With regard to iterative methods, we note that in time-dependent problems there is a natural choice of the initial vector in solving the system $\boldsymbol{R}\hat{\boldsymbol{u}}_{n+1} = \boldsymbol{e}_{n+1}$; namely, $\hat{\boldsymbol{u}}_n$.

**The case** $f = f(x, y, t)$.

When $f = f(x, y, t)$, the vector $\boldsymbol{b}(t)$ in the ODE system (5.16) is illustrated by

$$\boldsymbol{b}(t) = \begin{bmatrix} b_1(t) - a_{1,2}f_2(t) - c_{1,2}f_2'(t) \\ f_2'(t) \\ b_3(t) - a_{3,2}f_2(t) - c_{3,2}f_2'(t) \\ b_4(t) - a_{4,2}f_2(t) - c_{4,2}f_2'(t) \end{bmatrix}$$

(See (5.15). More generally, the coefficients appearing in $\boldsymbol{b}(t)$ are

$$a_{i,j}\,,\quad c_{i,j}\,,\quad j \in S_\Gamma\,,\ i \in S_\Omega^{(j)} \tag{5.42}$$

Exploiting symmetry we may write $a_{i,j}$ and $c_{i,j}$ as $a_{j,i}$ and $c_{j,i}$, respectively, when $j < i$, so all of the coefficients in (5.42) can be found in the upper triangles of arrays $r$ and $s$ at the end of Step 1 above. However, since they would be overwritten by the implementation of the Dirichlet boundary condition and since they are needed in each time step, they must be saved.

The new computation involves replacing Steps 2 and 8 above by Steps 2A and 8A below. Adding Step 11 enforces the correct solution at the boundary nodes.

**Step 2A:**

**Algorithm 22:** Step 2A. Modification of system matrices (2D, time-dependent).

**for** $i \in S_\Gamma$
  $r[i,i] := 0\,,\ \ s[i,i] := 1\,,\ \ b[i] := 0$
  **for** $j \in S_\Omega^{(i)}$
    **if** $j < i$
      Store the contents of $r[j,i]$ (i.e. $a_{j,i}$) and $s[j,i]$ (i.e. $c_{j,i}$) in another data structure.
      $b[j] := b[j] - r[j,i] * f_i(0) - s[j,i] * f_i'(0)$
      $r[j,i] := 0\,,\ \ s[j,i] := 0$
    **else**
      Store the contents of $r[i,j]$ (i.e. $a_{i,j}$) and $s[i,j]$ (i.e. $c_{i,j}$) in another data structure.
      $b[j] := b[j] - r[i,j] * f_i(0) - s[i,j] * f_i'(0)$
      $r[i,j] := 0\,,\ \ s[i,j] := 0$

**Step 8A**

**Algorithm 23:** Step 8A. Modification of system vector (2D, time-dependent).

**for** $i \in S_\Gamma$
  $b[i] := 0$
  **for** $j \in S_\Omega^{(i)}$
    **if** $j < i$
      $b[j] := b[j] - a_{j,i}\, f_i(t_{n+1}) - c_{j,i}\, f_i'(t_{n+1})$
    **else**
      $b[j] := b[j] - a_{i,j}\, f_i(t_{n+1}) - c_{i,j}\, f_i'(t_{n+1})$

**Step 11:**

**Algorithm 24:** Step 11. Modification of system vector (2D, time-dependent).

**for** $i \in S_\Gamma$
  $(\hat{\boldsymbol{u}}_{n+1})_i := f_i(t_{n+1})$

Insight into the computation can be gained by considering the case when the interior nodes of the mesh are ordered before the boundary nodes. Then (5.16) can be expressed in the partitioned form

$$\begin{bmatrix} \boldsymbol{C}_{1,1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \frac{d}{dt}\hat{\boldsymbol{u}}_\Omega \\ \frac{d}{dt}\hat{\boldsymbol{u}}_\Gamma \end{bmatrix} = - \begin{bmatrix} \boldsymbol{A}_{1,1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}}_\Omega \\ \hat{\boldsymbol{u}}_\Gamma \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_\Omega(t) \\ \frac{d}{dt}\boldsymbol{f}_\Gamma(t) \end{bmatrix} \tag{5.43}$$

For example, if the node ordering associated with (5.15) were such that $S_\Omega = \{1,2,3\}$ and $S_\Gamma = \{4\}$, then (5.15) would become the following special case of (5.43):

$$\begin{aligned} \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & 0 \\ c_{2,1} & c_{2,2} & c_{2,3} & 0 \\ c_{3,1} & c_{3,2} & c_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{d}{dt}\hat{u}_1 \\ \frac{d}{dt}\hat{u}_2 \\ \frac{d}{dt}\hat{u}_3 \\ \frac{d}{dt}\hat{u}_4 \end{bmatrix} = & - \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \end{bmatrix} \\ & + \begin{bmatrix} b_1(t) - a_{1,4} f_4(t) - c_{1,4} \frac{d}{dt} f_4(t) \\ b_2(t) - a_{2,4} f_4(t) - c_{2,4} \frac{d}{dt} f_4(t) \\ b_3(t) - a_{3,4} f_4(t) - c_{3,4} \frac{d}{dt} f_4(t) \\ \frac{d}{dt} f_4(t) \end{bmatrix} \end{aligned} \tag{5.44}$$

From the definitions of $\boldsymbol{R}$ and $\boldsymbol{S}$ in (5.38) we see that these matrices now assume the partitioned form

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{1,1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}, \quad \boldsymbol{S} = \begin{bmatrix} \boldsymbol{S}_{1,1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}, \tag{5.45}$$

This allows (5.21) to be decoupled as follows:

$$\boldsymbol{R}_{1,1}(\boldsymbol{u}_\Omega)_{n+1} = \boldsymbol{S}_{1,1}(\boldsymbol{u}_\Omega)_n + \Delta t\,\theta\,\boldsymbol{b}_\Omega(t_{n+1}) + \Delta t\,(1-\theta)\,\boldsymbol{b}_\Omega(t_n) \tag{5.46}$$

$$(\boldsymbol{u}_\Gamma)_{n+1} = (\boldsymbol{u}_\Gamma)_n + \Delta t\,\theta\,\boldsymbol{f}'_\Gamma(t_{n+1}) + \Delta t\,(1-\theta)\,\boldsymbol{f}'_\Gamma(t_n) \tag{5.47}$$

The key observation is that when $\boldsymbol{f}'_\Gamma \neq \mathbf{0}$ then (5.47) introduces a discretization error in $(\boldsymbol{u}_\Gamma)_{n+1}$. Since we know $(\boldsymbol{u}_\Gamma)_{n+1}$ exactly from the boundary condition, we overwrite the value of $(\boldsymbol{u}_\Gamma)_{n+1}$ computed by the equation solver by performing Step 11 above. Further, since we do not use the value of $(\boldsymbol{u}_\Gamma)_{n+1}$ computed by the equation solver, we can replace $\boldsymbol{f}'_\Gamma$ in (5.47) by the zero vector. Thus at the beginning of Steps 2A and 8A we put $b[i] := 0$ for $i \in S_\Gamma$ instead of $b[i] := f'_i(0)$ in Step 2A and $b[i] := f'_i(t_{n+1}))$ in Step 8A.

Step 11 is not needed when $f = f(x, y)$ since in this case the equation solver makes $(\boldsymbol{u}_\Gamma)_{n+1} = (\boldsymbol{u}_\Gamma)_n$ with the result that we have $(\boldsymbol{u}_\Gamma)_{n+1} = (\boldsymbol{u}_\Gamma)_0$ for $n = 0, 1, 2 \ldots$, as required.

# Exercises

## Exercise 5.1

a) Modify subroutine `assembly` into a subroutine that implements Step 1 in Section 5.4.

b) Modify subroutine `dirbc` into a subroutine that implements Step 2.

## Exercise 5.2

a) Write a program that solves the following problem

$$\begin{aligned} u_t &= \nu\,(u_{xx}+u_{yy})\,, \quad (x,y)\in\Omega\,, \quad t>0 \\ u(x,y,0) &= \sin(\pi x)\sin(\pi y)\,, \quad (x,y)\in\Omega \\ u(x,y,t) &= 0\,, \quad (x,y)\in\Gamma\,, \quad t\geq 0 \end{aligned}$$

where $\nu=\lambda/c$, a positive constant. The domain is the unit square $0\leq x,y\leq 1$. The analytical solution is

$$u(x,y,t)=e^{-2\pi^2\nu t}\sin(\pi x)\sin(\pi y)$$

Use the computational procedure given in Section 5.4, but omit steps 6, 7, 8 and 9, since in this case these contribute nothing to the computation .

b) Let $\theta=0$ in the time-stepping method. Use your program to show that large values of $\Delta t$ lead to oscillations that grow with time, and that small values lead to a solution that goes to zero.

c) Let $\theta=1/2$. Use your program to show that large values of $\Delta t$ lead to slowly decaying oscillations.

d) Let $\theta=1$. Use your program to show that even large values of $\Delta t$ lead to smooth (i.e., non-oscillating) decay.

**N.B.** A fixed mesh can be used in all cases. Your results should be in the form of plots, where the horizontal axis is time and the vertical axis is the computed solution at some fixed node.

## Exercise 5.3

In this exercise the program from Exercise 5.2 will be used to study the convergence order of the time-stepping method.

We consider a fixed mesh and a fixed time value $t$. For the sequence of time steps

$$\Delta t = t/2^p\,, \quad p=0,1,\ldots$$

we define

$$\tilde{\mathbf{u}}_{\mathbf{p}}=\hat{\mathbf{u}}_{\mathbf{t/\Delta t}}$$

where the final time $t/\Delta t$ is fixed with the size of the time step determining the total number of steps.

All vectors $\tilde{\mathbf{u}}_\mathbf{p}$ are approximate solutions of the problem in Exercise 5.2 at the same time value $t$. Let

$$q_p = \log_2 \frac{\| \tilde{\mathbf{u}}_{\mathbf{p-2}} - \tilde{\mathbf{u}}_{\mathbf{p-1}} \|_\infty}{\| \tilde{\mathbf{u}}_{\mathbf{p-1}} - \tilde{\mathbf{u}}_{\mathbf{p}} \|_\infty}, \quad p = 2, 3, \ldots$$

and suppose that for some value $q$ we have $q_p \to q$, $p \to \infty$. Then it can be shown that

$$\| \hat{\boldsymbol{u}}_{t/\Delta t} - \hat{\mathbf{u}}(\mathbf{t}) \|_\infty = \mathcal{O}((\boldsymbol{\Delta t})^{\mathbf{q}})$$

$q$ is the order of convergence of the time-stepping method.

a) By computing values of $q_p$, verify the orders of convergence given at the end of Section 5.3 for the cases $\theta = 0, 1/2$ and $1$.

## Exercise 5.4

This exercise continues the two previous exercises.

To get an approximate solution that is close to the analytical solution, one must refine the mesh as well as the time step. For any uniform mesh we define an element size measure $h$ matching that of the edge lengths of the triangles. Let $t$ be a fixed time point, and consider the sequence of mesh parameters

$$(h,\, \Delta t) = \frac{1}{2^p}(1, t), \quad p = 1, 2, \ldots$$

For each value of $p$ let $\tilde{\mathbf{u}}_\mathbf{p} = \hat{\mathbf{u}}_{\mathbf{t}/\boldsymbol{\Delta t}}$, as before, and let $\boldsymbol{u}_p$ denote the analytical solution on the $p'th$ mesh at time $t$. Let

$$\tilde{q}_p = \log_2 \frac{\| \tilde{\mathbf{u}}_{\mathbf{p-1}} - \mathbf{u}_{\mathbf{p-1}} \|_\infty}{\| \tilde{\mathbf{u}}_{\mathbf{p}} - \mathbf{u}_{\mathbf{p}} \|_\infty}, \quad p = 2, 3, \ldots$$

and suppose that for some value $\tilde{q}$ we have $\tilde{q}_p \to \tilde{q}$, $p \to \infty$. Then it can be shown that

$$\| \hat{\mathbf{u}}_{\mathbf{t}/\boldsymbol{\Delta t}} - \mathbf{u}(\mathbf{t}) \|_\infty = \mathcal{O}((\boldsymbol{\Delta t})^{\tilde{\mathbf{q}}})$$

a) Use your program to determine $\tilde{q}$ when $\theta = 1/2$. (The value $\tilde{q}$ is the order of convergence of the *entire* numerical method).

## Exercise 5.5

a) Add Steps 6, 7, 8 and 9 to the program from Exercise 5.2 and use the new program to solve the problem

$$u_t = \nu \left( u_{xx} + u_{yy} + \pi^2 e^{-\pi^2 \nu t} \sin(\pi x) \sin(\pi y) \right), \quad (x, y) \in \Omega, \quad t > 0$$

$$u(x, y, 0) = \sin(\pi x) \sin(\pi y) + xy, \quad (x, y) \in \Omega$$

$$u(x, y, t) = xy, \quad (x, y) \in \Gamma, \quad t \geq 0$$

The analytical solution is

$$u(x, y, t) = e^{-\pi^2 \nu t} \sin(\pi x) \sin(\pi y) + xy$$

b) Repeat Exercise 5.4 to show that second-order convergence is preserved for $\theta = 1/2$.

## Exercise 5.6

a) Solve the problem

$$u_t = \nu\,(u_{xx} + u_{yy}) + \frac{\pi}{2} xy \cos\left(\frac{\pi}{2} t\right)\,, \qquad (x,y) \in \Omega\,, \quad t > 0$$

$$u(x,y,0) = \sin(\pi x)\sin(\pi y)\,, \qquad (x,y) \in \Omega$$

$$u(x,y,t) = xy \sin\left(\frac{\pi}{2} t\right)\,, \qquad (x,y) \in \Gamma\,, \quad t \geq 0$$

The analytical solution is

$$u(x,y,t) = e^{-2\pi^2 \nu t} \sin(\pi x)\sin(\pi y) + xy \sin\left(\frac{\pi}{2} t\right)$$

b) Show that second-order convergence is preserved for $\theta = 1/2$.

# Appendix A

# Projects

In this chapter several advanced projects are suggested for the last week of the course. The projects have been suggested by the persons accredited.

# A.1 Added Mass Coefficient

*By Per Grove Thomsen and Stefan Mayer*

## Introduction

A ship or barge or any other body that is accelerated through a fluid will experience a reactive force from the fluid. Part of that force is drag, which is a function of the relative velocity of the body relative to the fluid and gives rise to energy losses. The other part is a function of the acceleration and is due to the inertia of the surrounding liquid which is accelerated together with the body. This extra inertia is expressed by a coefficient, the **'added mass coefficient'**, that depends only on the body geometry, given certain assumptions, [41].

For simple body geometries it is possible to find a theoretical value of the added mass coefficient and for a square [41] gives the result:

$$m_{11} = 4.754...$$

Until recently no more digits were known in the result and We will attempt to find a better value using a numerical solution to a boundary value problem. The result has been confirmed in a later study [1] where the exact result is found. Some more background details can be found in [56].

## A boundary value problem

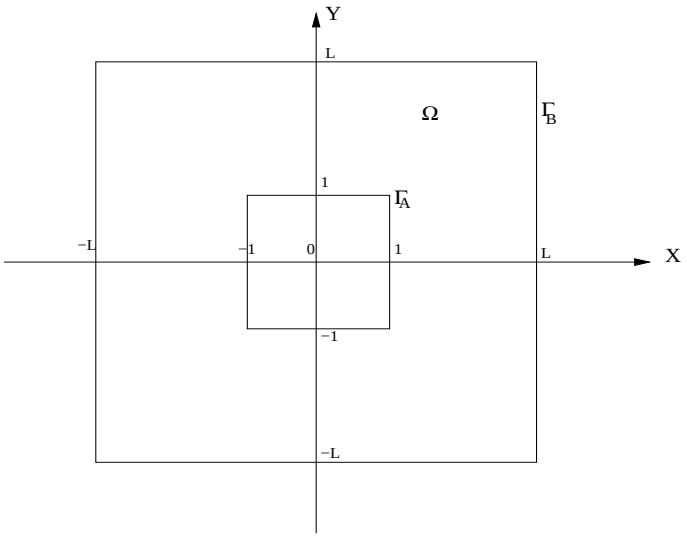



Figure A.1: The domain of the PDE

The added mass coefficient may be computed numerically by carrying out an experiment where the square is given a unit acceleration from left to right and the pressure in the resulting flow in the liquid is found by solving a boundary value problem as illustrated in figure 1 numerically. The outer boundary is an approximation to infinity, the inner boundary is the inner square, and the liquid is in the space between the two boundaries. In the liquid we must satisfy the Laplace

equation:

$$q_{xx} + q_{yy} = 0 \tag{A.1}$$

The inner boundary condition can be expressed:

$$q_n = -\underline{n} \cdot \underline{e}_1 = -n_1 \tag{A.2}$$

$q_n$ : The outward directed normal derivative
$\underline{n} = [n_1, n_2]$: the outward directed normal vector
$\underline{e}_1 : [1, 0]$

The outer boundary condition is:

$$q_n = 0 \tag{A.3}$$

or, alternatively,

$$q = 0 \tag{A.4}$$

The first part of this exercise consists of solving the partial differential equation with the Finite

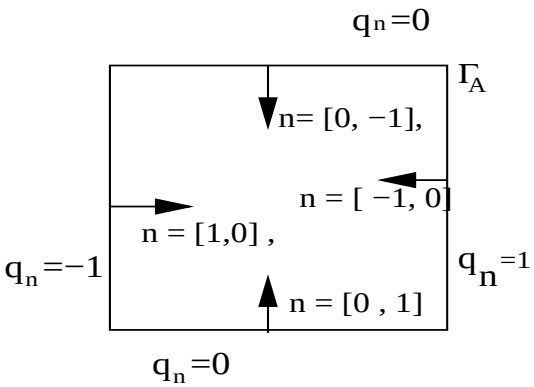



Figure A.2: Inner boundary, normals and conditions

Element method in the given domain.

