MATLAB® *and* C Programming *for* Trefftz Finite Element Methods

MATLAB® *and* C Programming *for* Trefftz Finite Element Methods

Qing-Hua Qin
Australian National University
Acton, Australia

Hui Wang
Henan University of Technology
Zhengzhou City, China

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A TAYLOR & FRANCIS BOOK

Consultant editor: Prof. D. R. Vij.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-7275-4 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Qin, Qing-Hua.
Matlab and C programming for Trefftz finite element methods / Qing-Hua Qin and Hui Wang.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4200-7275-4 (alk. paper)
1. Finite element method--Data processing. 2. MATLAB. 3. C (Computer program language) I. Wang, Hui, 1976- II. Title.

TA347.F5Q248 2008
518'.2502855268--dc22 2008013045

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Preface

The Hybrid-Trefftz finite element method, employing a class of finite elements associated with the Trefftz method, is a hybrid method which includes the use of an auxiliary inter-element displacement or traction frame to link the internal displacement fields of the elements. These internal fields, chosen so as to *a priori* satisfy the governing differential equations, have conveniently been represented as the sum of a particular integral of non-homogeneous equations and a suitably truncated T-complete set of regular homogeneous solutions (the series of the homogeneous solution to the differential governing equation) multiplied by undetermined coefficients. Inter-element continuity is enforced by using a modified variational principle together with an independent frame field defined on each element boundary. Formulation of the element, during which the internal parameters are eliminated at the element level, in the end leads to the standard force-displacement relationship, with a symmetric positive definite stiffness matrix. In contrast to conventional conforming finite element methods, the main advantages of the Trefftz finite element method are (a) the formulation calls for integration along the element boundaries only, which enables arbitrary polygonal or even curve-sided elements to be generated; (b) the method is likely to represent the optimal expansion bases for hybrid-type elements where inter-element continuity need not be satisfied *a priori*, which is particularly important for generating a quasi-conforming plate bending element; and (c) the method offers the attractive possibility of developing accurate crack singular, corner or perforated elements, simply by using appropriate known local solution functions as the trial functions of the intra-element displacements.

Research to date on the Trefftz finite element method has resulted in the publication of a great deal of new information and has given us a powerful computational tool in the analysis of plane elasticity, thin and thick plate bending, Poisson's equation, heat conduction, and piezoelectric materials. Articles on this subject have been published in a wide range of journals, attracting the attention of both researchers and practitioners with backgrounds in the mechanics of solids, applied physics, applied mathematics, mechanical engineering and materials science. Up to the present, however, no extensive, detailed computer programming treatment of this subject has been available, which significantly hinders the development and application of this new method in practical engineering. Although there is an enormous gulf between the basic theory and a working computer code, no textbooks have been devoted primarily to the programming aspects of the method. It now appears timely to collect significant information and to present a unified computer programming treatment of these useful but scattered results. Those results should be made available to professional engineers, research scientists, workers and students in applied mechanics and

material engineering, such as physicists, metallurgists and materials scientists. That is the motivation to write this book.

The purpose of this book is to provide the detailed MATLAB®* and C programming processes in applications of the Trefftz finite element method to potential and elastic problems, helping readers to take what may be a painful step from mathematical formulation to computer programming, enabling them to develop appropriate programs for their own particular applications. The methods and programming process in this book are described with the objective of making them easy to understand and accessible to research scientists, professional engineers and postgraduate students.

This book consists of nine chapters and four appendices. The first chapter gives an introduction and overview of the Hybrid-Trefftz finite element method. Basic concepts and general element formulations of the method are described, as well as comparison of T-elements with conventional finite elements and boundary elements. Chapters 2 and 3 concentrate on the essentials of MATLAB and C programming, to provide a source for reference in later chapters. Chapter 4 presents MATLAB and C subroutines commonly applied in Chapters 5-8. Applications of T-elements to potential problems and linear plane elasticity are described in Chapters 5 and 6. Chapter 7 deals with domain integrals caused by generalised body force terms. Based on the concept used in dual-reciprocity boundary element method, an approach to deal with arbitrary sources in potential problems and body force in elasticity by way of radial basis functions is presented. Chapter 8 describes MATLAB and C programming for special circular hole elements. Some advanced topics, including construction of Trefftz *p*-elements, dimensionless transformation, and an alternative formulation to HT-FEM, are discussed in Chapter 9.

Many people have been most generous in their support of this writing effort. We would like to especially thank Professor D. R. Vij of Kurukshetra University, Dr. John Navas and Ms. Marsha Pronin of Taylor & Francis Group for their commitment to excellence in all aspects of the publication of this book. We are very grateful to the reviewers who made suggestions and comments for updating and improving the text. Part of the research results presented in this text was obtained by the authors at the Department of Engineering, Australian National University, Australia; the College of Civil Engineering and Architecture, Henan University of Technology, China; and the Department of Mechanics, Tianjin University, China. Support from these universities and the Australian Research Council is gratefully acknowledged. Finally, we wish to acknowledge the individuals and organizations cited in the book for their permission to use that material.

Qing-Hua Qin and Hui Wang

*For product information, please contact: The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, Tel: 508-647-7000, Fax: 508-647-7001, E-mail: info@mathworks.com, Web: www.mathworks.com.

Contents

1 Introduction to Trefftz finite element method 1
- 1.1 Historical background . . . 1
- 1.2 Trefftz finite element procedure . . . 3
  - 1.2.1 Basic field equations and boundary conditions . . . 3
  - 1.2.2 Assumed fields . . . 4
  - 1.2.3 Element stiffness equation . . . 8
- 1.3 Variational principles . . . 9
- 1.4 Concept of the T-complete solution . . . 10
- 1.5 Comparison of Trefftz FEM and conventional FEM . . . 12
  - 1.5.1 Assumed element fields . . . 12
  - 1.5.2 Variational functionals . . . 14
  - 1.5.3 Assessment of the two techniques . . . 15
- 1.6 Comparison of T-elements with boundary elements . . . 17
  - 1.6.1 Boundary elements . . . 19
  - 1.6.2 T-elements . . . 20
  - 1.6.3 Assessment of the two numerical models . . . 20

References . . . 23

2 Foundation of MATLAB programming 29
- 2.1 Introduction . . . 29
- 2.2 Basic data types in MATLAB . . . 29
  - 2.2.1 Array and variable . . . 29
  - 2.2.2 Types of variables . . . 30
  - 2.2.3 Built-in variables . . . 30
- 2.3 Matrix manipulations . . . 31
  - 2.3.1 Initialising matrix variable . . . 31
  - 2.3.2 Matrix indexing . . . 31
  - 2.3.3 Common array and matrix operations . . . 32
  - 2.3.4 Hierarchy of operations . . . 35
- 2.4 Control structures . . . 36
  - 2.4.1 Relational and logical operators . . . 36
  - 2.4.2 The `if` construct . . . 37
  - 2.4.3 The `switch` construct . . . 37
  - 2.4.4 The `for` loop . . . 38
  - 2.4.5 The `while` loop . . . 38
  - 2.4.6 Jump commands for loop control . . . 39