**Computation of the added mass coefficient.** For the computation of the added mass coefficient we need to compute the following integral.

$$m_{11} = \int_{\Gamma_A} qn_1 ds = \int_{\text{right edge}} -qds + \int_{\text{left edge}} qds \tag{A.5}$$

Results will show that $m_{11}$ is a function of L, it may be assumed that the result will be better when L is increased, $L \to \infty$.

$$\text{Wanted} : \lim_{L\to\infty} m_{11}(L) \tag{A.6}$$

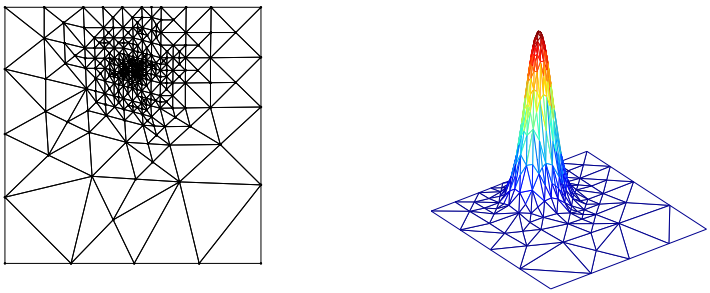
## Remarks

This exercise extends the material in the notes in the following respects:

- There is a "hole" in the domain.
- The problem is singular with the boundary condition $q_n = 0$ on $\Gamma_B$.
- An integral of the solution must be computed (by some quadrature formula).
- Regarding the FEM treatment of this problem, beware that the need for large L and small elements makes computations very expensive because of the largeness of the systems involved. This means that for an efficient computation we need a graded mesh, you have to choose a way to construct this mesh appropriately.

## A.2 Adaptive Mesh Generation

*By Allan P. Engsig-Karup*



### Introduction

A main motivation for solving a Boundary Value Problem (BVP) is to find a solution which is sufficiently accurate. When solving a BVP using a numerical method, e.g. FEM, we usually split the problem in two, namely, the problem of setting up a solver for the problem together with the mesh generation problem. The mesh problem is intimately connected with the need for sufficient accuracy and also has a direct impact on the CPU time because it relates to the degrees of freedom (size of problem) in a discrete formulation of the BVP. Thus, usually we want to distribute elements only as needed and since the exact solution is rarely known, this is difficult to do at the stage of generating an initial mesh. Thus, for the mesh generation problem it is desirable if we can both easily and automatically generate a given mesh suitable for our accuracy requirements. This has been and still is a very active research area and mesh refinement techniques are typically applied for problems, e.g. in problems involving multi-scale physics, to capture the features of solutions with steep gradients or discontinuities in the function or its gradients.

Thus, the goals of this project is to implement an algorithm which automatically can generate a suitable mesh to provide the basis for a solution within a given accuracy, and to make some simple performance comparisons and measurements. The project will naturally extend work done in Chapter 2 (and possibly Chapter 4) of the lecture notes and is based on the presentation in [24].

### The Model Problem

Let us consider the following model problem

$$u_{xx} + u_{yy} = f(x, y), \quad (x, y) \in \Omega([0, 1]^2) \tag{A.7}$$

with sufficient boundary conditions to make the problem well-posed. We assume an exact solution (illustrated on first page) is given as

$$u(x, y) = e^{-100((x-x_0)^2+(y-y_0)^2)} \tag{A.8}$$

with coordinates of the peak given as $(x_0, y_0) = (0.5, 0.75)$. The exact solution can be used for defining boundary conditions.

## The Mesh

For solving the BVP you will need a way to generate an unstructured initial mesh. You may manually define a mesh, employ your own routines or any favorite mesh generator. You need to be able to generate a mesh for the square domain $\Omega([0,1]^2)$ in terms of the mesh data tables `EToV`, `VX` and `VY`. Make sure that all elements are counter-clockwise oriented (cf. Appendix C in the lecture notes).

For adaptive mesh refinement, a basic algorithm can be stated as

- Create initial mesh
- Repeat while total error is larger than acceptable tolerance;
  - Solve the BVP on the mesh
  - Estimate the error in the solution
  - Mark elements for refinement according to some tolerance criterium
  - Locally refine the marked elements (and possibly their neighbors)

To solve the BVP you will need the solver routines that was created for the solution of the exercises in Chapter 2.

The three basic components of the local mesh-refinement algorithm are

- An algorithm for locally refining a mesh
- Element-by-element error estimator
- Marking strategy for choosing which triangles to refine

There are two basic methods for dividing triangles, namely, regular bisection where the triangle is split in four smaller sub-triangles, and bisection, where the triangle is split in two smaller triangles. A widely used algorithm for locally refining a mesh is the Newest-node bisection algorithm. This algorithm needs to be combined with local error estimation that can be used in a procedure for choosing (i.e marking) which elements should be refined.

### A newest-node bisection

The Newest-node bisection algorithm is relatively simple to implement and never creates any nonconforming triangles during the local refinements. The algorithm is based on an initial mesh where edges should be made compatibly divisible.

Some notation will come ind handy in the explanation of the algorithm. For the elements, we will need to define a peak and a base. The peak will be the vertex which is positioned opposite to the base edge of an element. When a triangle is marked for local refinement and thereafter bisected, we will need to create a new vertex and add it to the list of existing vertices. The newest node is always defined to be the peak of the new triangles that are created in the bisection.

You will need to create a Matlab routine with the following header

```
function EToV=trianglelabeling(VX,VY,EToV);
```

which prepares the initial mesh by determining the peak and the base of each element with the base being the longest edge of each element. As a part of the update the vertices of each element it can be necessary to cyclically reorder (to maintain a counter-clockwise ordering) the node positions to position the peak node of each element in the column `EToV(:,1)`.

To prepare the initial mesh and the starting point is a labeling algorithm which prepares the mesh for longest-edge bisection.

It is possible to deduce that there will be three cases for bisection of a triangular element $e_k$ and it's neighboring triangle element $e_j$.

- The base of triangle $e_k$ is a boundary edge.
- The neighboring triangle $e_j$ (across base edge in $e_k$) has the same base.
- The triangle $e_j$ on the other side of the base of $e_k$ has a different base.

In the first case, the triangle $e_k$ is split into two smaller triangles by bisection from peak to the middle point of the base.

An edge will be compatibly divisible if two neighboring triangles share an edge which is a base in both triangles. Therefore, in the second case the triangle $e_k$ and it's neighbor $e_j$ is split by bisection replacing the two triangles with four smaller ones. Such a division may result in the bisection of triangles not marked for refinement, but is necessary to maintain mesh conformity.

In the third case, one can show that if we recursively bisect the neighboring triangle $e_j$ before bisection of $e_k$ then one of the new sub-triangles of $e_j$ created will share a base with $e_k$ and we can proceed as described above.

**Algorithm: Newest-node bisection**

Given a triangle $e_k$ to bisect
**if** the base of $e_k$ is a boundary edge
    Replace $e_k$ by two triangles.
**else**
    Let $e_j$ be neighboring $e_k$ across base of $e_k$
    **if** the base of $e_k$ is also the base of $e_j$
        Replace each of $e_k$ and $e_j$ by two sub-triangles
    **else**
        Recursively call this algorithm to bisect $e_j$
        Recursively call this algorithm to bisect $e_k$

**End of algorithm**

In setting up the algorithm we will need the usual mesh data tables `EToV, VX` and `VY`. However, it is also useful to determine an Element-To-Element table `EToE`. This table stores for every element a row the numbers of elements that connects to the edge in question. Thus, `EToE(n,2)` will give the element number of the element which is adjacent to the second edge of element $n$. If the element number is the same as the element in question, it is a boundary edge. The mesh table can be determined by calling the supplied routine

```
>> [EToE]=tiConnect2D(EToV,Nfaces);
```

where `Nfaces` is the number of faces (i.e. edges) in an element for two-space dimensions.

Create a routine in Matlab with the following header

```
function [EToV,VX,VY] = NewestNodeBisection(EToV,VX,VY,k)
```

that makes it possible to bisect a marked element $e_k$ and make relevant updates to the mesh data tables.

Make sure to validate the routine you make by testing the algorithm on a small example mesh where the algorithm is tested on elements in the mesh that resembles the three cases for bisection mentioned.

**Error estimation**

To the local mesh-refinement algorithm we need an error estimator to guide the selection of elements for the local refinement to reduce the errors and to determine when to stop the refinement process. The main purpose of the adaptive mesh refinement is to make the solution procedure efficient. Therefore, for an error estimator to be practical is should effectively estimate the magnitude of the local errors in some appropriate norm. One possibility is to try and estimate the errors using the $L^\infty$-norm which for continuous functions are defined as

$$E = ||u - \hat{u}||_\infty = \max_{(x,y)\in\Omega} |u(x,y) - \hat{u}(x,y)| \tag{A.9}$$

This norm is straightforward to compute and can be used in a first test of the bisection algorithm where the solution is interpolated at the mesh nodes. Then, compute the average of the solution at the mesh nodes to get an approximation at the element centers and compare this to the exact solution for the local error estimation. This is a cheap way to test the algorithm as we do not need to solve the BVP every iteration of the refinement. Another possibility is to use the energy norm

$$||u - \hat{u}||_E = \sqrt{\int\int_\Omega (\nabla(u - \hat{u}) \cdot \nabla(u - \hat{u}) dxdy} \tag{A.10}$$

In practice, we seldom know the exact solution and therefore we need to come up with a way to estimate the errors in terms of the computed solution. There is a number of such estimators, e.g. based on differences between coarse and fine grid solutions, smoothness of solution, etc. However, we will cheat a little bit in this project, since we only solve model problems with a known solution. Thus, we can use the exact solution in our error estimator and use this for comparison with the computed solutions.

A simple (yet expensive) way of computing the Energy norm, is to make use of the element matrix $\mathcal{K}^{(n)}$ for each element with the matrix elements defined from

$$k_{r,s}^{(n)} = \int\int_{e_i} [(N_r^n)_x (N_s^n)_x + (N_r^n)_y (N_s^n)_y] dxdy \tag{A.11}$$

and use it for an explicit quadrature integration of the local error vector $e = u - \hat{u}^{(n)}$ as

```
% Energy norm of local error for n'th element
>> localerror = sqrt(e(:)'*Kn*e(:));
```

Can you verify that this expression is correct?

**Marking elements for refinement**

An essential part of a local mesh-refinement algorithm is a method for selecting which triangles to refine. The refinement should be based on some assessment of the magnitude of the local errors in some appropriate norm.

Having estimated the local errors, a very simple approach is to then select (mark) those elements $e_i$ which fail to meet an acceptable error tolerance `tol`, pick element $e_i$ which fulfills the condition

$$E_i > \texttt{tol} \tag{A.12}$$

This follows the idea, that the optimal mesh for a given problem has a uniform distribution of the errors. Also, it suggest that we should always start out with a coarse initial mesh, in order not to have regions in the mesh which are too accurate according to our accuracy condition.

## Results

You should produce a small report that covers the work you have done. This includes

- Derive the weak formulation for the model problem.
- Setup and solve the model problem using your Matlab toolbox and validate the code setup by making a convergence test to verify that you get the expected rate of convergence.
- Describe what steps has been taken to make the Matlab routines efficient and make a small performance study of solving the BVP with various parameters of your refinement algorithm (e.g. marking criteria, number of iterations, etc.). What is the minimum number of elements that can be used for an absolute tolerance of `tol=1e-2`?[1]
- Produce convergence plots of the errors vs. the degrees of freedom to assess the performance of the adaptive mesh refinement algorithm vs. a uniform refinement of the domain for the model BVP.
- If time permits, you may test the algorithm on a FEM-based solution strategy. Also, you may consult literature on the subject and try and make improvements to the suggested basic algorithms (consult teacher for relevant pieces of literature).

Make sure to discuss and comment on all results produced.

[1] Teacher's solution: 510 elements (energy norm) and 844 elements (max norm at nodes).

## A.3 Multigrid Algorithm

*By Anton Evgrafov*

### Introduction

A multigrid algorithm for solving partial differential equations is an extremely efficient iterative process constructing a solution through a sequence of inexpensive operations on a series of succesively refining meshes (grids). The algorithm is optimal both theoretically and, even more importantly, in practice for a wide class of problems in the following sense: the total cost of solving the problem depends linearly on the number of nodes in the mesh.

In this assignment you will implement a multigrid algorithm and study its performance on a model eliptic problem you have encountered during the week one of this course.

### The model problem

Let us consider the following model problem: find a twice differentiable function $u$ satisfying

$$\begin{aligned} -[\kappa(x)u'(x)]' &= f(x), \qquad \text{for all } x \in ]a,b[, \\ u(a) &= c, \\ u(b) &= d, \end{aligned} \tag{A.13}$$

where $\kappa$ is continuously differentiable on $]a,b[$ and $f$ is continuous on $]a,b[$. In addition, we assume that there are two positive constants $\kappa_L, \kappa_U$ such that $0 < \kappa_L \leq \kappa(x) \leq \kappa_U < +\infty$ for every $x \in ]a,b[$.

Note: since $\kappa(x)$ is not constant, you may need to resort to numerical integration when computing the stiffness matrix for this system, see Appendix B in the lecture notes.

### Coarsening/interpolation

Let us consider two FEM discretizations of the model problem (A.18) using the approach described in Chapter 1 of the lecture on two different meshes, say with $M_1$ and $M_2$ nodes chosen such that $M_1 - 1 \approx 2(M_2 - 1)$, that is, mesh two is roughly half as fine as the mesh one. We will refer to the global basis functions on the first mesh as $N_{1,i}$, $i = 1, \ldots, M_1$, and on the second mesh as $N_{2,i}$, $i = 1, \ldots, M_2$. Let $\hat{u}^1 = A_1^{-1} b_1$ and $\hat{u}^2 = A_2^{-1} b_2$ be the corresponding finite element solutions. Then we can write the following equality

$$\sum_{i_1=1}^{M_1} \hat{u}^1_{i_1} N_{1,i_1}(x) \approx u_{\text{exact}}(x) \approx \sum_{i_2=1}^{M_2} \hat{u}^2_{i_2} N_{2,i_2}(x), \tag{A.14}$$

where $u_{\text{exact}}(x)$ is the exact solution to the problem (A.18), and the precise mathematical meaning of "$\approx$" in (A.3) is obtained from error bounds for finite element methods, such as for example the ones found in Section 1.7 of the lecture notes.

We will utilize the two-sided approximation equality for two "dual" purposes: (1) for computing a function on a fine grid (grid one in this case) defined through its expansion coefficients $\hat{u}^1$ when its coarse-grid representation $\hat{u}^2$ is known (interpolation); (2) vice versa, for computing $\hat{u}^2$ if we know $\hat{u}^1$ (coarsening). Namely, since $\hat{u}^1_i = \sum_{i_1=1}^{M_1} \hat{u}^1_{i_1} N_{1,i_1}(x^1_i) \approx u(x^1_i)$, where $x^1_i$ is the coordinate

of the node $i$ of mesh one, we can approximate its value with $\hat{u}_i^1 \approx \sum_{i_2=1}^{M_2} \hat{u}_{i_2}^2 N_{2,i_2}(x_i^1)$. If piecewise linear shape functions are used, one can utilize Matlab's function `interp1` to perform the interpolation:

```
uhat1 = interp1(VX2,uhat2,VX1);
```

where `VX1` and `VX2` are the coordinates of vertices in meshes one and two, respectively. Similarly, one can do coarsening as $\hat{u}_i^2 \approx \sum_{i_1=1}^{M_1} \hat{u}_{i_1}^1 N_{1,i_1}(x_i^2)$, or in case of linear shape functions in Matlab:

```
uhat2 = interp1(VX1,uhat1,VX2);
```

For notational convenience, we will refer to the interpolation or coarsening operators as $I_{M_2}^{M_1} : \mathbb{R}^{M_2} \to \mathbb{R}^{M_1}$ and $I_{M_1}^{M_2} : \mathbb{R}^{M_1} \to \mathbb{R}^{M_2}$.

## Interpolation errors and smoothing

Let $\hat{u}^2$ be the FEM solution on the coarse grid with $M_2$ points, and let $u^1 := I_{M_2}^{M_1}$ be its interpolation onto the fine grid with $M_1$ points. Let us define the interpolation error $e^1 := u^1 - \hat{u}^1$, where $\hat{u}^1 = (A^1)^{-1} b^1$ it the FEM solution on a fine grid. It turns out that $e$ will be highly oscilatory, which may be easily alleviated with a *smoothing* operator. One has more than one option here; we will perform a few relaxed Jacobi iterations to reduce the oscilatory part of the error introduced by the interpolation operator. One relaxed Jacobi iteration updates $u^1$ as follows:

$$u^1 := u^1 + \omega[\mathrm{diag} A^1]^{-1}(b^1 - A^1 u^1), \tag{A.15}$$

where $0 < \omega < 1$ is the relaxation parameter, and $\mathrm{diag} A^1$ is the diagonal matrix having the same diagonal as $A^1$.

## Pre-smoothing and error equation

If the simple smoothing process, such as relaxed Jacobi iteration (A.15), can significantly reduce the "high-frequency" components of the error, we can carefully choose the right hand side of our linear system so that the solution to this new system has no highly oscilatory components. Namely, let $\hat{u}^1 = (A^1)^{-1} b^1$ be the FEM solution on the fine grid, and let $u^1$ be a result of several iterations of a (pre-)smoothing process applied to some initial guess. Then the error $e^1 := u^1 - \hat{u}^1$ should have no highly oscilating components remaining, and therefore we should be able to represent it very well on a coarse grid. Let us derive the equation for $e^1$:

$$A^1 e^1 = A^1(u^1 - \hat{u}^1) = A^1 u^1 - b^1 =: -r^1, \tag{A.16}$$

where $r^1$ is the residual vector corresponding to $u^1$. A good approximation $e^2$ to $e^1$ on a coare grid should solve the equation

$$A^2 e^2 = -r^2, \tag{A.17}$$

where $r^2$ is the representation of $r^1$ on a coarse grid. We would like to use the coarsening operator $I_{M_1}^{M_2}$ to compute $r^2$ from $r^1$; however, there is a small technical issue here. Namely, the residual should scale with the mesh size in exactly the same way as the right hand side vector does. One can check by substituting, e.g., a constant function into the expression for the right hand side that $b^1$ on a fine mesh is related to $b^2$ on a coarse mesh as approximately $b^2 \approx (h_2/h_1) I_{M_1}^{M_2} b^1$, where $h_2 = (b-a)/(M_2 - 1)$ and $h_1 = (b-a)/(M_1 - 1)$ are the element sizes on the coarse and the fine grids.

Summarizing, $e^2$ should approximately solve $A^2 e^2 = -(h_2/h_1) I_{M_1}^{M_2} r^1$, from which we compute $e^1 \approx I_{M^2}^{M_1} e^2$, and eventually the new approximation $u^1$ to $\hat{u}^1$ as $u^1 := u^1 - e^1$. Finally, as the interpolation operator $I_{M^2}^{M_1}$ introduces oscilatory error components into the solution, as we discussed in the previous section, we apply several (post-)smoothing iterations to $u^1$.

### Two-grid algorithm

We can now fully describe a two-grid iterative algorithm for solving a linear system $A^1u = b^1$. We start the iteration with $u^1$ being a suitable initial guess for $\hat{u}^1$.

- Pre-smooth: apply a few iterations of Jacobi smoothing (A.15) to $u^1$.
- Residual: $r^1 := b^1 - A^1u^1$.
- Coarsen: $r^2 := (h_2/h_1)I_{M_1}^{M_2}r^1$.
- Solve: solve the system $A^2\delta u^2 = r^2$.
- Interpolate: $\delta u^1 := I_{M_2}^{M_1}\delta u^2$.
- Error-correct: $u^1 := u^1 + \delta u^1$.
- Post-smooth: apply a few iterations of Jacobi smoothing (A.15) to $u^1$.

The iterative process $(i)$–$(vii)$ is stopped when the norm of the residual $r^1$ is sufficiently small.

### Test case

We will test the two-grid (and later multigrid) algorithm on an instance of the problem (A.18) with the data $a = 0.0$, $b = 1.0$, $\kappa(x) = 1.0+x$, $u_{\text{exact}}(x) = \cos(2\pi x)\exp(-x)$. The remaining parameters can be determined from the given data: $c = u_{\text{exact}}(a)$, $d = u_{\text{exact}}(b)$, $f(x) = -[\kappa(x)u'_{\text{exact}}(x)]'$.

- Implement the routine assembling the finite element matrix $A$ and another one for assembling the right-hand side $b$ for the model boundary value problem (A.18) on a uniform mesh with $M$ points, $M$ being a parameter.
- Given $M_1$ and $M_2$ as described in Section A.3, compute $\hat{u}^1$ and $\hat{u}^2$ using for example a direct solver. Compute and plot the interpolation error $I_{M_2}^{M_1}\hat{u}^2 - \hat{u}^1$.
- Implement a relaxed Jacobi smoothing routine, taking the number of smoothing iterations as well as the relaxation $\omega$ as input parameters. Apply 3 iterations of smoothing with $\omega = 2/3$ to $u^1 = I_{M_2}^{M_1}\hat{u}^2$. Plot the "smoothed" error, or rather error between the smoothed $u^1$ and $\hat{u}^1$. Compare with the previous step.
- Implement an iterative two-grid algorithm described in Section A.3. Use a direct solver when computing $\delta u^2$. Start with the initial guess $u^1$ being a zero vector, and establish a dependece between the number of iterations needed to achieve a given relative accuracy $\|r^1\|/\|b^1\| < \varepsilon$ and a mesh size $h_1 = (b-a)/(M_1-1)$, while keeping $h_2 \approx 2h_1$ on a sequence of successively refined meshes.
- Finally, extend the two-grid algorithm to a multigrid algorithm. Basically, recursively apply the idea of a two-grid algorithm towards solving a linear system $A^2\delta u^2 = r^2$ in step $(iv)$ of the two-grid algorithm. In order to stop the recursion, use a direct solver whenever the size of the system is too small, say, no more that 10 degrees of freedom.

  As in the previous step, study the number of iterations needed to achieve a given accuracy as a function of the mesh size.

When you are implementing the multigrid algorithm, think about how to implement it efficiently. Namely, unnecessary re-assembly of the matrices $A^i$ should be avoided; smoothing steps should be implemented in an efficient way, etc.

## Possible generalizations

Here is a couple suggestions, in no particular order:

- Try to apply a multigrid solver to a FEM discretization of (A.18) with higher order methods (see Chapter 3 in the lecture notes).
- Apply the multigrid solver to a model 2D problem, such as the steady state heat conduction problem (Chapter 2 in the lecture notes).

## Literature notes

Both [50] and [12] contain chapters dedicated to multigird methods. There is also an online resource https://computation.llnl.gov/casc/people/henson/mgtut/welcome.html dedicated to multigrid, which contains some introductory slides.

## Numerical integration

During the week 1 of the course, you have transformed an integral over $]x_e, x_{e+1}[$ into an integral over $]-1, 1[$:

$$\int_{x_e}^{x_{e+1}} f(x)\,dx = \frac{h_e}{2}\int_{-1}^{1} f(x_{e+1/2} + yh_e/2)\,dy,$$

where $h_e = x_{e+1} - x_e$, and $x_{e+1/2} = (x_e + x_{e+1})/2$.

Further, an integral over $]-1, 1[$ can be numerically approximated using Gaussian quadratures:

$$\int_{-1}^{1} f(y)\,dy \approx \sum_{i=1}^{Q} f(q_i)w_i,$$

where $Q$ is the number of quadrature points, and $q_i$, $w_i$ are respectively coordinates and the weights associated with quadrature points. Gaussian quadratures are "optimal" in the sence that they are exact for polynomials up the the order $2Q-1$. A Matlab function producing Gaussian quadrature points and weights up to the order 4 is listed below.

**Question 1**
*Verify that 1-point Gaussian quadrature is exact for every linear function $f(x) = ax + b$.*

```
function [q,w] = qgauss(n)
switch(n)
 case 1
  q=0;
  w=2;
 case 2
  q=[-1/sqrt(3); 1/sqrt(3)];
  w=[1; 1];
 case 3
  q=[-sqrt(3/5); 0; +sqrt(3/5)];
  w=[5/9; 8/9; 5/9];
 case 4
  q=[-sqrt((3+2*sqrt(6/5))/7); ...
     -sqrt((3-2*sqrt(6/5))/7); ...
      sqrt((3-2*sqrt(6/5))/7); ...
      sqrt((3+2*sqrt(6/5))/7)];
  w=[(18-sqrt(30))/36; (18+sqrt(30))/36; ...
```

```
     (18+sqrt(30))/36; (18-sqrt(30))/36];
 otherwise
  error('Unknown quadrature order');
end
```

## A.4 Dirichlet-Neumann Domain Decomposition Algorithm

*By Anton Evgrafov*

### Introduction

Domain decomposition generally speaking refers to any method for solving a partial differential equation by transforming it into a system of coupled equations on smaller subdomains. The algorithm is particularly well suited for performing high-fidelity finite element simulations on parallel computers, see for example Figure A.3, when all subdomain problems may be solved independently and simultaneously, and only interface/transmission conditions require collaboration between different parallel processes.

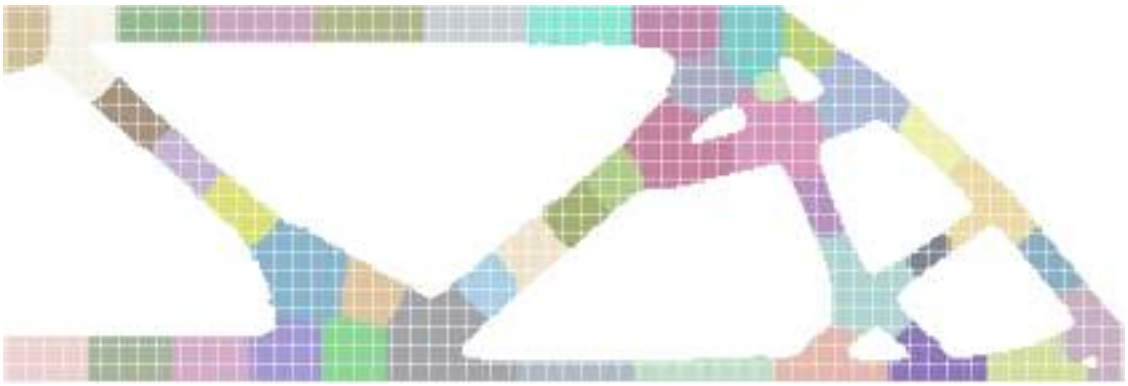


Figure A.3: Domain decomposition (mesh decoposition) of a Messerschmitt–Bölkow–Blohm (MBB) aircraft floor support beam. Different subdomains have different colors.

Instead of having a generic discussion about domain decomposition approaches, let us make the ideas specific by considering a simple domain decomposition algorithm for a model problem.

### The model problem

Let us consider the following model problem: find a twice differentiable function $u$ satisfying

$$\begin{aligned} -[\kappa(x)u'(x)]' &= f(x), \qquad \text{for all } x \in ]a,b[, \\ u(a) &= c, \\ u(b) &= d, \end{aligned} \tag{A.18}$$

where $\kappa$ is continuously differentiable on $]a,b[$ and $f$ is continuous on $]a,b[$. In addition, we assume that there are two positive constants $\kappa_L, \kappa_U$ such that $0 < \kappa_L \leq \kappa(x) \leq \kappa_U < +\infty$ for every $x \in ]a,b[$.

### The domain decomposition

For simplicity, let us decompose the domain $\Omega = ]a,b[$ into only two subdomains $\Omega_1 = ]a_1,b_1[$ and $\Omega_2 = ]a_2,b_2[$, where $a = a_1 < b_1 = a_2 < b_2 = b$. Then the problem (A.18) is *equivalent* to the

following coupled problem:

$$\begin{aligned} -[\kappa(x)u_1'(x)]' = f(x), \quad x \in ]a_1, b_1[, \qquad -[\kappa(x)u_2'(x)]' = f(x), \quad x \in ]a_2, b_2[, \\ u_1(a_1) = c, \qquad\qquad u_2(b_2) = d, \\ u_1(b_1) = u_2(a_2), \\ \kappa(b_1)u_1'(b_1) = \kappa(a_2)u_2'(a_2), \end{aligned} \tag{A.19}$$

in the sense that if $u$ is the unique solution of (A.18) and $(u_1, u_2)$ is the unique solution of (A.19) then $u|_{]a_1,b_1[} = u_1$, and $u|_{]a_2,b_2[} = u_2$. The two conditions on the interface between the subdomains, in our case, the two last equalities in (A.19), are known as the *transmission conditions*.

If we knew the value $u(b_1) = u(a_1)$ *or* the value $u'(b_1) = u'(a_1)$, where $u$ is the solution to (A.18), then we could independently solve two subdomain problems in (A.19) with either Dirichlet or Neumann (or any linear combination thereof) boundary condition on the subdomain boundary. The satisfaction of the transmission condition would then have followed from the equivalence between (A.18) and (A.19) and the uniqueness of solutions. In any practical situation we of course do not know any of these values, and most domain decomposition algorithms try to satisfy the transmission conditions iteratively, where at each iteration one or several subdomain problems are to be solved. In the next section we describe perhaps the simplest of such algorithms.

## The Dirichlet–Neumann algorithm

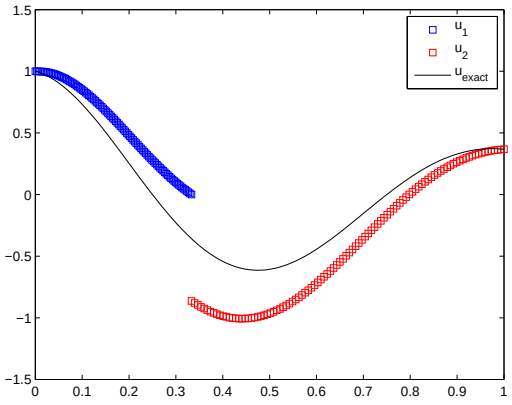



Figure A.4: The finite element solution to the model problem after one iteration of the Dirichlet–Neumann algorithm.

With two subdomains, the Dirichlet–Neumann algorithm can be described as follows. At the step $n$ of the algorithm, we have an approximation $u_\Gamma^n$ of the value of $u(b_1) = u(a_1)$. We then solve the boundary value problem on sub-domain 1 with a Dirichlet boundary condition on the sub-domain boundary:

$$\begin{aligned} -[\kappa(x)(u_1^{n+1/2})'(x)]' = f(x), \qquad \text{for all } x \in ]a_1, b_1[, \\ u_1^{n+1/2}(a_1) = c, \\ u_1^{n+1/2}(b_1) = u_\Gamma^n, \end{aligned} \tag{A.20}$$

from which we obtain the value $\kappa(b_1)(u_1^{n+1/2})'(b_1)$. We then substitute this value into the problem on the second subdomain with a Neumann boundary condition on the subdomain boundary

(whence the name of the algorithm):

$$\begin{aligned} -[\kappa(x)(u_2^{n+1})'(x)]' &= f(x), \qquad \text{for all } x \in ]a_2, b_2[, \\ \kappa(a_2)(u_2^{n+1})'(a_2) &= \kappa(b_1)(u_1^{n+1/2})'(b_1), \\ u_2^{n+1}(b_2) &= d, \end{aligned} \tag{A.21}$$

Finally, we update the value $u_\Gamma^n$ as

$$u_\Gamma^{n+1} = \theta u_2^{n+1}(a_2) + (1-\theta)u_\Gamma^n, \tag{A.22}$$

where $\theta > 0$ is a chosen relaxation parameter. Of course, the iterations stop when a desired accuracy is achieved, that is, when $|u_\Gamma^{n+1} - u_\Gamma^n| < \varepsilon$, where $\varepsilon > 0$ is the prescribed accuracy. Figure A.4 shows a configuration obtained after one iteration of the algorithm applied to an instance of the model problem (A.18) (see Section A.9).

## Implementation

Start by implementing an algorithm for solving the subdomain problems (A.20) and (A.21) given the boundary data, following the guidelines given in Chapter 1 of the lecture notes. Note that you will have to repeatedly solve these problems with the same matrices and differing right hand sides, and therefore you may try to take advantage of this by pre-assembling the matrices and possibly pre-factorizing them. Also note that the second problem has Neumann boundary conditions on one part of the boundary, and therefore you will need to modify your assembly routine as described in Section 2.9 of the lecture notes.

Note, that when $\kappa(x)$ is not constant over the finite element, you may need to utilize numerical quadratures to compute elemental stiffness matrices and vectors:

$$\begin{aligned} k_{ij}^{(e)} &= \int_e \kappa(x)(N_i^{(e)})'(x)(N_j^{(e)})'(x)\,dx, \\ f_i^{(e)} &= \int_e f(x)N_i^{(e)}(x)\,dx, \end{aligned}$$

see Appendix B in the lecture notes.