2.5 M-file functions . . . 40
2.5.1 M-file function structure . . . 40
2.5.2 Global and local variables . . . 42
2.5.3 Executing an m-file function . . . 42
2.6 I/O file manipulation . . . 43
2.6.1 Open and close a file . . . 43
2.6.2 Input manipulation . . . 44
2.6.3 Output manipulation . . . 46
2.7 Vectorization programming with MATLAB . . . 47
2.8 Common built-in MATLAB functions . . . 48
References . . . 50

3 C programming **53**
3.1 Data types, variable declaration and operators . . . 53
3.1.1 Data types . . . 53
3.1.2 Variable declaration . . . 54
3.1.3 Operators . . . 56
3.2 Control structures . . . 57
3.2.1 `if-else` structure . . . 57
3.2.2 `switch-case` structure . . . 58
3.2.3 `for` loop . . . 59
3.2.4 `while` loop . . . 60
3.2.5 Jump control commands . . . 61
3.3 Advanced array and pointer action . . . 61
3.3.1 Arrays . . . 62
3.3.2 Pointers . . . 62
3.3.3 Pointers and arrays . . . 63
3.3.4 Initialisation of array and storage management . . . 63
3.4 Functions and parameter transfer . . . 64
3.4.1 Types of functions . . . 64
3.4.2 Function call and parameter transfer . . . 66
3.5 File manipulation . . . 68
3.5.1 Open and close a file . . . 68
3.5.2 Input data from a file . . . 69
3.5.3 Output data to a file . . . 69
3.6 Create and execute C codes in visual C++ platform . . . 69
3.6.1 Creating a project . . . 70
3.6.2 Creating a C source file . . . 70
3.6.3 Compile, build and execute a C program . . . 72
3.6.4 Output result . . . 73
3.7 Common library functions and related head files . . . 73
References . . . 75

4 Commonly used subroutines **77**
4.1 Introduction . . . 77
4.2 Input and output . . . 77
4.2.1 Input of data . . . 77
4.2.2 MATLAB codes for input of data . . . 81
4.2.3 C codes for input of data . . . 84
4.2.4 Output of results . . . 87
4.2.5 MATLAB codes for output of results . . . 88
4.2.6 C codes for output of results . . . 90
4.3 Numerical integration over element edges . . . 92
4.3.1 MATLAB codes . . . 93
4.3.2 C codes . . . 94
4.4 Shape functions along element edge . . . 96
4.4.1 MATLAB codes . . . 97
4.4.2 C codes . . . 99
4.5 Assembly of elements . . . 100
4.5.1 MATLAB codes . . . 101
4.5.2 C codes . . . 101
4.6 Introduction of essential boundary conditions . . . 102
4.6.1 MATLAB codes . . . 104
4.6.2 C codes . . . 105
4.7 Solution of global stiffness equation . . . 106
4.7.1 MATLAB codes . . . 108
4.7.2 C codes . . . 108
References . . . 111

5 Potential problems **113**
5.1 Introduction . . . 113
5.2 Basic equations of potential problems . . . 114
5.3 Trefftz FE formulation . . . 115
5.3.1 Non-conforming intra-element field . . . 115
5.3.2 Auxiliary conforming frame field . . . 116
5.3.3 Modified variational principle . . . 117
5.3.4 Recovery of rigid-body motion . . . 119
5.4 T-complete functions . . . 120
5.5 Computation of H and G matrix . . . 122
5.5.1 Geometric characteristics of boundary edges . . . 122
5.5.2 Computation of H matrix . . . 122
5.5.3 Computation of G matrix . . . 123
5.6 Computation of equivalent nodal load . . . 124
5.7 Program structure for HT-FEM . . . 125
5.8 MATLAB programming for potential problems . . . 125
5.9 C computer programming . . . 141
5.10 Numerical examples . . . 171
References . . . 185

6 Plane stress/strain problems **187**
6.1 Introduction . . . 187
6.2 Linear theory of elasticity . . . 187
6.2.1 Equilibrium equations . . . 188
6.2.2 Strain-displacement relations . . . 188
6.2.3 Constitutive relations (stress-strain relations) . . . 189
6.2.4 Boundary conditions . . . 190
6.2.5 Governing equations in terms of displacements . . . 191
6.3 Trefftz finite element formulation . . . 191
6.3.1 Non-conforming intra-element field . . . 191
6.3.2 Auxiliary conforming frame field . . . 192
6.3.3 Modified variational functional . . . 194
6.3.4 Recovery of rigid-body motion . . . 196
6.4 T-complete functions . . . 196
6.5 Computation of H and G matrix . . . 199
6.5.1 Geometric characteristics of boundary edges . . . 199
6.5.2 Computation of matrix **H** . . . 200
6.5.3 Computation of matrix **G** . . . 201
6.6 Evaluation of equivalent nodal loads . . . 201
6.7 MATLAB functions for plane elastic problems . . . 203
6.8 C computer programming . . . 214
6.9 Numerical examples . . . 234
References . . . 244

7 Treatment of inhomogeneous terms using RBF approximation **247**
7.1 Introduction . . . 247
7.2 Radial basis functions . . . 248
7.2.1 Basics of radial basis functions . . . 248
7.2.2 RBF approximation . . . 249
7.2.3 Stability and convergence of RBF approximation . . . 251
7.3 Non-homogeneous problems . . . 253
7.3.1 Basic equations for Poisson's problems . . . 253
7.3.2 Basic equations for plane stress/strain problems . . . 255
7.4 Solution procedure of HT-FEM for non-homogeneous problems . 256
7.4.1 Assumed fields . . . 256
7.4.2 Variational functional . . . 257
7.4.3 Discussion . . . 258
7.5 Particular solutions in terms of RBFs . . . 258
7.5.1 Particular solutions for Poisson's equation . . . 259
7.5.2 Particular solutions for plane stress/strain equations . . . 260
7.6 Modification of the program structure . . . 265
7.6.1 Forming equivalent nodal flux . . . 265
7.6.2 Introducing nodal potential condition . . . 266
7.7 MATLAB functions for particular solutions . . . 266
7.7.1 Two-dimensional Poisson's problems . . . 266

7.7.2 Plane stress/strain problems . . . 277
7.8 C programming . . . 291
7.8.1 Two-dimensional Poisson's problems . . . 291
7.8.2 Plane stress/strain problems . . . 309
7.9 Numerical examples . . . 330
7.9.1 Poisson's problems . . . 330
7.9.2 Plane stress/strain problems . . . 334
References . . . 340

8 Special purpose T-elements 343
8.1 Introduction . . . 343
8.2 Basic concept of special Trefftz functions . . . 343
8.3 Special purpose elements for potential problems . . . 346
8.3.1 Trefftz-complete solutions for circular hole elements . . . 346
8.4 Special purpose elements for linear elastic problems . . . 347
8.4.1 Special Trefftz solutions for circular hole elements . . . 347
8.5 Programming implementation . . . 354
8.5.1 Data preparation . . . 354
8.5.2 Special Trefftz functions . . . 355
8.5.3 Output quantities . . . 355
8.6 MATLAB functions for special T-elements . . . 355
8.6.1 Potential problems . . . 355
8.6.2 Elastic problems . . . 366
8.7 C programming for special T-elements . . . 379
8.7.1 Potential problems . . . 379
8.7.2 Elastic problems . . . 394
8.8 Test examples . . . 410
8.8.1 Potential problems . . . 410
8.8.2 Plane elastic problems . . . 413
References . . . 418

9 Advanced topics for further programming development 419
9.1 Introduction . . . 419
9.2 Construction of Trefftz elements . . . 419
9.3 Dimensionless transformation . . . 420
9.3.1 Dimensionless transformation in regular HT element for plane potential problems . . . 421
9.3.2 Dimensionless transformation in special HT element for plane potential problems . . . 425
9.3.3 Dimensionless transformation in regular element for plane elastic problems . . . 425
9.3.4 Dimensionless transformation in hole element for plane elastic problems . . . 426
9.4 Nodal stress evaluation-smooth techniques . . . 427
9.5 Generating intra-element points for outputting field results . . . 429

9.6 Sparse matrix generation and solving procedure 429
9.7 An alternative formulation to HT-FEM 431
References 434

Appendix A Format of input data **437**

Appendix B Glossary of variables **439**

Appendix C Glossary of subroutines **443**

Appendix D Plane displacement and stress transformations **445**

1

Introduction to Trefftz finite element method

1.1 Historical background

The Trefftz finite element (FE) model, or T-element model for short, was originally developed in 1977 for the analysis of the effects of mesh distortion on thin plate elements [1]. During the subsequent three decades, the potential of T-elements for the solution of different types of applied science and engineering problems was recognised. Over the years, the Hybrid-Trefftz (HT) FE method has become increasingly popular as an efficient numerical tool in computational mechanics and has now become a highly efficient computational tool for the solution of complex boundary value problems. In contrast to conventional FE models, the class of FE associated with the Trefftz method is based on a hybrid method which includes the use of an auxiliary inter-element displacement or traction frame to link the internal displacement fields of the elements. Such internal fields, chosen so as to *a priori* satisfy the governing differential equations, have conveniently been represented as the sum of a particular integral of non-homogeneous equations and a suitably truncated T (Trefftz)-complete set of regular homogeneous solutions multiplied by undetermined coefficients. The mathematical fundamentals of T-complete sets were established mainly by Herrera and his co-workers [2 - 5], who named such a system a T-complete system. Following a suggestion by Zienkiewicz, Herrera changed this to a T-complete system of solutions, in honor of the originator of such non-singular solutions. As such, the terminology "TH-families" is usually used when referring to systems of functions which satisfy the criterion originated by Herrera [3]. Inter-element continuity is enforced by using a modified variational principle together with an independent frame field defined on each element boundary. The element formulation, during which the internal parameters are eliminated at the element level, leads to a standard force–displacement relationship, with a symmetric positive definite stiffness matrix. Clearly, while the conventional FE formulation may be assimilated to a particular form of the Rayleigh-Ritz method, the HT FE approach has a close relationship with the Trefftz method [6]. As noted in Refs. [7, 8], the main advantages stemming from the HT FE model are: (a) The formulation calls for integration along the element boundaries only, which enables arbitrary polygonal or even curve-sided elements to be generated. As a result, it may be considered a special, symmetric, substructure-oriented boundary solution approach and thus possesses the advantages of the boundary element method (BEM). In contrast to conventional boundary el-

ement formulation, however, the HT FE model avoids the introduction of singular integral equations and does not require the construction of a fundamental solution which may be very laborious to build. (b) The HT FE model is likely to represent the optimal expansion bases for hybrid-type elements where inter-element continuity need not be satisfied, *a priori*, which is particularly important for generating a quasi-conforming plate bending element. And (c) The model offers the attractive possibility of developing accurate crack-tip, singular corner or perforated elements, simply by using appropriate known local solution functions as the trial functions of intra-element displacements.

The first attempts to generate a general purpose HT FE formulation were published by Jirousek and Leon [1] and Jirousek [9]. It was immediately noticed that T-complete functions represented an optimal expansion basis for hybrid-type elements where inter-element continuity need not be satisfied *a priori*. Since then, the concept of Trefftz-elements has become increasingly popular, attracting a growing number of researchers into this field [10 - 23]. Trefftz-elements have been successfully applied to problems of elasticity [24 - 28], Kirchhoff plates [7, 22, 29 - 31], moderately thick Reissner-Mindlin plates [32 - 36], thick plates [37 - 39], general 3-D solid mechanics [20, 40], axisymmetric solid mechanics [41], potential problems [42, 43], shells [44], elastodynamic problems [16, 45 - 47], transient heat conduction analysis [48], geometrically nonlinear plates [49 - 52], materially nonlinear elasticity [53 - 55], and piezoelectric materials [56, 57]. Further, the concept of special purpose functions has been found to be of great efficiency in dealing with various geometries or load-dependent singularities and local effects (e.g., obtuse or reentrant corners, cracks, circular or elliptic holes, concentrated or patch loads. See Jirousek and Guex [30]; Jirousek and Teodorescu [24]; Jirousek and Venkatesh [25]; Piltner [27]; Venkatesh and Jirousek [58] for details). In addition, the idea of developing *p*-versions of T-elements, similar to those used in the conventional FE model, was presented in 1982 [24] and has already been shown to be particularly advantageous from the point of view of both computation and facilities for use [13, 59]. Huang and Li [60] presented an Adini's element coupled with the Trefftz method which is suitable for modeling singular problems. The first monograph to describe in detail the HT FE approach and its applications in solid mechanics was published in 2000 [61]. Moreover, a wealthy source of information pertaining to the HT FE approach exists in a number of general or special review type articles such as those of Herrera [12, 62], Jirousek [63], Jirousek and Wroblewski [9, 64], Jirousek and Zielinski [65], Kita and Kamiya [66], Qin [67 - 69], and Zienkiewicz [70].