In order to evaluate the value $\kappa(b_1)(u_1^{n+1/2})'(b_1)$ the following approach can be taken. Let $N_{b_1}$ be the shape function associated with the node $b_1$ in the mesh, and $u_1$ be the exact solution of the subdomain problem (A.20). Then, we have the following equality by virtue of the fundamental theorem of calculus:

$$\begin{aligned} \int_{a_1}^{b_1} [(\kappa(x)u_1'(x))'N_{b_1}(x) + \kappa(x)u_1'(x)N_{b_1}'(x)]\,dx &= \kappa(b_1)u_1'(b_1)N_{b_1}(b_1) - \kappa(a_1)u_1'(a_1)N_{b_1}(a_1) \\ &= \kappa(b_1)u_1'(b_1), \end{aligned} \tag{A.23}$$

where the last equality is due to the fact that $N_{b_1}(b_1) = 1$ and $N_{b_1}(a_1) = 0$. On the other hand, we know that $-(\kappa(x)u_1'(x))' = f(x)$. Therefore, we arrive at the following expression for $\kappa(b_1)u_1'(b_1)$:

$$\kappa(b_1)u_1'(b_1) = \int_{a_1}^{b_1} \kappa(x)u_1'(x)N_{b_1}'(x)\,dx - \int_{a_1}^{b_1} N_{b_1}(x)f(x)\,dx. \tag{A.24}$$

It only remains to substitute a finite element approximation of $u_1^{n+1/2}$ into (A.24) to get an approximation of $\kappa(b_1)(u_1^{n+1/2})'(b_1)$.

## Test case

Apply the Dirichlet–Neumann domain decomposition algorithm to the solution of the model problem (A.18) with the data $a = 0.0$, $b = 1.0$, $\kappa(x) = 1.0 + x$, $u_{\text{exact}}(x) = \cos(2\pi x)\exp(-x)$. Set $b_1 = a_2 = 1/3$, and solve the problem on a sequence of refining meshes with $M_1 - 1 = 2(M_2 - 1)$ (where $M_i$ is the number of nodes in the discretization of $]a_i, b_i[$) to a fixed accuracy $\varepsilon = 10^{-10}$ with a relaxation parameter $\theta = 1/2$. The remaining parameters can be determined from the given data: $c = u_{\text{exact}}(a)$, $d = u_{\text{exact}}(b)$, $f(x) = -[\kappa(x)u'_{\text{exact}}(x)]'$.

Verify your implementation against the following error estimate:

$$\|u_i(x) - u_{\text{exact}}(x)\|_{L^2(a_i,b_i)} = \left( \int_{a_i}^{b_i} [u_i(x) - u_{\text{exact}}(x)]^2 \, dx \right)^{1/2} \leq Ch^2, \tag{A.25}$$

where $u_{\text{exact}}$ is the exact solution to the model problem (A.18), and $u_i$ is a solution to the finite element problem on the subdomain $i$. In (A.25), the constant $C$ is independent from the mesh size $h = \max_{1 \leq e \leq M-1} h_e$, but will generally speaking depend on the smoothness of the solution $u_{\text{exact}}$, and the behaviour of $\kappa$. If $u_{\text{exact}}$ is known, the integral in (A.25) can be evaluated using numerical integration over elements:

$$\|u(x) - u_{\text{exact}}(x)\|^2_{L^2(a_i,b_i)} = \sum_{e=1}^{M_i-1} \int_{x_e}^{x_{e+1}} [u_{\text{exact}}(x) - \hat{u}_1^{(e)} N_1^{(e)}(x) - \hat{u}_2^{(e)} N_2^{(e)}(x)]^2 \, dx. \tag{A.26}$$

Also, verify the optimality of the Dirichlet–Neumann algorithm in the sense that the number of iterations needed to achieve a given accuracy for a given problem does not increase with the decreasing mesh size, and in many cases does not depend on the mesh size.

## Possible generalizations

Here is a couple suggestions, in no particular order:

1. Try to solve the subdomain problems with higher order methods (see Chapter 3 in the lecture notes) in order to improve the error estimate (A.25).
2. Apply the same idea of decomposing the domain into two parts to a model 2D problem, such as the steady state heat conduction problem (Chapter 2 in the lecture notes).

It is possible to generalize the Dirichlet–Neumann algorithm to more than two subdomains, though in practice other flavours of domain decomposition algorithms are employed for this purpose.

## Literature notes

A comprehensive treatment of all aspects of domain decomposition algorithms is found in [57], but other excellent books such as [12] or [50] contain chapters dedicated to domain decomposition algorithms as well.

## Numerical integration

During the week 1 of the course, you have transformed an integral over $]x_e, x_{e+1}[$ into an integral over $]-1, 1[$:

$$\int_{x_e}^{x_{e+1}} f(x) \, dx = \frac{h_e}{2} \int_{-1}^{1} f(x_{e+1/2} + y h_e/2) \, dy,$$

where $h_e = x_{e+1} - x_e$, and $x_{e+1/2} = (x_e + x_{e+1})/2$.

Further, an integral over $]-1,1[$ can be numerically approximated using Gaussian quadratures:

$$\int_{-1}^{1} f(y)\,dy \approx \sum_{i=1}^{Q} f(q_i)w_i,$$

where $Q$ is the number of quadrature points, and $q_i$, $w_i$ are respectively coordinates and the weights associated with quadrature points. Gaussian quadratures are "optimal" in the sence that they are exact for polynomials up the the order $2Q-1$. A Matlab function producing Gaussian quadrature points and weights up to the order 4 is listed below.

**Question 2**
*Verify that 1-point Gaussian quadrature is exact for every linear function $f(x) = ax + b$.*

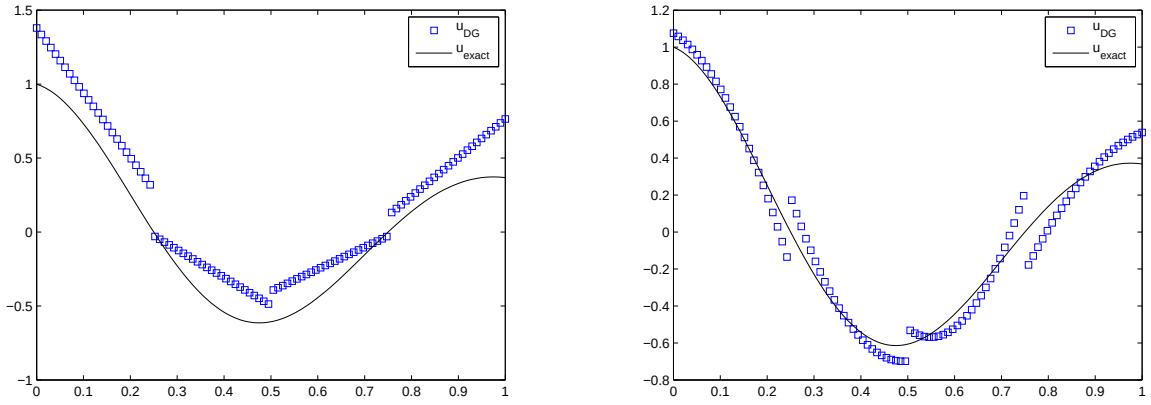
```
function [q,w] = qgauss(n)
switch(n)
 case 1
  q=0;
  w=2;
 case 2
  q=[-1/sqrt(3); 1/sqrt(3)];
  w=[1; 1];
 case 3
  q=[-sqrt(3/5); 0; +sqrt(3/5)];
  w=[5/9; 8/9; 5/9];
 case 4
  q=[-sqrt((3+2*sqrt(6/5))/7); ...
     -sqrt((3-2*sqrt(6/5))/7); ...
      sqrt((3-2*sqrt(6/5))/7); ...
      sqrt((3+2*sqrt(6/5))/7)];
  w=[(18-sqrt(30))/36; (18+sqrt(30))/36; ...
     (18+sqrt(30))/36; (18-sqrt(30))/36];
 otherwise
  error('Unknown quadrature order');
end
```

## A.5 Discontinuous Galerkin Methods for Elliptic Problems

*By Anton Evgrafov*

### Introduction

Discontinus Galerkin (DG) methods constitute a modern subclass of finite element methods, which are characterized by the fact that the solution is sought in a space of *discontinuous* piece-wise polynomial functions (cf. Figure A.5), whence the name. The methods originate from the need to precisely approximate discontinuous solutions to hyperbolic partial differential equations (conservation laws), but have recently gained popularity for numerically solving elliptic and parabolic problems as well. The reason for re-birth of discontinuous Galerkin methods is that they enjoy several very desirable from computational point of view properties. For example, they make it easy to implement *hp*-adaptive algorithms, perform simulations on meshes consisting of several non-conforming pieces, and are well suited for parallel computing which becomes a commodity in coming years. Of course, these methods also have their own drawbacks compared to "standard" continuous Galerkin methods, most notably the increase of the number of degrees of freedom needed to represent the solution, especially in the case low order polynomials are used.




(a) (b)

Figure A.5: Example of (a) piece-wise linear, and (b) piece-wise quadratic discontinuous approximation of a smooth function.

In this assignment you will have an opportunity to get a hands-on experience with discontinuous Galerkin discretization of the model elliptic problem you have encountered (and solved using the continuous Galerkin approach) during week one of this course.

### The model problem

Let us consider the following model problem: find a twice differentiable function $u$ satisfying

$$\begin{aligned} -[\kappa(x)u'(x)]' &= f(x), \qquad \text{for all } x \in ]0, L[, \\ u(0) &= c, \\ u(L) &= d, \end{aligned} \tag{A.27}$$

where $\kappa$ is continuously differentiable on $]0, L[$ and $f$ is continuous on $]0, L[$. In addition, we assume that there are two positive constants $\kappa_L, \kappa_U$ such that $0 < \kappa_L \leq \kappa(x) \leq \kappa_U < +\infty$ for every $x \in ]0, L[$.

## The mesh

The domain $]0, L[$ is discretized into $M-1$ elements $e_i = ]x_i, x_{i+1}[$ of size $h_i = (x_{i+1} - x_i) > 0$, $i = 1, \ldots, M-1$, where $0 = x_1 < x_2 < \cdots < x_{M-1} < x_M = L$.

## The weak form

Within the framework of discontinuous Galerkin methods we deal with functions, which are smooth on every element but may be discontinuous across inter-elemental boundaries. Therefore we cannot perform integration by parts across the whole domain, $]0, L[$ in the present case, as we normally do within "standard" continuous Galerking framework. Instead, we multiply the differential equaiton with a piece-wise smooth test function and carry over integration by parts over *each element*. We then add all contributions together, and introduce several additional terms for symmetrizing the weak form and also for penalizing the discontinous solutions. These calculations, whereas not difficult, are quite tedious and therefore we chose to present the final resulting weak form and refer the interested reader to for example [48].

For a piecewise-continuous function with jumps across elemental boundaries, say $w$, it will be convenient to define the jump

$$[\![w(x_i)]\!] := w(x_i^-) - w(x_i^+), \tag{A.28}$$

and the average

$$\{w(x_i)\} := 0.5[w(x_i^-) + w(x_i^+)], \tag{A.29}$$

operators, where $x_i$, $i = 2, \ldots, M-1$ is an arbitrary interior mesh node, and

$$w(x_i^+) := \lim_{\varepsilon \downarrow 0} w(x_i + \varepsilon), \qquad w(x_i^-) := \lim_{\varepsilon \downarrow 0} w(x_i - \varepsilon),$$

are respectively right and left limits of this function at $x_i$. At the boundary nodes, $x_1 = 0$ and $x_M = L$, we define average and jump simply as

$$\begin{aligned} [\![w(x_1)]\!] &:= -w(x_1^+), & [\![w(x_M)]\!] &:= w(x_M^-), \\ \{w(x_1)\} &:= w(x_1^+), & \{w(x_M)\} &:= w(x_M^-), \end{aligned} \tag{A.30}$$

Let us now state a weak form associated with a model equation (A.27) and a chosen mesh on $]0, L[$:

$$\begin{aligned} &\sum_{e=1}^{M-1} \int_{x_e}^{x_{e+1}} \kappa(x) u'(x) v'(x)\, dx \\ &+ \sum_{n=1}^{M} \left( - \{\kappa(x_n) u'(x_n)\} [\![v(x_n)]\!] - \{\kappa(x_n) v'(x_n)\} [\![u(x_n)]\!] + \frac{\delta \kappa(x_n)}{\min(h_n, h_{n+1})} [\![u(x_n)]\!] [\![v(x_n)]\!] \right) \\ &= \int_0^L f(x) v(x)\, dx \\ &+ \{\kappa(x_1) v'(x_1)\} c - \frac{\delta \kappa(x_1)}{h_1} [\![v(x_1)]\!] c \\ &- \{\kappa(x_M) v'(x_M)\} d + \frac{\delta \kappa(x_M)}{h_{M-1}} [\![v(x_M)]\!] d, \end{aligned} \tag{A.31}$$

where $\delta > 0$ is a parameter to be chosen later. Note that the first sum goes over all *elements* in the mesh, whereas the second sum goes over all elemental boundaries (nodes) in the mesh, whence the summation indices $e$ and $n$.

**Question 3**
*If the terms in the weak form, containing both $u$ and $v$, are symmetric with respect to changing the roles of $u$ and $v$, then the resulting finite element matrices will be symmetric. Is the form (A.31) symmetric?*

**Question 4**
*Let $u$ and $v$ be two* continuous *functions on $]0, L[$, which are smooth on every element $e_i$ (example: piece-wise linear functons considered in Chapter 1 of the lecture notes). Show that the second sum on the left hand side of (A.31) equals zero in this case. In this sense, the weak form (A.31) reduces to the weak form you have used in conjunction with the continuous Galerkin based FEM if no discontinuous functions are allowed.*

**Question 5**
*Let us choose the test function $v$ to be smooth over every element $e_i = ]x_i, x_{i+1}[$, $i = 1, \ldots, M-1$ (it still may be discontinuous across inter-elemental boundaries). Let $u$ be a solution to the model problem (A.27) (in particular, both $u$ and $\kappa u'$ are continuous on $]0, L[$). Show that such $u$ and $v$ satisfy the equation (A.31). Hint: multiply equation (A.27) with the test function $v$ and inegrate over every element $e_i$, $i = 1, \ldots, M-1$. Integrate by parts over every element (this is possible because $v$ is smooth over every element). Now add contributions from every element. Certain terms disappear from (A.31) because of the continuity of $u$, while others will simplify owing to the continuity of $\kappa u'$.*

## Finite element basis functions

Construction of global shape function is extremely simplified within discontinuous Galerkin framework as we do not need to enforce inter-elemental continuity. Therefore, we may start by specifying local basis shape functions for every element, and then simply use a collection of all local shape functions associated with all finite elements as a collection of global shape functions.

Suppose we want to construct a method, which uses piece-wise linear approximations on each element. We need two coefficients to represent a linear function, resulting in two basis functions associated with every element. In principle, we can use any pair of linearly independent linear polynomials, such as for example:

$$N_1^{(e)} = \begin{cases} 1, & \text{if } x_e < x < x_{e+1}, \\ 0, & \text{otherwise,} \end{cases} \qquad N_2^{(e)} = \begin{cases} (2x - (x_e + x_{e+1}))/h_e, & \text{if } x_e < x < x_{e+1}, \\ 0, & \text{otherwise.} \end{cases} \tag{A.32}$$

We can now expand a solution to a finite element problem in terms of these basis functions:

$$u(x) = \sum_{e=1}^{M-1} \sum_{i=1}^{2} \hat{u}_i^{(e)} N_i^{(e)}(x). \tag{A.33}$$

## Implementation

### Ordering of degrees of freedom

To ease the discussion of the implementation, let us choose the following ordering of degrees of freedom in the problem:

$$\hat{u} = (\hat{u}_1^{(1)}, \hat{u}_2^{(1)}, \hat{u}_1^{(2)}, \hat{u}_2^{(2)}, \ldots, \hat{u}_1^{(M-1)}, \hat{u}_2^{(M-1)})^T \in \mathbb{R}^{2(M-1)}. \tag{A.34}$$

| $e$ | 1 | 2 |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 3 | 4 |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $M-2$ | $2M-5$ | $2M-4$ |
| $M-1$ | $2M-3$ | $2M-2$ |

| $n$ | 1 | 2 |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 1 | 2 |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $M-1$ | $M-2$ | $M-1$ |
| $M$ | $M-1$ | 0 |

Table A.1: The content of arrays `EDOF` (left) and `VToE` (right). Note that in the latter array we use zero to indicate that a vertex has no adjacent element on the left or on the right.

### Data structures

It could be convenient to assemble a few data structures in order to treat the finite element assembly in a uniform way: `VX` , `EToV` , `VToE` , and `EDOF` . The arrays `VX` and `EToV` have been described in Section 1.1 of the lecture notes. In array `EDOF` we will store the degrees of freedom, associated with each element (we did not need this array in conjunction with linear continuous Galerkin method, as in that case numbers of vertices and numbers of degrees of freedom coincide). The array `VToE` will contain the information about vertex-to-element connectivity, necessary to evaluate the "jump" contributions to the stiffness matrix (second sum in (A.31)). The content of these arrays for our 1D mesh is shown in Table A.1.

### Elemental contributions

Every element contributes with a $2 \times 2$ elemental stiffness matrix, computed as

$$k_{ij}^{(e)} = \int_{x_e}^{x_{e+1}} \kappa(x)(N_i^{(e)})'(N_j^{(e)})' \, dx, \tag{A.35}$$

where $i = 1, 2$, $j = 1, 2$, $x_e =$ `EToV(e,1)`, $x_{e+1} =$ `EToV(e,2)`, and formulas for $N_i^{(e)}$ are given by (A.32). If $\kappa(x)$ is constant over the element $e$, then quantities (A.35) may be pre-computed analytically; otherwise, numerical quadratures may be used (see see Appendix B in the lecture notes).

Assembly of all local contributions into the global stiffness matrix can be done as follows:

```
for e=1:(M-1),
  xe = VX(EToV(e,:));
  ke = element_matrix(xe,kappa);
  A(EDOF(e,:),EDOF(e,:)) = A(EDOF(e,:),EDOF(e,:))+ke;
end
```

Similarly, contributions of elements to the right hand side are given by

$$f_i^{(e)} = \int_{x_e}^{x_{e+1}} f(x) N_i^{(e)} \, dx, \tag{A.36}$$

and they may be assembled into the global vector as follows:

```
for e=1:(M-1),
  xe = VX(EToV(e,:));
  fe = element_vector(xe,f);
  b(EDOF(e,:)) = b(EDOF(e,:))+fe;
end
```

### Interior nodal contributions

Every interior node $n = 2, \ldots, M-1$ is adjacent to two elements $e_1^n = \texttt{VToE(n,1)}$ and $e_2^n = \texttt{VToE(n,1)}$. It contributes with a $2 \times 2$ block matrix

$$k^{(n)} = \begin{pmatrix} k^{(n),1,1} & k^{(n),1,2} \\ k^{(n),2,1} & k^{(n),2,2} \end{pmatrix}, \tag{A.37}$$

where each block $k^{(n),\ell,m}$, $\ell = 1, 2$, $m = 1, 2$ is in turn a $2 \times 2$ matrix resulting from the second sum in the weak form (A.31). After careful examination of (A.31), (A.28), and (A.29), one arrives at the following expressions for $k_{ij}^{(n),\ell,m}$:

$$\begin{aligned}
k_{ij}^{(n),1,1} &= -0.5\kappa(x_n)(N_j^{e_1^n})'(x_n^-)N_i^{e_1^n}(x_n^-) - 0.5\kappa(x_n)(N_i^{e_1^n})'(x_n^-)N_j^{e_1^n}(x_n^-) + \frac{\delta\kappa(x_n)}{\min(h_{e_1^n}, h_{e_2^n})} N_j^{e_1^n}(x_n^-)N_i^{e_1^n}(x_n^-), \\
k_{ij}^{(n),2,2} &= +0.5\kappa(x_n)(N_j^{e_2^n})'(x_n^+)N_i^{e_2^n}(x_n^+) + 0.5\kappa(x_n)(N_i^{e_2^n})'(x_n^+)N_j^{e_2^n}(x_n^+) + \frac{\delta\kappa(x_n)}{\min(h_{e_1^n}, h_{e_2^n})} N_j^{e_2^n}(x_n^+)N_i^{e_2^n}(x_n^+), \\
k_{ij}^{(n),1,2} &= -0.5\kappa(x_n)(N_j^{e_2^n})'(x_n^+)N_i^{e_1^n}(x_n^-) + 0.5\kappa(x_n)(N_i^{e_1^n})'(x_n^-)N_j^{e_2^n}(x_n^+) - \frac{\delta\kappa(x_n)}{\min(h_{e_1^n}, h_{e_2^n})} N_j^{e_2^n}(x_n^+)N_i^{e_1^n}(x_n^-), \\
k_{ij}^{(n),2,1} &= +0.5\kappa(x_n)(N_j^{e_1^n})'(x_n^-)N_i^{e_2^n}(x_n^+) - 0.5\kappa(x_n)(N_i^{e_2^n})'(x_n^+)N_j^{e_1^n}(x_n^-) - \frac{\delta\kappa(x_n)}{\min(h_{e_1^n}, h_{e_2^n})} N_j^{e_1^n}(x_n^-)N_i^{e_2^n}(x_n^+),
\end{aligned} \tag{A.38}$$

$i = 1, 2$, $j = 1, 2$.

Finally, global assembly of contributions from intertior nodes may be done as follows:

```
for n=2:(M-1),
  e12=VToE(n,:);
  e12dof=[EDOF(e12(1),:), EDOF(e12(2),:)];
  [kn]=nodal_matrix(n,VX,EToV,VToE,kappa,delta);
  A(e12dof,e12dof) = A(e12dof,e12dof)+kn;
end
```

### Boundary contributions

Boundary nodes contribute to both the matrix and the right hand side of the resulting linear system. One should be careful and remember that jump and average operators are defined differently for the boundary nodes, see (A.30). Therefore, the contribution of the node 1 to the system is defined with the expressions for the block $k^{(n),2,2}$, see (A.38), with $n = 1$ and coefficient 0.5 replaced with 1.0. Similarly, for the node $n = M$ we can take the expression for the block $k^{(M),1,1}$ replacing 0.5 with 1.0.

Finally, contributions of the boundary nodes to the right hand side of the system are computed as follows:

$$\begin{aligned}
f_i^{(1)} &= +\kappa(x_1)(N_i^{(1)})'(x_1^+)c + \frac{\delta\kappa(x_1)}{h_1} N_i^{(1)}(x_1^+)c, \\
f_i^{(M)} &= -\kappa(x_M)(N_i^{(M-1)})'(x_M^-)d + \frac{\delta\kappa(x_M)}{h_{M-1}} N_i^{(M-1)}(x_M^-)d,
\end{aligned} \tag{A.39}$$

$i = 1, 2$.

### Error estimate

One can show that there is a constant $\delta_0 > 0$, independent from the mesh size, such that for every $\delta > \delta_0$ the following error estimate holds:

$$\|u(x) - u_{\text{exact}}(x)\|_{L^2} = \left( \int_0^L [u(x) - u_{\text{exact}}(x)]^2 \, dx \right)^{1/2} \leq Ch^2, \tag{A.40}$$

where $u_{\text{exact}}$ is the exact solution to the model problem (A.27), and $u$ is given by (A.33) with expansion coefficients $\hat{u}$ being the solution to the system of linear equations resulting from the discontinuous Galerkin discretization. In (A.40), the constant $C$ is independent from the mesh size $h = \max_{1 \leq e \leq M-1} h_e$, but will generally speaking depend on the smoothness of the solution $u_{\text{exact}}$, behaviour of $\kappa$, and parameter $\delta$. If $u_{\text{exact}}$ is known, the integral in (A.40) can be evaluated using numerical integration over elements:

$$\|u(x) - u_{\text{exact}}(x)\|_{L^2}^2 = \sum_{e=1}^{M-1} \int_{x_e}^{x_{e+1}} [u_{\text{exact}}(x) - \hat{u}_1^{(e)} N_1^{(e)}(x) - \hat{u}_2^{(e)} N_2^{(e)}(x)]^2 \, dx. \tag{A.41}$$

### Test case

Solve the model problem (A.33) with the data $L = 1.0$, $\kappa(x) = 1.0 + x$, $u_{\text{exact}}(x) = \cos(2\pi x) \exp(-x)$ using the discontinuos Galerkin approach with parameter $\delta = 2.0$. The remaining parameters can be determined from the given data: $c = u_{\text{exact}}(0.0)$, $d = u_{\text{exact}}(L)$, $f(x) = -[\kappa(x) u'_{\text{exact}}(x)]'$.

Solve the problem on a sequence of refining meshes and verify the error against the estimate (A.40). Experiment with different quadrature orders.

### Possible generalizations

You may try to increase the order of polynomial approximation to obtain better convergence rates. E.g., one may add a quadratic shape function $N_3^{(e)}$ to all (or some) elements:

$$N_3^{(e)}(x) = 4h_e^{-2}(x - x_{e+1/2})^2.$$

It is also possible to treat $2D$ problems in a similar fasion. Jump terms in the weak form transform to integrals over edges of elements, and everything else essentially remains the same after defining

$$[\![w(x)]\!] = w_1(x)\vec{n}_1(x) + w_2(x)\vec{n}_2(x),$$
$$\{\kappa(x)\nabla w(x)\} = 0.5\kappa(x)\nabla w_1(x) + 0.5\kappa(x)\nabla w_2(x),$$

for $x \in E_1 \cap E_2$, an edge between two elements $E_1$ and $E_2$. $w_i$, $\nabla w_i$ is the value of the function or its gradient on element $E_i$, and $\vec{n}_i$ is the outwards directed normal for the element $E_i$.

### Literature notes

An accessible place to start learning about DG methods for elliptic and parabolic PDEs is a textbook by Béatrice Rivière [48]. A more technical treament of the material is found in [2].

### Numerical integration

During the week 1 of the course, you have transformed an integral over $]x_e, x_{e+1}[$ into an integral over $]-1, 1[$:

$$\int_{x_e}^{x_{e+1}} f(x) \, dx = \frac{h_e}{2} \int_{-1}^{1} f(x_{e+1/2} + y h_e/2) \, dy,$$

where $h_e = x_{e+1} - x_e$, and $x_{e+1/2} = (x_e + x_{e+1})/2$.

Further, an integral over $]-1, 1[$ can be numerically approximated using Gaussian quadratures:

$$\int_{-1}^{1} f(y)\, dy \approx \sum_{i=1}^{Q} f(q_i) w_i,$$

where $Q$ is the number of quadrature points, and $q_i$, $w_i$ are respectively coordinates and the weights associated with quadrature points. Gaussian quadratures are "optimal" in the sence that they are exact for polynomials up the the order $2Q - 1$. A Matlab function producing Gaussian quadrature points and weights up to the order 4 is listed below.

**Question 6**
*Verify that 1-point Gaussian quadrature is exact for every linear function $f(x) = ax + b$.*

```
function [q,w] = qgauss(n)
switch(n)
 case 1
  q=0;
  w=2;
 case 2
  q=[-1/sqrt(3); 1/sqrt(3)];
  w=[1; 1];
 case 3
  q=[-sqrt(3/5); 0; +sqrt(3/5)];
  w=[5/9; 8/9; 5/9];
 case 4
  q=[-sqrt((3+2*sqrt(6/5))/7); ...
     -sqrt((3-2*sqrt(6/5))/7); ...
      sqrt((3-2*sqrt(6/5))/7); ...
      sqrt((3+2*sqrt(6/5))/7)];
  w=[(18-sqrt(30))/36; (18+sqrt(30))/36; ...
     (18+sqrt(30))/36; (18-sqrt(30))/36];
 otherwise
  error('Unknown quadrature order');
end
```

## A.6 The ketchup mystery

*By Anton Evgrafov*

### Introduction

Have you ever wondered why ketchups behave almost as solids, that is until one tries to pour them out of their containers by shaking them violently, at which point the sauce suddenly demonstrates its liquid properties and spills all over? Many other liquids, for example yoghurts, whipped cream, paints, toothpastes, nail polish, blood, and some plastics behave in a similar fashion.

One way of explaining this phenomenon within the framework of fluid mechanics is to assume that fluid's viscosity may depend on the rate of shear deformation in the liquid. For a large class of fluids, including ketchups, viscosity decreases with more shear applied to the fluid—this is known as shear thinning, or pseudoplasticity. (This is in a stark contrast with Newtonian fluids, such as water, for which viscosity can be assumed to be a constant independent from the flow conditions.) When the ketchup bottle is shaken, the material inside becomes less viscous and as a result more mobile.

### The model problem

Let us consider a laminar, steady, incompressible fully developed flow of a shear-thinning fluid in a slab formed by two infinite planes $y = 0$ and $y = L$. We assume that the pressure driving the flow has the form $p(x, y, z) = \Delta p\, x$, where $\Delta p$ is a given constant, which implies that flow velocity is only non-zero in $x$ coordinate direction. Furthermore, this velocity may only vary in $y$ coordinate direction; we will denote this velocity with $u(y)$. Under these assumptions the Navier–Stokes equations simplify to

$$\begin{aligned} -\partial\tau/\partial y + \Delta p &= 0, \\ \tau &= \mu\, \partial u/\partial y, \end{aligned} \tag{A.42}$$

where $\tau$ is the shear stress and $\mu$ is the flow viscosity. As we already mentioned, for Newtonian fluids such as water their viscosity $\mu$ is a constant so that the system (A.42) becomes a system of linear differential equations. For non-Newtonian fluids, $\mu$ depends on the rate of shear $\partial u/\partial y$, thus rendering the system (A.42) non-linear. We assume the following model for our ketchup (this is an approximation of a Bingham pseudo-plastic model):

$$\mu = \mu_0 + g[\epsilon + (u')^2]^{-1/2}, \tag{A.43}$$

where we write $u' = \partial u/\partial y$, $\mu_0$ and $g$ are positive constants, and $\epsilon$ is a small positive constant (we recover Bingham plastic behaviour in the limit $\epsilon \downarrow 0$).

Putting (A.42) and (A.43) together, we arrive at the following non-linear boundary value problem:

$$\begin{aligned} -\frac{\partial}{\partial y}\left\{\left[\mu_0 + \frac{g}{[\epsilon + (\partial u/\partial y)^2]^{1/2}}\right]\frac{\partial u}{\partial y}\right\} + \Delta p = 0, \qquad & y \in ]0, L[ \\ u(0) = u(L) = 0. & \end{aligned} \tag{A.44}$$

### The variational formulation

Proceeding as in Chapter 1 of the notes, we multiply (A.44) with a piecewise-smooth continuous test function $v$ such that $v(0) = v(L) = 0$ and integrate by parts once. We finally obtain the

following system of equations:

$$\begin{aligned} &A(u,v)=0, \qquad \forall v: v(0)=v(L)=0, v \text{ is continuous and piecewise smooth,} \\ &u(0)=u(L)=0, \end{aligned} \tag{A.45}$$

where $A$ is linear in $v$ but is non-linear with respect to $u$.

**Question 7**
*Derive the exact expression for $A(u,v)$ entering (A.45).*

## Newton's algorithm

If we proceed directly to discretizing (A.45) by approximating $u$ in a basis of piece-wise linear shape-functions, we end up with a system of non-linear equations. Instead, we linearize the system of equations (A.45) and then apply FEM to the system of linear differential equations. Newton's algorithm is an iterative method based on such a successive linearization idea. At the stage $k=0,1,2,\ldots$ of the algorithm we compute the approximate solution $u^{(k+1)}$ based on the first order Taylor series expansion (linearization) at the point $u^{(k)}$. Namely, we write, for all test functions $v$ as in (A.45):

$$A(u^{(k+1)},v) \approx F^{(k)}(v) + J^{(k)}(\Delta u^{(k)},v) = 0, \tag{A.46}$$

where

$$\begin{aligned} \Delta u^{(k)} &:= u^{(k+1)} - u^{(k)}, \\ F^{(k)}(v) &:= A(u^{(k)},v), \\ J^{(k)}(w,v) &:= \lim_{t\downarrow 0} \frac{A(u^{(k)}+tw,v)-A(u^{(k)},v)}{t} = \frac{d}{dt}A(u^{(k)}+tw,v)|_{t=0}, \end{aligned} \tag{A.47}$$

is the directional derivative of $A$ with respect to $u$ in the direction $w$, evaluated at $(u^{(k)},v)$.

**Question 8**
*Derive the exact expression for $F^{(k)}$, $J^{(k)}$ entering (A.46).*

Note that (A.46) constitutes a variational formulation of a linear boundary value problem for $\Delta u^{(k)}$, which can be solved using the same approach as in Chapter 1 of the Notes. Since $u^{(k+1)}(0)=u^{(k)}(0)$ and $u^{(k+1)}(L)=u^{(k)}(L)$, it follows that $\Delta u^{(k)}(0)=\Delta u^{(k)}(L)=0$.

We are now ready to describe the Newton's algorithm.

1. Set $k=0$, choose a starting point $u_k$ verifying the boundary conditions $u_k(0)=u_k(L)=0$ (e.g., $u_k=0$).
2. Find an approximate $\Delta u^{(k)}$ solving (A.46) using FEM.
3. If $\|\Delta u^{(k)}\|$ (e.g., $L^2(0,L)$ norm) is small, stop; otherwise, set $u^{(k+1)} := u^{(k)} + \Delta u^{(k)}$ and $k := k+1$, go to step $(ii)$.

## Test case

Implement the Newton's algorithm for finding the flow profile of a non-Newtonian pseudoplastic fluid. Compare the parabolic profile one gets by setting $g=0$ with a plug-flow for $g>0$—note how the pseudo-plastic behaves as a rigid solid in the middle of the flow, that is, all material particles move with the same velocity. Note that for small $\epsilon$ (i.e., when our approximate model is close to a Bingham plastic model) and small $\Delta p$ (i.e., when small shear deformations are applied to the fluid) the material will not flow, which is consistent with our experiences with ketchup bottles.

## A.7 Dealing with systems of PDEs

*By Anton Evgrafov*

### Introduction

Most partial differential equations encountered in applications are in fact coupled systems of equations. In this assignment you will extend the framework of the Chapter 2 of the notes and consider one of the most important and ubiquitously occurring systems of coupled elliptic partial differential equations, namely the equations of linearized elasticity describing small defirmations of elastic solids.