1.2 Trefftz finite element procedure

1.2.1 Basic field equations and boundary conditions

Taking a plane stress problem as an example, the partial differential governing equation in the Cartesian coordinates X_i (i=1,2) is given by

$$\sigma_{ij,j} + b_i = 0 \qquad \text{in } \Omega \tag{1.1}$$

where σ_{ij} is the stress tensor, a comma followed by an index denotes partial differentiation with respect to that index, b_i is body force vector, Ω is the solution domain, and the Einstein summation convention over repeated indices is used. For a linear elastic solid, the constitutive relation is

$$\sigma_{ij} = \frac{\partial \Pi(\varepsilon_{ij})}{\partial \varepsilon_{ij}} = c_{ijkl}\varepsilon_{kl} \tag{1.2}$$

for ε_{ij} as basic variable, and

$$\varepsilon_{ij} = \frac{\partial \Psi(\sigma_{ij})}{\partial \sigma_{ij}} = s_{ijkl}\sigma_{kl} \tag{1.3}$$

for σ_{ij} as basic variable, where c_{ijkl} and s_{ijkl} are stiffness and compliance coefficient tensors, respectively, ε_{ij} is the elastic strain tensor, Π and Ψ are, respectively, potential energy and complementary energy functions, which are defined by

$$\Pi(\varepsilon_{ij}) = \frac{1}{2} c_{ijkl}\varepsilon_{ij}\varepsilon_{kl} \tag{1.4}$$

and

$$\Psi(\sigma_{ij}) = \frac{1}{2} s_{ijkl}\sigma_{ij}\sigma_{kl} \tag{1.5}$$

The relation between strain tensor and displacement component, u_i, is given by

$$\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \tag{1.6}$$

The boundary conditions of the boundary value problem (BVP) (1.1) - (1.6) are given by:

$$u_i = \bar{u}_i \qquad \text{on } \Gamma_u \tag{1.7}$$

$$t_i = \sigma_{ij} n_j = \bar{t}_i \qquad \text{on } \Gamma_t \tag{1.8}$$

where $\bar{u}_i$ and $\bar{t}_i$ are, respectively, prescribed boundary displacement and traction vector, a bar above a symbol denotes prescribed value, and $\Gamma = \Gamma_u + \Gamma_t$ is the boundary of the solution domain Ω.

In the Trefftz FE approach, Eqs. (1.1) - (1.8) should be completed by adding the following inter-element continuity requirements:

$$u_{ie} = u_{if} \qquad (\text{on } \Gamma_e \cap \Gamma_f, \text{ conformity}) \tag{1.9}$$

$$t_{ie} + t_{if} = 0 \qquad (\text{on } \Gamma_e \cap \Gamma_f, \text{ reciprocity}) \tag{1.10}$$

where "e" and "f" stand for any two neighbouring elements. Eqs. (1.1) - (1.10) are taken as a basis to establish the modified variational principle for Trefftz FE analysis in solid mechanics.

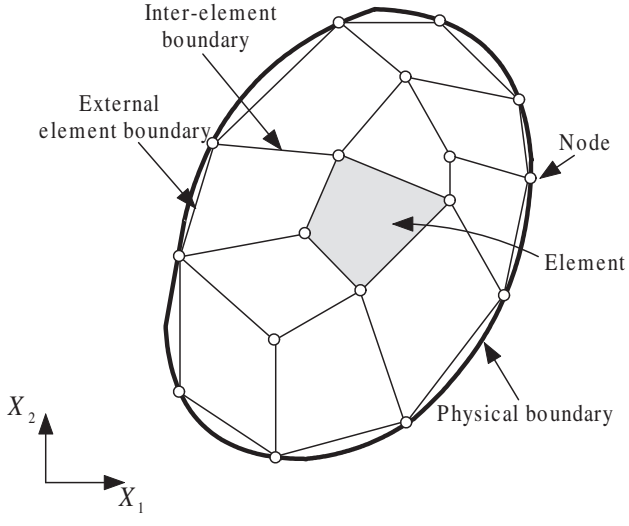


FIGURE 1.1
Configuration of hybrid Trefftz finite element method (FEM) discretisation

1.2.2 Assumed fields

The main idea of the HT FE model is to establish a FE formulation whereby inter-element continuity is enforced on a non-conforming internal field chosen so as to *a priori* satisfy the governing differential equation of the problem under consideration [61]. In other words, as an obvious alternative to the Rayleigh-Ritz method as a basis for a conventional FE formulation, the model here is based on the method of Trefftz [6], for which Herrera [71] gave a general definition as: *Given a region of an Euclidean space of some partitions of that region, a "Trefftz Method" is any procedure for solving boundary value problems of partial differential equations or systems of*

such equations, on such region, using solutions of those differential equations or its adjoint, defined in its subregions. With this method the solution domain Ω is subdivided into elements (see Figure 1.1), and over each element e, two independent fields are assumed in the following way:

(a) The non-conforming intra-element field is expressed by

$$\mathbf{u} = \breve{\mathbf{u}} + \sum_{i=1}^{m} \mathbf{N}_i \mathbf{c}_i = \breve{\mathbf{u}} + \mathbf{N}\mathbf{c} \tag{1.11}$$

where $\breve{\mathbf{u}}$ and $\mathbf{N}$ are known functions, $\mathbf{c}$ is a coefficient vector and m is its number of components. The choice of m is discussed in Section 2.6 of Ref. [61]. For the reader's convenience, we described here briefly the basic rule for determining m. It is important to choose the proper number m of trial functions $\mathbf{N}_i$ for the Trefftz element with the hybrid technique. The basic rule used to prevent spurious energy modes is analogous to that in the hybrid-stress model. The necessary (but not sufficient) condition for the matrix $\mathbf{H}_e$, defined in later Eq. (1.25), to have full rank is stated as [61]

$$m \geq j - r \tag{1.12}$$

where j and r are numbers of nodal degrees of freedom (DOF) of the element under consideration and of the discarded rigid-body motion terms, or more generally the number of zero eigenvalues ($r = 1$ for potential problem in Chapter 5 and $r = 3$ for the plane strain/stress problems in Chapter 6). Although the use of the minimum number $m = j - r$ of flux mode terms in Eq. (1.12) does not always guarantee a stiffness matrix with full rank, full rank may always be achieved by suitably augmenting m. The optimal value of m for a given type of element should be found by numerical experimentation.

Regarding the function $\mathbf{N}$, it can be determined in the following way. If the governing differential equations are written in terms of displacement fields, we have

$$\Re\mathbf{u}(\mathbf{x}) = \mathbf{b}(\mathbf{x}), \qquad (\mathbf{x} \in \Omega_e) \tag{1.13}$$

where $\Re$ stands for the differential operator matrix, $\mathbf{x}$ for the position vector, a bar above a symbol indicates the imposed quantities and Ω_e stands for the eth element sub-domain, then $\breve{\mathbf{u}} = \breve{\mathbf{u}}(\mathbf{x})$ and $\mathbf{N}_i = \mathbf{N}_i(\mathbf{x})$ in Eq. (1.11) have to be chosen such that

$$\Re\breve{\mathbf{u}} = \mathbf{b} \quad \text{and} \quad \Re\mathbf{N}_i = 0, \qquad (i = 1, 2, \ldots, m) \tag{1.14}$$

everywhere in Ω_e. Again, taking plane stress problems as an example, a complete system of homogeneous solutions $\mathbf{N}_i$ has been generated in a systematic way from Muskhelishvili's complex variable formulation [25]. For convenience, we list the

results presented in Ref. [25] as follows:

$$2G\mathbf{N}^*_{1k} = \begin{Bmatrix} \mathrm{Re}Z_{1k} \\ \mathrm{Im}Z_{1k} \end{Bmatrix} \text{ with } Z_{1k} = i\kappa z^k + ikz\bar{z}^{k-1} \tag{1.15}$$

$$2G\mathbf{N}^*_{2k} = \begin{Bmatrix} \mathrm{Re}Z_{2k} \\ \mathrm{Im}Z_{2k} \end{Bmatrix} \text{ with } Z_{2k} = \kappa z^k - kz\bar{z}^{k-1} \tag{1.16}$$

$$2G\mathbf{N}^*_{3k} = \begin{Bmatrix} \mathrm{Re}Z_{3k} \\ \mathrm{Im}Z_{3k} \end{Bmatrix} \text{ with } Z_{3k} = i\bar{z}^k \tag{1.17}$$

$$2G\mathbf{N}^*_{4k} = \begin{Bmatrix} \mathrm{Re}Z_{4k} \\ \mathrm{Im}Z_{4k} \end{Bmatrix} \text{ with } Z_{4k} = -\bar{z}^k \tag{1.18}$$

where G is shear modulus, $\kappa = (3-\nu)/(1+\nu)$ for plane stress and $\kappa = 3-4\nu$ for plane strain, ν is Poisson's ratio, and $z = x_1 + ix_2$ and $i = \sqrt{-1}$.

The particular solution $\breve{\mathbf{u}}$ can be obtained by means of its source function. The source function corresponding to Eq. (1.1) has been given in Ref. [61] as

$$u^*_{ij}(r_{pq}) = \frac{1+\nu}{4\pi E}\left[(\nu-3)\,\delta_{ij}\ln r_{pq} + (1+\nu)\,r_{pq,i}r_{pq,j}\right] \tag{1.19}$$

where E is Young's modulus, $r_{pq} = [(x_{1q}-x_{1p})^2 + (x_{2q}-x_{2p})^2]^{1/2}$, $u^*_{ij}(r_{pq})$ denotes the ith component of displacement at the field point q of the solid under consideration when a unit point force is applied in the jth direction at the source point p. Using this source function, the particular solution can be expressed by

$$\breve{\mathbf{u}} = \begin{Bmatrix} \breve{u}_1 \\ \breve{u}_2 \end{Bmatrix} = \int_\Omega b_j \begin{Bmatrix} u^*_{1j} \\ u^*_{2j} \end{Bmatrix} \mathrm{d}\Omega \tag{1.20}$$

Finally, the unknown coefficient $\mathbf{c}$ may be calculated from the conditions on the external boundary and/or the continuity conditions on the inter-element boundary. Thus Trefftz-element models can be obtained by using different approaches to enforce these conditions. In the majority of approaches a hybrid technique is usually used, whereby the elements are linked through an auxiliary conforming displacement frame which has the same form as in conventional FE method. This means that, in the T-element approach, a conforming potential (or displacement in solid mechanics) field should be independently defined on the element boundary to enforce the potential continuity between elements and also to link the coefficient $\mathbf{c}$, appearing in Eq. (1.11), with nodal DOF $\mathbf{d}$ (=$\{d\}$). Thus:

(b) An auxiliary conforming field

$$\tilde{\mathbf{u}}(\mathbf{x}) = \tilde{\mathbf{N}}(\mathbf{x})\,\mathbf{d}, \qquad (\mathbf{x} \in \Gamma_e) \tag{1.21}$$

is independently assumed along the element boundary in terms of nodal DOF $\mathbf{d}$, where the symbol "$\sim$" is used to specify that the field is defined on the element

boundary only, **d=d(c)** stands for the vector of the nodal displacements which are the final unknowns of the problem, Γ_e represents the boundary of element e, and $\tilde{\mathbf{N}}$ is a matrix of the corresponding shape functions, typical examples of which are displayed in Figure 1.2. The tractions $\mathbf{t} = \{t_1 \;\; t_2\}^{\mathrm{T}}$ appearing in Eq. (1.8) can also

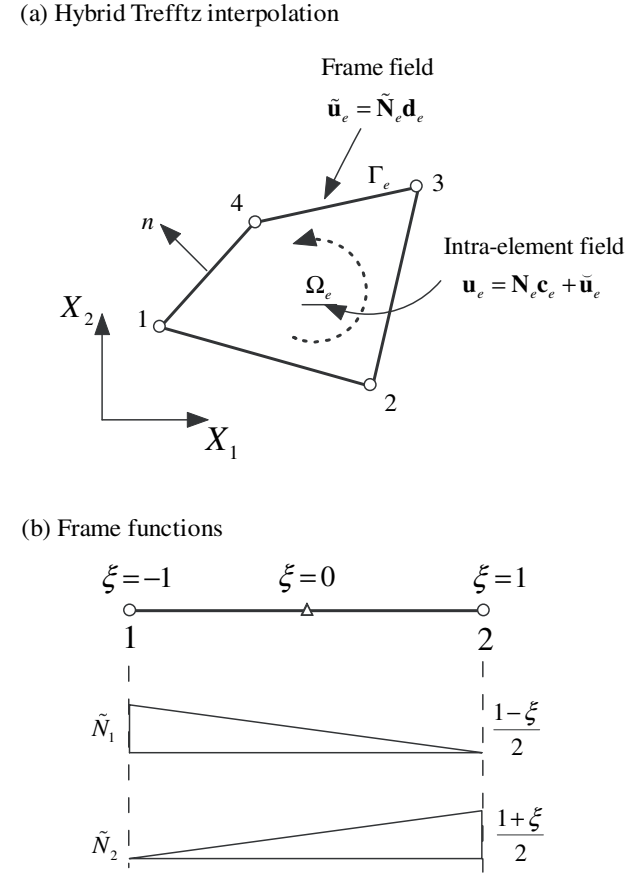


FIGURE 1.2
Typical 4-node HT element together with its frame functions

be expressed in terms of **c** as

$$\mathbf{t} = \begin{Bmatrix} t_1 \\ t_2 \end{Bmatrix} = \begin{Bmatrix} \sigma_{1j} n_j \\ \sigma_{2j} n_j \end{Bmatrix} = \mathbf{Q}\mathbf{c} + \breve{\mathbf{t}} \tag{1.22}$$

where **Q** and $\breve{\mathbf{t}}$ are, respectively, related to **N** and $\breve{\mathbf{u}}$ by way of Eqs. (1.2) and (1.8).