### The model problem

Consider an elastic solid $\Omega \subset \mathbb{R}^2$. Let $u : \Omega \to \mathbb{R}^2$ be a function describing the displacement $u(x_1, x_2)$ of a point $(x_1, x_2) \in \Omega$ from its initial position when a force $f = (f_1, f_2) : \Omega \to \mathbb{R}^2$ is acting on the body. Then $u = (u_1, u_2)$ is known to satisfy the following system of partial differential equations:

$$-\sum_{j,k,l=1}^{2} \frac{\partial}{\partial x_j}\left[C_{ijkl}\frac{\partial u_l}{\partial x_k}\right] = f_i, \qquad i = 1, 2. \tag{A.48}$$

In (A.48), the coefficients $C_{ijkl} : \Omega \to \mathbb{R}$, $i, j, k, l = 1, 2$, describe material properties of the elastic solid. For most common materials we need to know only two constants $(\lambda, \mu)$, known as Lamé's parameters, to define all coefficients $C_{ijkl}$:

$$C_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}), \tag{A.49}$$

where $\delta_{ij}$ denotes standard Kronecker's delta symbol ($\delta_{ij} = 1$ if $i = j$, and 0 otherwise). Approximate values of Lamé's constants for some common materials are found in Table A.1.

| Material | $\lambda$, [N/m$^2$] | $\mu$, [N/m$^2$] |
|---|---|---|
| Steel | $1.0 \cdot 10^{11}$ | $7.7 \cdot 10^{10}$ |
| Aluminum | $5.0 \cdot 10^{10}$ | $2.6 \cdot 10^{10}$ |
| Rubber | $1.6 \cdot 10^{9}$ | $3.4 \cdot 10^{7}$ |

Table A.2: Approximate values of Lamé's constants for some commonly occurring materials.

To identify a unique solution $u$ to (A.48) we assume that the boundary $\partial\Omega$ is divided into two disjoint parts, $\partial\Omega = \Gamma_D \cup \Gamma_N$. On the Dirichlet part of the boundary, the values of the displacements are prescribed: $u(x_1, x_2) = \widetilde{u}(x_1, x_2)$, $(x_1, x_2) \in \Gamma_D$. On the Neumann part of the boundary, we know the external forces: $-\sum_{j,k,l=1}^{2} C_{ijkl}\frac{\partial u_l}{\partial x_k}n_j(x_1, x_2) = g_i(x_1, x_2)$, $i = 1, 2$, where $n : \partial\Omega \to \mathbb{R}^2$ is the outwards pointing normal for $\Omega$, and $g = (g_1, g_2) : \Gamma_N \to \mathbb{R}^2$ are the prescribed boundary traction forces.

### Variational formulation

Let us multiply the equation (A.48) with a piece-wise smooth test function $v : \Omega \to \mathbb{R}^2$, such that $v|_{\Gamma_D} = 0$. Utilizing the equation (A.49) and integrating by parts in (A.48), we arrive at the

variational formulation:

$$\sum_{k,l=1}^{2} \int_\Omega \lambda(x_1,x_2) \frac{\partial u_l}{\partial x_l}(x_1,x_2) \frac{\partial v_k}{\partial x_k}(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2 +$$
$$\sum_{k,l=1}^{2} \int_\Omega \mu(x_1,x_2) \frac{\partial u_l}{\partial x_k}(x_1,x_2) \left[ \frac{\partial v_l}{\partial x_k}(x_1,x_2) + \frac{\partial v_k}{\partial x_l}(x_1,x_2) \right] \mathrm{d}x_1\mathrm{d}x_2 = \tag{A.50}$$
$$\sum_k \int_\Omega f_k(x_1,x_2) v_k(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2 + \sum_k \int_{\Gamma_N} g_k(x_1,x_2) v_k(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2.$$

**Question 9**
*Provide the details of the derivation of* (A.57).

## Discretization

Let us consider a decomposition of $\Omega$ into triangular finite elements. Exactly as in Chapter 2 of the lecture notes, we will try to approximate the solution of (A.57) within the space of continuous functions, which are polynomial on every element. In order to carry on with this program we need an appropriate basis (shape) functions, which span the space of vector-valued piecewise polynomial functions. One way of constructing such a basis is as follows. We start with (scalar-valued) basis functions $N_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, M$ constructed in Chapter 2, and define for $i = 1, \ldots, M$

$$\widehat{N}_{2i-1}(x_1,x_2) = \begin{pmatrix} N_i(x_1,x_2) \\ 0 \end{pmatrix}, \qquad \widehat{N}_{2i}(x_1,x_2) = \begin{pmatrix} 0 \\ N_i(x_1,x_2) \end{pmatrix}. \tag{A.51}$$

**Question 10**
*Assuming that $N_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, M$ spans the space of all (scalar) continuous functions on $\Omega$ which are affine on every element, explain why $\widehat{N}_i : \Omega \to \mathbb{R}^2$, $i = 1, \ldots, 2M$ spans the space of all $\mathbb{R}^2$-vector valued continuous functions, which are affine on every element.*

We can now proceed exactly as in Chapter 2. That is, we expand an approximate solution in terms of the basis functions

$$\hat{u}(x_1,x_2) = \sum_{j=1}^{2M} \hat{u}_j \widehat{N}_j(x_1,x_2), \tag{A.52}$$

and substitute this expression into (A.57). We also take test functions $v$ to be the basis functions $\widehat{N}_j$, $j = 1, \ldots, 2M$. This results in a system of linear algebraic equations $A\hat{u} = b$, which we solve to determine the expansion coefficients in (A.59).

**Question 11**
*Provide a detailed description of elemental contributions to the global stiffness matrix and the right-hand-side vector of the resulting linear algebraic system (cf. Section 2.7 in the notes).*

## Test case

Let $\Omega = ]0, 1.0[\times]0, 0.5[ \subset \mathbb{R}^2$, and consider $\bar{u}(x_1,x_2) = (\sin(x_1), x_1\cos(3\pi x_2))$, $\Gamma_D = \{0\}\times]0, 0.5[$, $\Gamma_N = \partial\Omega \setminus \Gamma_D$. Assuming $\lambda$ and $\mu$ are constant in $\Omega$, compute the analytical expressions for $\widetilde{u} : \Gamma_D \to \mathbb{R}^2$, $f : \Omega \to \mathbb{R}^2$, $g : \Gamma_N \to \mathbb{R}^2$ corresponding to the given solution $\bar{u}$ of (A.48).

Implement an algorithm for solving the instance of (A.48) specified above using FEM in Matlab. Verify the correctness of your implementation by solving the test case problem on a sequence of refining quasi-uniform meshes and demonstrating that for small element sizes the error behaves as

$$\|\widehat{u} - \bar{u}\|_{[L^2(\Omega)]^2} = \left[ \int_\Omega \{[\widehat{u}_1(x_1,x_2) - \bar{u}_1(x_1,x_2)]^2 + [\widehat{u}_2(x_1,x_2) - \bar{u}_2(x_1,x_2)]^2\}\,\mathrm{d}x_1\mathrm{d}x_2 \right]^{1/2} = Ch^2, \tag{A.53}$$

where $h$ is the diameter of the largest element in the mesh. Use two materials with properties given in Table A.2: steel and rubber. Estimate and compare the size of the constant $C$ in (A.60) for two test cases.

## A.8 Iterative solution of the Laplace problem for Marine Hydrodynamics

*By Allan Peter Engsig-Karup*

### Introduction

For the efficient solution of large linear systems, one often needs to employ iterative methods. In Marine Hydrodynamics one of the most important solvers are based on Fully Nonlinear Potential Flow equations, that can be described as follows.

We consider the governing equation for FNPF applications stated in the Eulerian form. For a detailed derivation, e.g. see Engsig-Karup et al. (2013). A wave tank contains the fluid domain $\Omega \in \mathbb{R}^d$, that is a bounded, connected domain with a piece-wise smooth boundary $\Gamma \in \mathbb{R}^{d-1}$. The time domain is denoted by $T : t \geq 0$. The scalar velocity potential function $\phi(x, y, z, t) : \Omega \times T \to \mathbb{R}$ satisfies the Laplace problem

$$\phi = \tilde{\phi} \quad \text{on} \quad \Gamma^{FS}\,, \tag{A.54a}$$

$$\nabla^2\phi = 0 \quad \text{on} \quad \Omega\,, \tag{A.54b}$$

$$\nabla\phi \cdot \nabla h = 0 \quad \text{on} \quad \Gamma^{b}\,, \tag{A.54c}$$

$$\nabla\phi \cdot \mathbf{n}^{body} = 0 \quad \text{on} \quad \Gamma^{body}\,. \tag{A.54d}$$

Here $h(x, y) : \Gamma^b \to \mathbb{R}$ describes the still water depth, which is assumed constant in this work. The temporal evolution of the unsteady water surface is described by $z = \eta(x, y, t) : \Gamma^{FS} \times T \to \mathbb{R}$. The body surface normal vector is denoted with $\mathbf{n}^{body}$. The notations are illustrated in Fig. **??**. The unsteady free surface kinematic and dynamic boundary conditions are expressed in the Zakharov (1968) form

$$\partial_t\eta = -\tilde{\nabla}\eta \cdot \tilde{\nabla}\tilde{\phi} + \tilde{w}(1 + \tilde{\nabla}\eta \cdot \tilde{\nabla}\eta) \qquad \text{in} \quad \Gamma^{\mathrm{FS}} \times T, \tag{A.55a}$$

$$\partial_t\tilde{\phi} = -g\eta - \frac{1}{2}\left(\tilde{\nabla}\tilde{\phi} \cdot \tilde{\nabla}\tilde{\phi} - \tilde{w}^2(1 + \tilde{\nabla}\eta \cdot \tilde{\nabla}\eta)\right) \qquad \text{in} \quad \Gamma^{\mathrm{FS}} \times T, \tag{A.55b}$$

where the horizontal gradient operator $\tilde{\nabla} = (\partial_x, \partial_y)$ is introduced. The '$\sim$' is used to denote free surface variables defined at $z = \eta$, and $g$ is the gravitational acceleration (set to 9.81 m/s$^2$).

In this project, we will focus on solving the Laplace problem and make use of properties of the formulation to design an iterative method for solving the system.

# A.9 Solving maze puzzles - path planning using the Laplace equation

*By Allan Engsig-Karup & Poul Hjorth*

## Introduction

In the following we will consider a method for path planning using the Laplace equation inspired by the work of Connolly, Burns & Weiss (1995)[2]. We will exploit that solutions to Laplace problems have no local minima to find a path through maze puzzles.

In this assignment, you will use the framework of Chapter 1 and/or Chapter 2 of the notets and use these to solve a Laplace problem from which the path from entry to exit can be found for a maze. are you ready to beat your friends in solving complex mazes fast?

## The model problem

Consider the Laplace equation in a domain $\Omega$

$$\partial_{xx}u + \partial_{yy}u = 0, \quad (x,y) \in \Omega \tag{A.56}$$

with Dirichlet boundary conditions imposed at two open boundaries, namely, $u(x_{entry}, y_{entry}) = 1$ and $u(x_{exit}, y_{exit}) = 0$ on $(x,y) \in \Gamma^{entry/exit}$. At all other boundary segments we will assume wall boundaries implying that $\mathbf{n} \cdot \nabla u = 0$ for $(x,y) \in \Gamma^{walls}$, where $\mathbf{n}$ is an outward point normal vector at the boundary point in question.

[2]Connolly, Christopher & Burns, J & Weiss, R.. (1995). Path Planning Using Laplace's Equation.

## Variational formulation

Let us multiply the equation (A.48) with a piece-wise smooth test function $v : \Omega \to \mathbb{R}^2$, such that $v|_{\Gamma_D} = 0$. Utilizing the equation (A.49) and integrating by parts in (A.48), we arrive at the variational formulation:

$$\begin{aligned}
&\sum_{k,l=1}^{2} \int_\Omega \lambda(x_1,x_2) \frac{\partial u_l}{\partial x_l}(x_1,x_2) \frac{\partial v_k}{\partial x_k}(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2 + \\
&\sum_{k,l=1}^{2} \int_\Omega \mu(x_1,x_2) \frac{\partial u_l}{\partial x_k}(x_1,x_2) \left[ \frac{\partial v_l}{\partial x_k}(x_1,x_2) + \frac{\partial v_k}{\partial x_l}(x_1,x_2) \right] \mathrm{d}x_1\mathrm{d}x_2 = \\
&\sum_k \int_\Omega f_k(x_1,x_2) v_k(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2 + \sum_k \int_{\Gamma_N} g_k(x_1,x_2) v_k(x_1,x_2)\,\mathrm{d}x_1\mathrm{d}x_2.
\end{aligned} \tag{A.57}$$

**Question 12**
*Provide the details of the derivation of* (A.57).

## Discretization

Let us consider a decomposition of $\Omega$ into triangular finite elements. Exactly as in Chapter 2 of the lecture notes, we will try to approximate the solution of (A.57) within the space of continuous functions, which are polynomial on every element. In order to carry on with this program we need an appropriate basis (shape) functions, which span the space of vector-valued piecewise polynomial functions. One way of constructing such a basis is as follows. We start with (scalar-valued) basis functions $N_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, M$ constructed in Chapter 2, and define for $i = 1, \ldots, M$

$$\widehat{N}_{2i-1}(x_1,x_2) = \begin{pmatrix} N_i(x_1,x_2) \\ 0 \end{pmatrix}, \qquad \widehat{N}_{2i}(x_1,x_2) = \begin{pmatrix} 0 \\ N_i(x_1,x_2) \end{pmatrix}. \tag{A.58}$$

**Question 13**
*Assuming that $N_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, M$ spans the space of all (scalar) continuous functions on $\Omega$ which are affine on every element, explain why $\widehat{N}_i : \Omega \to \mathbb{R}^2$, $i = 1, \ldots, 2M$ spans the space of all $\mathbb{R}^2$-vector valued continuous functions, which are affine on every element.*

We can now proceed exactly as in Chapter 2. That is, we expand an approximate solution in terms of the basis functions

$$\hat{u}(x_1,x_2) = \sum_{j=1}^{2M} \hat{u}_j \widehat{N}_j(x_1,x_2), \tag{A.59}$$

and substitute this expression into (A.57). We also take test functions $v$ to be the basis functions $\widehat{N}_j$, $j = 1, \ldots, 2M$. This results in a system of linear algebraic equations $A\hat{u} = b$, which we solve to determine the expansion coefficients in (A.59).

**Question 14**
*Provide a detailed description of elemental contributions to the global stiffness matrix and the right-hand-side vector of the resulting linear algebraic system (cf. Section 2.7 in the notes).*

## Test case

Let $\Omega = ]0, 1.0[\times]0, 0.5[ \subset \mathbb{R}^2$, and consider $\bar{u}(x_1,x_2) = (\sin(x_1), x_1\cos(3\pi x_2))$, $\Gamma_D = \{0\}\times]0, 0.5[$, $\Gamma_N = \partial\Omega \setminus \Gamma_D$. Assuming $\lambda$ and $\mu$ are constant in $\Omega$, compute the analytical expressions for $\widetilde{u} : \Gamma_D \to \mathbb{R}^2$, $f : \Omega \to \mathbb{R}^2$, $g : \Gamma_N \to \mathbb{R}^2$ corresponding to the given solution $\bar{u}$ of (A.48).

Implement an algorithm for solving the instance of (A.48) specified above using FEM in Matlab. Verify the correctness of your implementation by solving the test case problem on a sequence of refining quasi-uniform meshes and demonstrating that for small element sizes the error behaves as

$$\|\widehat{u}-\bar{u}\|_{[L^2(\Omega)]^2}=\left[\int_\Omega\{[\widehat{u}_1(x_1,x_2)-\bar{u}_1(x_1,x_2)]^2+[\widehat{u}_2(x_1,x_2)-\bar{u}_2(x_1,x_2)]^2\}\,\mathrm{d}x_1\mathrm{d}x_2\right]^{1/2}=Ch^2, \tag{A.60}$$

where $h$ is the diameter of the largest element in the mesh. Use two materials with properties given in Table A.2: steel and rubber. Estimate and compare the size of the constant $C$ in (A.60) for two test cases.

# Appendix B

# Visualization

There are various commands for visualization in Matlab.

```
% 2-D Triangular plot (also works for quadrilaterals!)
>> triplot(EToV,VX,VY,'k')

% 3-D Visualization of solution
>> trimesh(EToV,VX,VY,u)

% 3-D Visualization of solution
>> trisurf(EToV,VX,VY,u)

% 3-D Visualization of part of solution
>> trisurf(EToV(idxlist,:),VX,VY,u)

% Visualization of node connections in matrix
>> gplot(A,p)
```

where `EToV` is a connectivity list and `VX` and `VY` are coordinate lists. Note that these commands can also be useful for debugging purposes.

For the visualization of the FEM solutions where first order polynomials are employed on each element, it is possible to use the connectivity table `EToV` together with the vertex node coordinate lists for plotting using a command of the form

```
>> triplot(EToV,VX,VY,'k')
```

To visualize a finite element interpolant or Spectral/*hp*-FEM solution of arbitrary order in two space dimensions it is possible to do a local triangulation of each element to make use of the same plotting routine in Matlab. For simple domains, it is possible to use the Delaunay algorithm in Matlab for creating a special connectivity table for the global coordinate lists and subsequently use this for plotting the solution using the commands

```
>> [TRI] = delaunay(x,y);
>> triplot(TRI,x,y);   % visualize mesh
```

This will work for simple convex domains.

# Appendix C

# Numerical integration

For the implementation of Finite Element Methods we need to deal with integrals. In one space dimension the integrals are typically of the form

$$\int_a^b f(x)dx \tag{C.1}$$

where $a$ and $b$ are the integration limits and $f(x)$ is some integrand represented as a function over the interval between the integration limits. To carry out the integration numerically, it is standard and convenient to first transform the integration limits to some standard interval, e.g. $r \in [-1, 1]$, such that

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^1 f(r)dr, \quad r = 2\frac{x-a}{b-a} - 1 \tag{C.2}$$

Then, numerical integration (also referred to as *quadrature*) is concerned with evaluating integrals by $n$-point finite summations of the form [17, 26, 27]

$$\int_{-1}^1 f(r)w(r)dr \cong \sum_{i=0}^{n-1} w_i f(r_i) \tag{C.3}$$

where $w_i$ are specified weights and $r_i$ are distinct nodes on the integration interval $-1 \leq r_i \leq 1$. In general, quadrature refer to numerical computation of univariate integrals (line integrals), and cubature for the numerical computation of a multiple integral (volume integral).

The most accurate numerical integration methods for the form (C.3) are known as *Gaussian quadratures*. There exist three different types of Gaussian quadrature, respectively, Gauss, Gauss-Radau and Gauss-Lobatto types. The Gauss quadratures are based on open integration where only interior points to the interval $-1 < r_i < 1$ is employed. In Gauss-Radau the interval include one of the endpoints $r_0 = -1$ or $r_{n-1} = 1$ of the interval, and in Gauss-Lobatto both endpoints $r_0 = -1$ and $r_{n-1} = 1$ are included. If the integrand is a polynomial on the reference interval $f(x) \in \mathcal{P}^{2n-k}([-1, 1])$, then the integral can be shown to be exact for the Gauss ($k = 1$), Gauss-Radau ($k = 2$) or the Gauss-Lobatto ($k = 3$) quadratures.

For the implementation of finite element type methods in one space dimension, gaussian quadrature of the form are typically used

$$\int_{-1}^1 f(r)dr \cong \sum_{i=0}^{n-1} w_i f(r_i) \tag{C.4}$$

where the weight function in (C.3) is a constant function of size unity, i.e. $w(r) = 1$. A gaussian quadrature of this type exists and is known as *Legendre integration*. To employ the formula, we need

a way to evaluate the integrand $f(r)$ values at appropriate nodes $-1 < r_i < 1$ and corresponding weights. Remark, that the integrand can be an arbitrary function, e.g. a product function of the type $f(r) = \psi(r)\phi(r)$. In such a case, one would need to be concerned about using sufficient collocation points to accurately represent the integrand $f(r)$ for the quadrature to be accurate.

However, a more general quadrature which contains the Legendre integration as a special case, but is also useful for multi-dimensional bases and integrals is the one based on the Jacobi polynomials

$$\int_{-1}^{1} (1-r)^{\alpha}(1+r)^{\beta} f(r)dr \cong \sum_{i=0}^{n-1} w_i^{\alpha,\beta} f(r_i) \tag{C.5}$$

Clearly, the special case for Legendre is one where $\alpha = \beta = 0$.

To determine the quadrature points and weights we can proceed as in [34] where the following recurrence for the general class of Jacobi polynomials is used

$$xP_n^{(\alpha,\beta)}(x) = a_n P_{n-1}^{(\alpha,\beta)}(x) + b_n P_n^{(\alpha,\beta)}(x) + a_{n+1} P_{n+1}^{(\alpha,\beta)}(x) \tag{C.6}$$

with coefficients given as

$$\begin{aligned} a_n &= \frac{2}{2n+\alpha+\beta}\sqrt{\frac{n(n+\alpha+\beta)(n+\alpha)(n+\beta)}{(2n+\alpha+\beta-1)(2n+\alpha+\beta+1)}} \\ b_n &= -\frac{\alpha^2-\beta^2}{(2n+\alpha+\beta)(2n+\alpha+\beta+2)} \end{aligned} \tag{C.7}$$

The recurrence is started using the initial values

$$\begin{aligned} P_0^{(\alpha,\beta)}(x) &= \sqrt{2^{-\alpha-\beta-1}\frac{\Gamma(\alpha+\beta+2)}{\Gamma(\alpha+1)\Gamma(\beta+1)}} \\ P_1^{(\alpha,\beta)}(x) &= \frac{1}{2}P_0^{(\alpha,\beta)}(x)\sqrt{\frac{\alpha+\beta+3}{(\alpha+1)(\beta+1)}}((\alpha+\beta+2)x+(\alpha-\beta)) \end{aligned} \tag{C.8}$$

with $\Gamma(x)$ the classical Gamma function.

To decided which type of polynomials the pair $(\alpha, \beta)$ should be chosen, e.g. Legendre polynomials ($\alpha = \beta = 0$), Chebyshev polynomials of the first kind ($\alpha = \beta = -\frac{1}{2}$) and of the second kind ($\alpha = \beta = \frac{1}{2}$). In fact, each of these classes of polynomials are special cases of ultraspherical polynomials ($\alpha = \beta$).

A Matlab script for evaluating Jacobi polynomials using this procedure is given in [34] and is called `acobiP.m . Furthermore, the same reference provides a Matlab script \verb` acobiGQ.m , which can be used to determine the corresponding weights for the quadrature formula on the Gauss quadrature points.

As an example, we can approximate the integral

$$I = \int_0^1 \sin(x)dx$$

using a Gauss-Legendre quadrature based on $n$ nodes with the commands

```
>> [r,w] = JacobiGQ(0,0,n-1);
>> b=1; a=0;
>> I = (b-a)/2*sin((r'+1)/2*(b-a)+a)*w;
```

where the computation of the integral follows the analytical expression in (C.4).

It is also possible to do this integration using the nodal spectral element Mass matrix on the $P+1$ Gauss-Legendre nodes ($P$ is then the maximum polynomial order in basis) starting with

```
>> [r] = JacobiGL(alpha,beta,P);
>> [V] = Vandermonde1D(P,r);
```

Then, construct the standard Mass Matrix according to (3.49).

```
>> M = inv(V'*V);
```

Integral can then be computed by the simple integration

```
>> I = (b-a)/2*ones(1,P+1)*M*sin((r+1)/2*(b-a)+a);
```

The latter approach allow for straightforward integration without the use of quadrature rules. Integration via a Mass Matrix is exact when the integrand can be represented by at most a polynomial of order $2P$.

The idea for numerical integration in multidimensional setting (cubature) [15] is the same as in the one-dimensional setting where Gauss quadrature is employed. Determine a set of nodes and corresponding weights and carry out the numerical integration by the finite summation of the inner product between the function values evaluated at the nodes and the corresponding weights while maximizing the order of the polynomial that can be integrated exactly.

# Appendix D

# Mesh generation

For the finite element solution of partial differential equations we are faced with the problem of generating a suitable mesh. This can be done using any mesh generator that can generate the required mesh data tables which completely defines the topology of the mesh.

Why unstructured meshes? Most realistic problem requires a realistic representation of the geometry. A main advantage of using unstructured meshes is the ability to generate meshes automatically using existing mesh generation methods. Unstructured meshes are more straightforward and requires less effort in general to adapt to a given geometry compared to a structured mesh. Furthermore, the unstructured meshes are easily built into a mesh hierarchy for locally adapting the mesh resolution through element subdivision.

The process of generating a mesh for Finite Element Analysis (FEA) should take into account considerations on how to choose the mesh size and density optimally for a given problem to minimize computational time, chose appropriate element types for the analysis to be performed to satisfy accuracy requirements and ensure that element shapes do not lead to singular or ill-conditioned numerical operations.

It is advantageous if we can make the distinction between the problem solution procedure involving setting up the solver and the problem of mesh generation, since then we can focus our effort on each problem separately. In the following we will consider how to generate unstructured meshes for our purposes with a mesh defined in terms of nonoverlapping simplices.

The two fundamental decision that we need to make is to choose

- the nodes of the interior of the domain that we want to represent discretely, and
- the edges of the discrete domain

The choices will determine the accuracy that can be achieved in comparison with the true solution that we are trying approximate.

## Mesh description

Any mesh topology and geometry is defined completely in terms of a set of unique vertex nodes and the connections among these. Thus the mesh problem consists of generating a coordinate table `p=[VX VY VZ]` holding the coordinates of the vertex nodes in the x-, y-, and z-directions, together with an Element-to-Vertex connection table `EToV`. In addition to these it is also common to specify boundary information which can be used in the problem solution procedure. For this purpose we may create a boundary table `BCType` holding information about the types of simplex boundaries (see Appendix D).

# Mesh generation techniques

## Delaunay

The Delaunay algorithm is a technique suitable for connecting points distributed over a domain in the review [47]. The technique can be used for the efficient triangulation of arbitrary complex domains.

The Delaunay algorithm may not preserve boundary integrity and therefore the boundary constraints needs to be taken into account - this is what is done in the so-called constrained Delaunay triangulation. To make sure that the discretization conforms to the boundary it is necessary to provide this information to the algorithm.

## DistMesh

DistMesh [44] is a simple mesh generator toolbox for Matlab which can be downloaded from the webpage http://www-math.mit.edu/~persson/mesh/ published by the authors. The package is easy to use and relies on the use of signed distance functions for defining the geometry of the spatial domains. For the generation of simple meshes the package is attractive as this can be done in very few lines of Matlab code.

The algorithm for generating a mesh using DistMesh is conceptually defined by the following four steps

1. Define a domain using signed distance functions.
2. Distribute a set of nodes interior to the domain.
3. Move interior nods to obtain force equilibrium.
4. Apply termination criterion when all nodes are (nearly) fixed in space.

Subsequently, it can be necessary to carry out some or all of the following steps to prepare the mesh for use with a numerical solver

1. Validate final output!.
2. Reorder element vertex nodes to be defined counter-clockwise (standard convention).
3. Setup boundary table.
4. Store mesh for reuse.

As an example, to generate a mesh for a square with a hole in the interior we can invoke the following commands

```
>> h0 = 0.125; % element size measure
>> radius = 0.4; % circle radius
>> xmin = -1; xmax = 1; ymin = -1; ymax = 1;
>> pfix = [xmin,ymin; xmin,ymin; xmin, ymax; xmax, ymax];
>> fd = @(p) ddiff(drectangle(p,xmin,xmax,ymin,ymax),dcircle(p,0,0,radius));
>> BoundingBox = [xmin,ymin;xmax,ymax];
>> [p,EToV] = distmesh2d(fd,@huniform,h0,BoundingBox,pfix);
```

For more details on the algorithm and other examples refer to [44].

DistMesh output is the two tables `p` and `EToV` and from these we can immediately generate the mesh constants

```
>> N=size(EToV,1);      % Number of elements
>> Nv=size(p,1);        % Number of vertex nodes in mesh
>> Nfaces=size(EToV,2); % Number of faces/element
>> VX = p(:,1);         % x-coordinates of vertex nodes
>> VY = p(:,2);         % y-coordinates of vertex nodes
```

When using the DistMesh mesh generator there is no guarantee that there are no duplicate nodes and that the vertex node orderings for the elements are the same. Duplicate nodes and counter-clockwise ordering of the vertex nodes can be obtained in two steps. First, removal of duplicate nodes and corresponding update of the mesh data tables

```
% identify unique vertex nodes
>> [p,I,J] = unique(p,'rows');
% change numbering of vertex nodes accordingly to sort order
>> EToV = reshape(J(EToV),size(EToV));
% remove duplicate entries in p and update EToV accordingly
>> [idx,I,J] = unique(EToV);
>> p = p(idx,:);
>> EToV = reshape(J,size(EToV));
```

Then the counter-clockwise ordering of the element vertex nodes can secured by a proper reordering based on the following metric

$$D = \hat{\mathbf{t}}_{31} \cdot \hat{\mathbf{n}}_{32} \tag{D.1}$$

where $\hat{\mathbf{t}}_{ij}$ is the direction vector from vertex node $i$ to vertex node $j$, and $\hat{\mathbf{n}}_{ij}$ is the orthogonal (right-hand rule) vector to the direction vector from vertex node $i$ to vertex node $j$. If $D < 0$ then the ordering is clockwise and if $D > 0$ the ordering is counter-clockwise. By detecting the elements with the wrong clockwise ordering, all we have to do is to exchange the second and third vertex nodes to ensure the same ordering of all elements.

```
function [EToV] = Reorder(EToV,VX,VY)
% Purpose: Reorder elements to ensure counter-clockwise orientation
ax = VX(EToV(:,1)); ay = VY(EToV(:,1));
bx = VX(EToV(:,2)); by = VY(EToV(:,2));
cx = VX(EToV(:,3)); cy = VY(EToV(:,3));
D = (ax-cx).*(by-cy)-(bx-cx).*(ay-cy);
i = find(D<0);
EToV(i,:) = EToV(i,[1 3 2]);
```

## Partial Differential Equation Toolbox (Matlab)

The Partial Differential Equation Toolbox in Matlab contains a few simple tools for the study and solution of partial differential equations (PDEs) in two-space dimensions (2D) and time using FEM. To solve a PDE using the toolbox and FEM the following six steps can be followed

1. Define Geometry (Draw in `pdetool`).
2. Specify boundary conditions for the Geometry.
3. Select proto-type PDE model problem that matches PDE problem to be solved. Define coefficients as appropriate.
4. Generate Mesh. Automated mesh generation and refinement procedures included.
5. Solve problem.
6. Visualization by various commands.

### Triangle

Triangle [51] is a state-of-the-art unstructured mesh generator for two-dimensional meshes and can be obtained from http://www.cs.cmu.edu/~quake/triangle.html.

A few of the main features of Triangle, is the opportunity to specify constraints on element angles and triangle areas, define holes and concavities and use exact arithmetic to improve robustness.

### Gmsh

Gmsh [23] is a three-dimensional finite element mesh generator which can also be used for creating two-dimensional meshes. The creation of meshes can be done interactively using a graphical user interface. Gmsh is distributed under the terms of the GNU General Public License (GPL).

### TetGen

TetGen is a three-dimensional finite element mesh generator which can be used to generate tetrahedral meshes of any 3D polyhedral domain. TetGen generates exact constrained Delaunay tetrahedralizations, boundary conforming Delaunay meshes, and Voronoi partitions. TetGen can be obtained form http://tetgen.berlios.de.

## Mesh post-processing

Having generated a mesh to be used for the solving a problem using a finite element method, we need to decide whether the mesh is suitable for our purposes. A decision can be made with support from various post-processing means for quantitatively assessing the quality of the mesh.

### Mesh quality

For any mesh that we will potentially use in our solution procedure we should be concerned about the mesh quality. The mesh quality can be assessed using various algebraic mesh quality measures [21,38,39,52]. The measures should make it possible to evaluate and choose good elements over bad elements according to their shape and sizes. To improve the quality of the mesh we can resort to mesh improvement techniques or at worst try and generate a new and better suitable mesh. Often the mesh quality is poor in local regions and if we target improvements to these local regions it is often possible to improve the mesh at a relatively cheap cost.

It is difficult to define a ”good” or optimal mesh. However, we are in general concerned about adequately representing the unknown solution using minimal number of elements for minimal cost in solution effort for a given numerical accuracy requirement. To achieve this we also need to represent the right geometry of the problem. If the geometry is not correct for the problem we are essentially solving a different problem. For example, if curvilinear boundaries are represented by straight-sided polygons we are introducing errors $\mathcal{O}(h)$ that reduce overall accuracy and reliability of the computed results.

There are three general rules dictated by error analysis for defining a ”good” mesh.

- Very large [5] and small element angles should be avoided.
- Elements should be placed most densely in regions where the solution of the problem are expected to vary rapidly.

- High accuracy requires a sufficiently fine mesh (or many nodes per element in the case where the solution is sufficiently smooth).

There are two basic strategies which can be combined for assessing the quality of a mesh *a priori* to using a given mesh, namely

- Visualize the mesh for inspection in the eyeball norm! (cf. Appendix B.)
- Assess the quality and detect bad elements using a computed quality measure.

In the following we adopt the strategy described in [44] for unstructured triangular meshes which gives a straightforward means to assess the quality of a given two-dimensional triangulated mesh. A common mesh quality measure is the ratio

$$q = 2\frac{r_{in}}{r_{out}} \tag{D.2}$$

where $r_{in}$ is the radius of the largest inscribed circle and $r_{out}$ is the smallest circumscribed circle to an element. The measure can be computed from Heron's formula for the area of the triangles together with the side lengths

$$r_{in} = \frac{A}{s}, \quad r_{out} = \frac{abc}{4A}, \quad A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \tfrac{1}{2}(a+b+c) \tag{D.3}$$

In Matlab the measure for each triangle can be computed using the commands

```
% Compute side lengths of triangles
>> lx = VX(EToV(:,[1 2 3])) - VX(EToV(:,[3 1 2]));
>> ly = VY(EToV(:,[1 2 3])) - VY(EToV(:,[3 1 2]));
>> l = sqrt(lx.^2+ly.^2);
>> a = l(:,1); b = l(:,2); c = l(:,3);
% Compute triangle measures
>> q = (b+c-a).*(c+a-b).*(a+b-c)./(a.*b.*c);
```

Using the computed measures it is possible to classify the triangles as follows

- Equilateral triangles have $q = 1$.
- Degenerate triangles have $q = 0$.
- "Good" triangles we defined as having $q > 0.5$ (rule of thumb).