1.2.3 Element stiffness equation

Based on the two independent assumed fields presented above, the element matrix equation can be generated by a variational approach [61]. For a plane stress problem, the variational functional Ψ_{me} presented in Ref. [73] is used to derive the finite element equation:

$$\begin{aligned}\Psi_{me} &= \frac{1}{2}\int_{\Omega_e} b_i u_i \mathrm{d}\Omega + \frac{1}{2}\int_{\Gamma_e} t_i u_i \mathrm{d}\Gamma + \int_{\Gamma_{te}} (\bar{t}_i - t_i)\,\tilde{u}_i \mathrm{d}\Gamma \\ &\quad - \int_{\Gamma_{Ie}} t_i \tilde{u}_i \mathrm{d}\Gamma - \int_{\Gamma_{ue}} t_i \bar{u}_i \mathrm{d}\Gamma \\ &= \frac{1}{2}\int_{\Omega_e} b_i u_i \mathrm{d}\Omega + \frac{1}{2}\int_{\Gamma_e} t_i u_i \mathrm{d}\Gamma - \int_{\Gamma_e} t_i \tilde{u}_i \mathrm{d}\Gamma + \int_{\Gamma_{te}} \bar{t}_i \tilde{u}_i \mathrm{d}\Gamma \end{aligned} \tag{1.23}$$

in which Γ_{Ie} represents the inter-element boundary of the element e (see Fig. 1.1), and the relationship $\bar{u}_i = \tilde{u}_i$ on Γ_u has been used. Substituting the expressions given in Eqs. (1.11), (1.21) and (1.22) into (1.23) produces

$$\Psi_{me} = \frac{1}{2}\mathbf{c}^{\mathrm{T}}\mathbf{H}_e\mathbf{c} - \mathbf{c}^{\mathrm{T}}\mathbf{G}_e\mathbf{d} + \mathbf{c}^{\mathrm{T}}\mathbf{h}_e + \mathbf{d}^{\mathrm{T}}\mathbf{g}_e + \text{ terms without } \mathbf{c} \text{ or } \mathbf{d} \tag{1.24}$$

in which the matrices $\mathbf{H}_e$, $\mathbf{G}_e$ and the vectors $\mathbf{h}_e$, $\mathbf{g}_e$ are as follows:

$$\begin{aligned}\mathbf{H}_e &= \int_{\Gamma_e} \mathbf{Q}^{\mathrm{T}}\mathbf{N}\mathrm{d}\Gamma = \int_{\Gamma_e} \mathbf{N}^{\mathrm{T}}\mathbf{Q}\mathrm{d}\Gamma \\ \mathbf{G}_e &= \int_{\Gamma_e} \mathbf{Q}^{\mathrm{T}}\tilde{\mathbf{N}}\mathrm{d}\Gamma \\ \mathbf{h}_e &= \frac{1}{2}\int_{\Omega_e} \mathbf{N}^{\mathrm{T}}\mathbf{b}\mathrm{d}\Omega + \frac{1}{2}\int_{\Gamma_e} \left(\mathbf{Q}^{\mathrm{T}}\breve{\mathbf{u}} + \mathbf{N}^{\mathrm{T}}\breve{\mathbf{t}}\right)\mathrm{d}\Gamma \\ \mathbf{g}_e &= \int_{\Gamma_{te}} \tilde{\mathbf{N}}^{\mathrm{T}}\bar{\mathbf{t}}\mathrm{d}\Gamma - \int_{\Gamma_e} \tilde{\mathbf{N}}^{\mathrm{T}}\breve{\mathbf{t}}\mathrm{d}\Gamma \end{aligned} \tag{1.25}$$

where $\bar{\mathbf{t}} = \{\bar{t}_1 \;\; \bar{t}_2\}^{\mathrm{T}}$ is the prescribed traction vector.

To enforce inter-element continuity on the common element boundary, the unknown vector $\mathbf{c}$ should be expressed in terms of nodal DOF $\mathbf{d}$. An optional relationship between $\mathbf{c}$ and $\mathbf{d}$ in the sense of variation can be obtained from

$$\frac{\partial \Psi_{me}}{\partial \mathbf{c}^{\mathrm{T}}} = \mathbf{H}_e\mathbf{c} - \mathbf{G}_e\mathbf{d} + \mathbf{h}_e = 0 \tag{1.26}$$

This leads to

$$\mathbf{c} = \mathbf{H}_e^{-1}\left(\mathbf{G}_e\mathbf{d} - \mathbf{h}_e\right) \tag{1.27}$$

and then straightforwardly yields the expression of Ψ_{me} only in terms of $\mathbf{d}$ and other known matrices

$$\Psi_{me} = -\frac{1}{2}\mathbf{d}^{\mathrm{T}}\left(\mathbf{G}_e^{\mathrm{T}}\mathbf{H}_e^{-1}\mathbf{G}_e\right)\mathbf{d} + \mathbf{d}^{\mathrm{T}}\left(\mathbf{G}_e^{\mathrm{T}}\mathbf{H}_e^{-1}\mathbf{h}_e + \mathbf{g}_e\right) + \text{ terms without } \mathbf{d} \tag{1.28}$$

Therefore, the element stiffness matrix equation can be obtained by taking the vanishing variation of the functional Ψ_{me} as

$$\frac{\partial \Psi_{me}}{\partial \mathbf{d}^{\mathrm{T}}} = 0 \Rightarrow \mathbf{K}_e \mathbf{d} = \mathbf{P}_e \tag{1.29}$$

where $\mathbf{K}_e = \mathbf{G}_e^{\mathrm{T}} \mathbf{H}_e^{-1} \mathbf{G}_e$ and $\mathbf{P}_e = \mathbf{G}_e^{\mathrm{T}} \mathbf{H}_e^{-1} \mathbf{h}_e + \mathbf{g}_e$ are, respectively, the element stiffness matrix and the equivalent nodal flow vector. Expression (1.29) is the elemental stiffness-matrix equation for Trefftz FE analysis.