Thus, with the measure (D.2) we can assess the quality and determine whether we should discard the mesh or try and improve the quality.

To improve mesh quality, it can be beneficial to apply some mesh smoothing procedure where the positions of the mesh nodes are adjusted without changing the mesh topology. For example, in the Matlab environment one can make use of `smoothmesh.m` which is a part of the Matlab toolbox Mesh2D v24. The toolbox can be obtained from Matlab Central Exchange. Often the mesh quality can improved remarkably with few smoothing operations, e.g., by using Laplacian smoothing based on weighted-Jacobi updates of the node positions as

$$\mathbf{x}_i^{(n+1)} = (1-\omega)\mathbf{x}_i^{(n)} + \frac{\omega}{S_i}\sum_{j=1}^{S_i}\mathbf{x}_j^{(n)} \tag{D.4}$$

where $S_i$ is the number of neighboring vertices of vertex $\mathbf{x}_i$ and $\omega$ is a relaxation parameter (fx 0.1). However, there are a few pitfalls that one should be aware of. Two typical problems are *mesh tangling* which can occur near reentrant corners and calls for special treatment, and reduction of local anisotropic mesh density (smearing of local mesh adaption) in the smoothing process.

## Mesh cleanup

The goal of mesh cleanup is to try and improve mesh quality by making local changes to the element connectivities, e.g. by improving local element angles by a diagonal node swap.

## Mesh refinement

The goal of mesh refinement is to minimize work effort through balancing accuracy with problem size. The mesh refinement process effectively reduces the local element sizes using techniques based on subdivision patterns, bisection and point insertion. A mesh refinement procedure should take into account the geometry of the domain of interest and one should therefore be careful to maintain an accurate description of the domain boundaries.

Mesh refinement techniques are typically based on

- Finding a reliable and cheap error estimation procedure
- Selecting a refinement criteria
- Assuming that a mesh is optimal when errors are equilibrated over the domain of interest
- Find a robust way to refine the existing mesh efficiently in few steps

## Creating boundary maps

To solve a PDE using a FEM solver we have created, we need to be able to impose boundary conditions at the correct locations in space. Thus, it is useful to have a simple procedure requiring minimal user input for determining special boundary maps (index lists) for imposing different boundary conditions. This requires a few steps, which are illustrated for use in Matlab (see also [34, Appendix B]) and makes use of freely available Matlab routines found in the DistMesh package described in [44].

A boundary table for all element faces is easily obtained from the `EToE` array as

```
>> BCType = int8(not(EToE));
```

However, if we want to be able to distinguish the different boundaries, we need an efficient way to specify boundary properties. This can conveniently be done using a distance function with the properties that $d = 0$ on the boundary and $d <> 0$ outside the boundary.

Let us consider an example. We set up a mesh for a unit circle with a hole in the middle as

```
>> h0 = 0.1;
>> fd=inline('-0.3+abs(0.7-sqrt(sum(p.^2,2)))');
>> [p,t]=distmeshnd(fd,@huniform,h0,[-1,-1;1,1],[]);
```

which is easily generated with three lines of code using DistMesh.

We wish to define two maps `mapI` and `mapO` that allow us to specify different boundary conditions on the inner and outer boundary of the unit circle with a hole. We note that the following distance functions describe the boundaries completely.

```
>> fd_inner = inline('sqrt(sum(p^2,2))-0.4','p');
>> fd_outer = inline('sqrt(sum(p^2,2))-1','p');
```

Using these functions, we can determine index lists of the vertex nodes on the boundaries by

```
>> tol = h0/1e3;
>> nodesInner = find(abs(fd_inner(p))<tol);
>> nodesOuter = find(abs(fd_outer(p))<tol);
```

The tolerance `tol` is defined such that nodes which are not positioned exactly (to the level of machine precision) at the boundary as defined by the signed distance functions can still be found. It is recommended to always check the defined boundary maps before making use of them. An easy way to do this is to plot them as

```
% Choose a map to plot
>> MAP = nodesInner;
% Show all vertex nodes and circle out map nodes in red
>> plot(p(:,1),p(:,2),'k.', p(MAP,1), p(MAP,2),'ro');
```

Now, we define the following conventions using constants

```
>> In=1; Out=2;
```

and use them to correct the face information in the table `BCType` by

```
>> BCType = CorrectBCTable_v2(EToV,VX,VY,BCType,nodesInner,fd_inner,In);
>> BCType = CorrectBCTable_v2(EToV,VX,VY,BCType,nodesOuter,fd_outer,Out);
```

which makes use of BCType to insert the correct codes for boundaries into that array. The routine `CorrectBCTable_v2` which can be used with straight-sided elements is defined as

```
function BCType = CorrectBCTable_v2(EToV,VX,VY,BCType,fd,BCcode)
% Purpose: Store boundary information in the BCType array
%    EToV   : Element-To-Vertice table
%    VX, VY : (x,y)-coordinates of mesh vertices
%    BCType : Table with types of faces for BC's
%    fd     : handle to distance function defining boundary
%    BCcode : Integer for specific boundary type
VNUM = [1 2;2 3;3 1]; % face orientations
pxc = 0.5*(VX(EToV)+VX(EToV(:,[2 3 1])));
pyc = 0.5*(VY(EToV)+VY(EToV(:,[2 3 1])));
dc  = abs(fd([pxc(:) pyc(:)])); % distances to boundaries from face centers
tol = 5e-2; % tolerance
idx = find(dc<tol);
BCType(idx) = BCcode;
return
```

Once this is generated, we just have to create the appropriate boundary maps for use in our routines, e.g. in the routines for Algorithm 17. The creation of special boundary maps is illustrated using the script `ConstructNodesMap.m` after having corrected the face information of `BCType`.

```
>> mapI = ConstructNodesMap(BCType,c,P,In);
>> mapO = ConstructNodesMap(BCType,c,P,Out);
```

where $P$ is the maximum polynomial order of the basis functions and `c` is the local-to-global connectivity table (construct with Algorithm 14). The routine `ConstructNodesMap.m` can be defined as

```
function [map] = ConstructNodesMap(BCType,c,P,BCcode)
```

```
Nfaces = size(BCType,2); Npf = P+1;
[elms,fids] = find(BCType == BCcode);
map = zeros(length(elms)*Npf,1);
count = 0;
for n = elms
    for f = fids
        lidx = [1:Nfaces Nfaces+(f-1)*(Npf-2)+(1:Npf-2)]; % local face indexes
        map(count+(1:Npf)) = c(n,lidx); % vertex nodes
        count = count + Npf;
    end
end
map = unique(map);
return
```

# Appendix E

# Pieces of approximation theory

## The Lebesque constant

Consider the approximation problem. Let $\mathbb{U}$ be a subspace of the normed linear space $\mathbb{L}$. Given $f \in \mathbb{L}$, determine an *approximation* $f^* \in \mathbb{U}$ such that

$$||f - f^*|| = \min_{g \in \mathbb{U}} ||f - g|| \tag{E.1}$$

Here $||f - g||$ is a measure of the distance between the two element $f$ and $g$ in the same linear space $\mathbb{L}$ using some appropriate norm. Thus, in the approximation problem we are looking for an approximation $f^*$ to a function $f$ in a linear space with minimal approximation error. For example, the best approximation to $f$ with minimal maximum error over the interval in question is called the *minimax approximation* to $f$.

Assume that we approximate $f$ with some polynomial $P_N$ of order $N$. Define the approximation error

$$e(x) = f - f^* = f - P_N \tag{E.2}$$

Then

$$||e|| = ||f - P_N|| = ||f - P_N - P_N^{opt} + P_N^{opt}|| \leq ||f - P_N^{opt}|| + ||P_N^{opt} - P_N|| \tag{E.3}$$

The first term is then the *minimax approximation error* for a polynomial of order $N$

$$\rho_N[f(x)] = ||f - P_N^{opt}|| \tag{E.4}$$

The second term, can be expressed as follows by applying the Lebesgue lemma which provides a bound for the projection error

$$||P_N^{opt} - P_N|| \leq ||P_N|| \cdot ||f(x) - P_n^{opt}(x)|| \tag{E.5}$$

where

$$||P_N|| \equiv \max(||P_N(f(x)))$$

and accordingly

$$||e(x)|| \leq (1 + \Delta_N)\rho_N[f(x)] \tag{E.6}$$

This error estimate states that the largest error we commit relative to the minimax approximation is governed by the Lebesgue constant $\Delta_N$ which is defined from

$$\Delta_N \equiv \max(\mathcal{L}_N(x)) \tag{E.7}$$

with the interpolating lagrange polynomial defined in terms of the approximation nodes as

$$\mathcal{L}_N(_x) \equiv \sum_{i=1}^{N+1} |l_{N,i}(x)| \tag{E.8}$$

Thus, this result suggests finding nodal sets which minimizes the Lebesque constant for the approximation space in question.

# Appendix F

# Auxiliary routines reference

A few auxiliary routines can be used for the development of the FEM toolbox in Matlab as described in this set of lecture notes. These routines are given here in full and the authors who supplied them as a part of their own work are acknowledged for their contribution by reference.

**JacobiGQ.m**

*Reference: [34]*

*Routine for computing weights and nodes for a gauss quadrature*

**Vandermonde1D.m**

*Reference: [34]*

*Routine for computing generalized Vandermonde matrix*

**GradVandermonde1D.m**

*Reference: [34]*

*Routine for computing generalized Vandermonde matrix with derivatives of the reference basis*

**JacobiP.m**

*Reference: [34]*

*Routine for computing Jacobi polynomials*

**GradJacobiP.m**

*Reference: [34]*

*Routine for computing the gradient of the Jacobi polynomials*

**JacobiGL.m**

*Reference: [34]*

*Routine for computing the Gauss-Legendre-Lobatto nodes*

**Nodes2D.m**

*Reference: [34]*

**tiConnect2D.m**

*Reference: [34]*

*Routine for computing additional mesh connection tables*

**Normals2DFEM.m**

*Reference: [34]*

*Routine for computing element normals from geometric weights*

**Warpfactor.m**

*Reference: [34]*

**xytors.m**

*Reference: [34]*

**rstoab.m**

*Reference: [34]*

**Simplex2DP.m**

*Reference: [34]*

**Vandermonde2D.m**

*Reference: [34]*

**GradSimplex2DP.m**

*Reference: [34]*

# Bibliography


[1] N. C. Albertsen. Added mass integral calculated by conformal mapping. IMM report 2 (10 pp.), 2001.

[2] Douglas N Arnold, Franco Brezzi, Bernardo Cockburn, and L Donatella Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5):1749–1779, 2001.

[3] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[4] O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems*. SIAM, 2001.

[5] I. Babuška and A. K. Aziz. On the angle condition in the finite element method. *SIAM J. Num. Anal.*, 13(2):214–226, 1976.

[6] I. Babuška and M. Suri. The $p$ and $h-p$ versions of the finite element method, basic principles and properties. *SIAM Review*, 36(4):578–632, 1994.

[7] I. Babuška and M. R. Dorr. Error estimates for combined $h$ and $p$ versions of the finite element method. *Numer. Math.*, 37:257–277, 1981.

[8] I. Babuška, B. A. Szabo, and I. N. Katz. The $p$-version of the finite element method. *SIAM J. Numer. Anal.*, 18:515–545, 1981.

[9] V. A. Barker and J. Reffstrup. The finite element method for partial differential equations, 1998.

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[11] J. P. Boyd. *Chebyshev and Fourier spectral methods*. DOVER Publications, Inc., 2000.

[12] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer, 2008. 3rd ed.

[13] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral methods in fluid dynamics*. Springer, 1987.

[14] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral methods - Fundamentals in single domains*. Springer, 2006.

[15] Ronald Cools. An encyclopaedia of cubature formulas. *J. Complex.*, 19:445–453, June 2003.

[16] S. H. Crandall. *Engineering analysis*. McGraw-Hill, 1956; AMR 12(1959), Rev. 1122, 1959.

[17] P. J. Davis and P. Rabinowitz. *Methods of numerical integration*. Computer Science and Applied Mathematics, Academic Press, New York, 1975.

[18] M.O. Deville, P.F. Fischer, and E.H. Mund. *High Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002.

[19] M. Dubiner. Spectral methods on triangles and other domains. *J. Scient. Comp.*, 6:345–390, 1991.

[20] L. Eldén, L. Wittmeyer-Koch, and H. B. Nielsen. *Introduction to Numerical Computation*. Studentlitteratur, 2004.

[21] D. A. Field. Qualitative measures for initial meshes. *Int. J. Num. Meth. Engng.*, 47(4):887–906, 2000.

[22] B. A. Finlayson and L. E. Scriven. The method of weighted residuals - a review. *Applied MEchanics Review*, 19(9), 1966.

[23] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Num. Meth. Engng.*, 79(11):1309–1331, 2009.

[24] M. S. Gockenbach. *Understanding and Implementing the Finite Element Method*. SIAM, 2006.

[25] G. Golub and C. Van Loan. *Matrix Computations, third edition*. John Hopkins University Press, 1996.

[26] G. H. Golub and J. Kautsky. Calculation of gauss quadrature rules with multiple free and fixed knots. *Numer. Math.*, 41:147–163, 1983.

[27] G. H. Golub and J. H. Welsch. Calculation of gauss quadrature rules. *Mathematics of Computation*, 23:221–230, 1969.

[28] W. J. Gordon and C. A. Hall. Construction of curvilinear coordinate systems and their applications to mesh generation. *Int. J. Num. Meth. Engng.*, 7:461–477, 1973.

[29] D. Gottlieb and S. A. Orszag. *Numerical analysis of spectral methods: Theory and applications*, volume 26. SIAM, Philadelphia, PA, 1977.

[30] J. Grooss. *A Level Set Discontinuous Galerkin Method for Free Surface Flows – and Water-Wave Modeling*. PhD thesis, Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2005.

[31] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. The Johns Hopkins University Press, Baltimore, MD, 1993.

[32] L. A. Hageman and D. M. Young. *Applied Iterative Methods*. Academic Press, New York, 1981.

[33] J. S. Hesthaven, S. Gottlieb, and D. Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge Monographs on Applied And Computational Mathematics 21. Cambridge University Press, Cambridge, UK, 2007.

[34] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, 2008.

[35] A. Jennings and J. J. McKeown. *Matrix Computation*. Wiley, Chichester, 1992.

[36] G. E. Karniadakis and S. J. Sherwin. *Spectral/*hp *element methods for CFD*. Oxford University Press, 1999.

[37] Robert C. Kirby. Algorithm 839: Fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software*, 30(4):502–516, 2004.

[38] P. M. Knupp. Algebraic mesh quality metrics. *SIAM J. Sci. Comput.*, 23:193–218, 2001.

[39] Patrick M. Knupp. Algebraic mesh quality metrics for unstructured initial meshes. *Finite Elem. Anal. Des.*, 39(3):217–241, January 2003.

[40] J. D. Lambert. *Numerical Methods for Ordinary Differential Equations: The Initial Value Problem*. Wiley, Chichester, 1991.

[41] J. N. Newman. *Marine Hydrodynamics*. MIT Press, Cambridge, Massachusetts, 1977.

[42] H. B. Nielsen and P. G. Thomsen. Numeriske metoder for sædvanlige differentialligninger, 1993.

[43] A. T. Patera. A Spectral element method for fluid dynamics: Laminar flow in a channel expansion. *J. Comput. Phys.*, 54:468–488, 1984.

[44] P.-O. Persson and G. Strang. A simple mesh generator in Matlab. *SIAM Review*, 46(2):329–345, 2004.

[45] C. Pozridikis. *Introduction to Finite and Spectral Element Methods using Matlab*. Chapman & Hall/CRC, 2006.

[46] J. Proriol. Sur une famille de polynomes · deux variables orthogonaux dans un triangle. *C. R. Acad. Sci. Paris*, 245:2459–2461, 1957.

[47] S. Rebay. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *J. Comput. Phys.*, 106:125–138, 1993.

[48] Béatrice Rivière. *Discontinuous Galerkin methods for solving elliptic and parabolic equations*. SIAM, 2008.

[49] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[50] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003. 2nd ed.

[51] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[52] J. R. Shewchuk. What is a good linear finite element? interpolation, conditioning, anisotropy, and quality measures, 2002.

[53] G. Strang and G. J. Fix. *An analysis of the finite element method*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[54] B. A. Szabo and I. Babuška. *Finite Element Analysis*. J. Wiley & Sons, New York, 1991.

[55] M. A. Taylor, B. A. Wingate, and R. E. Vincent. An algorithm for computing fekete point in the triangle. *SIAM J. Numer. Anal.*, 38:1707–1720, 2000.

[56] P. G. Thomsen and S. Mayer. A little note on the physical background of the technical exercise. IMM report, 2002.

[57] Andrea Toselli and Olof B. Widlund. *Domain Decomposition Methods - Algorithms and Theory*. Springer, 2005.

[58] P. Šolín, K. Segeth, and I. Doležel. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC, 2004.

[59] T. Warburton. An explicit construction for interpolation nodes on the simplex. *J. Engineering Math.*, 56(3):247–262, 2006.