1.3 Variational principles

It can be seen from Section 1.2 that, in FE analysis, physical fields such as potential or stresses are obtained by way of minimising a variational functional. In general, the variational functional consists of all the energies associated with the particular FE model. The minimum of the functional is found by setting the derivative of the functional with respect to the unknown nodal parameters to zero. It is known from variational calculus [72] that the minimum of any function has a slope or derivative equal to zero. Thus, the basic equation for FE analysis is

$$\frac{\mathrm{d}\Pi(\mathbf{u})}{\mathrm{d}\mathbf{u}} = 0 \tag{1.30}$$

where Π is the energy functional and $\mathbf{u}$ is the unknown nodal potential or displacement to be calculated. The above simple equation is the basis for FE analysis. The functional Π and unknown $\mathbf{u}$ vary with the type of problem, as described in later chapters.

As was shown in Ref. [61], the functional Π should obey a relationship called Euler's equation. Substitution of the functional in the appropriate Euler's equation yields the differential equation of the physical system. Thus, the FE solution obeys the appropriate differential equation. In light of this understanding, a variational functional used for deriving HT FE formulation may be constructed in such a way that the stationary conditions of the functional satisfy Eqs. (1.7) - (1.10) for the category of HT-displacement formulation, as Eq. (1.1) is satisfied, *a priori*, through the use of the assumed field (1.11). Keeping this in mind, a modified variational functional used for deriving the HT-displacement element model can be constructed in the way presented in Ref. [73]. For the boundary value problem (BVP) described by Eqs. (1.1) - (1.10), the corresponding variational functional can be given in the form [73]

$$\Psi_m = \sum_e \Psi_{me} = \sum_e \left\{ \Psi_e + \int_{\Gamma_{te}} (\bar{t}_i - t_i)\, \tilde{u}_i \mathrm{d}\Gamma - \int_{\Gamma_{Ie}} t_i \tilde{u}_i \mathrm{d}\Gamma \right\} \tag{1.31}$$

$$\Pi_m = \sum_e \Pi_{me} = \sum_e \left\{ \Pi_e + \int_{\Gamma_{ue}} (\bar{u}_i - \tilde{u}_i)\, t_i \mathrm{d}\Gamma - \int_{\Gamma_{Ie}} t_i \tilde{u}_i \mathrm{d}\Gamma \right\} \tag{1.32}$$

where

$$\Psi_e = \int_{\Omega_e} \Psi(\sigma_{ij})\,\mathrm{d}\Omega - \int_{\Gamma_{ue}} t_i \bar{u}_i \mathrm{d}\Gamma \tag{1.33}$$

$$\Pi_e = \int_{\Omega_e} (\Pi(\varepsilon_{ij}) - b_i u_i)\,\mathrm{d}\Omega - \int_{\Gamma_{te}} \bar{t}_i \bar{u}_i \mathrm{d}\Gamma \tag{1.34}$$

in which Eq. (1.1) are assumed to be satisfied, *a priori*. The term "modified principle" refers to the use of a conventional complementary functional (Ψ_m here) and some modified terms for the construction of a special variational principle to account for additional requirements such as the condition defined in Eqs. (1.9) and (1.10).

The boundary Γ_e of a particular element consists of the following parts:

$$\Gamma_e = \Gamma_{ue} \cup \Gamma_{te} \cup \Gamma_{Ie} \tag{1.35}$$

where

$$\Gamma_{ue} = \Gamma_u \cap \Gamma_e, \quad \Gamma_{te} = \Gamma_t \cap \Gamma_e \tag{1.36}$$

It has been shown in Ref. [73] that the stationary condition of the functional (1.31) leads to Eqs. (1.7) - (1.10) and the existence of extremum of the functional (1.31), which ensures that an approximate solution can converge to the exact one. This indicates that the stationary condition of the functional satisfies the required boundary equations and can thus be used for deriving HT-displacement element formulations.

1.4 Concept of the T-complete solution

As discussed in Section 1.2, Trefftz FE formulation is based on two groups of independently assumed fields. One, defined on the element boundary, is called the frame field; the other, defined in the domain of the element, is often known as the intra-element field (or internal field). In contrast with the standard BEM in which singular (Green's type) solutions are used as trial functions, non-singular, T-complete functions (regular homogeneous solutions of the differential equation) are used as trial (or shape) functions in T-element formulation to interpolate the intra-element field. In this section we show how T-complete solutions are derived, as well as how they are related to the interpolation matrix $\mathbf{N}$ in Eq. (1.11), as is essential in establishing Trefftz FE formulation.

For simplicity, we take the Laplace equation as an example and consider the equation

$$\nabla^2 u = 0 \tag{1.37}$$

Its T-complete solutions are a series of functions satisfying Eq. (1.37) and being complete in the sense of containing all possible solutions in a given domain Ω. The T-complete function when weighted by any other function, say v, has the following

property:

$$\int_{\Omega} \left(\nabla^2 u\right) v \mathrm{d}\Omega = 0 \tag{1.38}$$

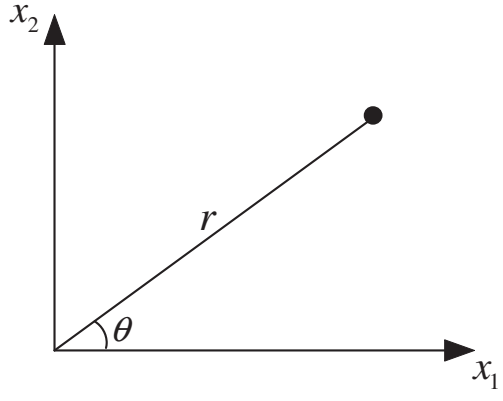


FIGURE 1.3
Illustration of the polar coordinate system

It can be shown that any of the following functions satisfies Eq. (1.37):

$$1,\ r\cos\theta,\ r\sin\theta,\cdots,\ r^m\cos m\theta,\ r^m\sin m\theta,\cdots \tag{1.39}$$

where r and θ are a pair of polar coordinates, as shown in Figure 1.3. As a consequence, the so-called T-complete set, denoted by $\mathbf{T}$, can be written as

$$\mathbf{T} = \left\{1,\ r^m\cos m\theta,\ r^m\sin m\theta\right\} = \left\{T_i\right\} \tag{1.40}$$

where $T_1 = 1,\ \ T_{2m} = r^m\cos m\theta$, and $T_{2m+1} = r^m\sin m\theta \quad (m = 1,2,\cdots)$.

The proof of the completeness of the T series (1.40) is beyond the scope of this book, and can be found in the work of Herrera [5].

Noting that the interpolation function $\mathbf{N}$ in Eq. (1.11) is also required to satisfy the condition (1.37), its components can be simply taken as those of $\mathbf{T}$, i.e.,

$$N_1 = r\cos\theta,\quad N_2 = r\sin\theta,\ \ N_3 = r^2\cos 2\theta,\ \ \cdots \tag{1.41}$$

1.5 Comparison of Trefftz FEM and conventional FEM

To enhance understanding of the difference between conventional finite element method (FEM) and Trefftz FEM, it is interesting to compare generally the formulation of T-elements with elements from conventional FE analysis. On the one hand, the conventional conforming assumed displacement model has become a popular and well-established tool in FE analysis. Theoretical and numerical experience has also clearly shown its high reliability and efficiency. On the other hand, an alternative FE model has recently been developed based on the Trefftz method, The main difference between these two models lies in their assumed displacement (or potential) fields and the variational functional used. This difference arises from the fact that the two models are based on opposite mathematical concepts (Ryleigh-Ritz for the conventional, Trefftz for T-element). Thus, engineers and researchers who have been exposed to conventional FE may ask why it is necessary to develop an alternative element model. The answer can be obtained through a systematic comparison of the two approaches. To this end, we consider a typical 4-node element in plane stress analysis (see Figure 1.4). The assumed fields within the element and the functionals used in the two models are compared, to provide an initial insight into the difference between them.

1.5.1 Assumed element fields

In the conventional element model, we can assume that the displacements u and v in the 4-node element (Figure 1.4) are given in the following form of polynomials in local coordinates variables x_1 and x_2:

$$\begin{aligned} u(x_1,x_2) &= c_1 + c_2x_1 + c_3x_2 + c_4x_1x_2 \\ v(x_1,x_2) &= c_5 + c_6x_1 + c_7x_2 + c_8x_1x_2 \end{aligned} \tag{1.42}$$

The unknown coefficients c_i $(i = 1,...,8)$, which are also called generalised coordinates, will be expressed in terms of the unknown element nodal point displacement vector $\mathbf{d}(= \{u_1 \;\; u_2 \;\; u_3 \;\; u_4 \;\; v_1 \;\; v_2 \;\; v_3 \;\; v_4\}^{\mathrm{T}})$.

Eq. (1.42) must hold for all nodal points of the element. Therefore, from Eq. (1.42) we have

$$\mathbf{d} = \mathbf{G}\mathbf{c} \tag{1.43}$$

where

$$\mathbf{c} = \{c_1 \;\; c_2 \;\; c_3 \;\; c_4 \;\; c_5 \;\; c_6 \;\; c_7 \;\; c_8\}^{\mathrm{T}}, \quad \mathbf{G} = \begin{bmatrix} \mathbf{G}_1 & 0 \\ 0 & \mathbf{G}_1 \end{bmatrix} \tag{1.44}$$

and

$$\mathbf{G}_1 = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{1.45}$$

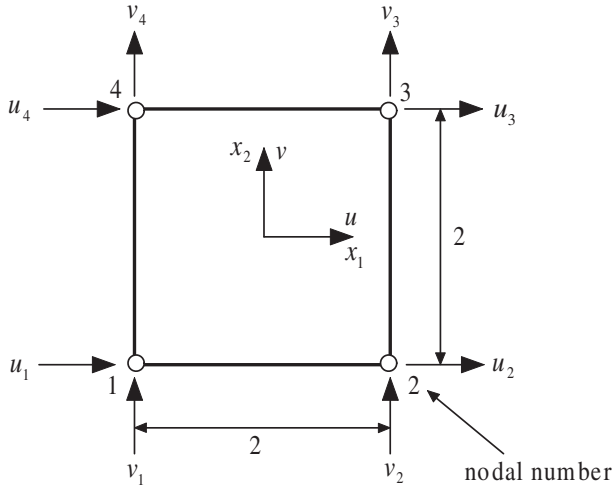


FIGURE 1.4
A typical two-dimensional 4-node element

Solving from Eq. (1.43) for **c** and substituting into Eq. (1.42), we obtain

$$\mathbf{u} = \begin{Bmatrix} u \\ v \end{Bmatrix} = \begin{bmatrix} \tilde{\mathbf{N}} & 0 \\ 0 & \tilde{\mathbf{N}} \end{bmatrix} \mathbf{d} \tag{1.46}$$

in which

$$\tilde{\mathbf{N}} = \frac{1}{4}\{(1-x_1)(1-x_2),\ (1+x_1)(1-x_2),\ (1+x_1)(1+x_2),\ (1-x_1)(1+x_2)\} \tag{1.47}$$

It is obvious that **c** can be related directly to **d** in the conventional element model.

Unlike the conventional FE model in which the same interpolation function is used to model both element domain and element boundary, two groups of independently assumed fields are required in the Trefftz finite formulation, as indicated in Section 1.2 (see also Figure 1.2). For the problem shown in Figure 1.1, Eq. (1.11) becomes

$$\mathbf{u} = \begin{Bmatrix} u \\ v \end{Bmatrix} = \begin{Bmatrix} \breve{u} \\ \breve{v} \end{Bmatrix} + \begin{bmatrix} \mathbf{N}_1 \\ \mathbf{N}_2 \end{bmatrix} \mathbf{c} = \breve{\mathbf{u}} + \mathbf{N}\mathbf{c} \tag{1.48}$$

where **N** is obtained by a suitably truncated T-complete system of plane stress problems, i.e., Ref. [61],

$$\mathbf{N} = [\mathbf{N}_1 \quad \mathbf{N}_2 \quad \cdots] = E \left[\begin{array}{ccc:cccc} 1 & 0 & x_2 & \mathrm{Re}A_1 & \mathrm{Re}B_1 & \mathrm{Re}C_1 & \mathrm{Re}D_1 & \cdots \\ 0 & 1 & -x_1 & \mathrm{Im}A_1 & \mathrm{Im}B_1 & \mathrm{Im}C_1 & \mathrm{Im}D_1 & \cdots \end{array}\right] \tag{1.49}$$

in which

$$A_k = (3-\mu)iz^k + (1+\mu)kiz\bar{z}^{k-1}, \quad B_k = (3-\mu)z^k - (1+\mu)kz\bar{z}^{k-1}$$
$$C_k = (1+\mu)i\bar{z}^k, \qquad D_k = -(1+\mu)\bar{z}^k \tag{1.50}$$

with $z = x_1 + ix_2$, $\bar{z} = x_1 - ix_2$ and $i = \sqrt{-1}$, and $\mathrm{Re}(f)$ and $\mathrm{Im}(f)$ standing for the real part and the imaginary part of f, respectively.

For the body force $\bar{b}_x = \text{const}$ and $\bar{b}_y = \text{const}$, the particular solution $\breve{\mathbf{u}}$ may be taken, for example, as

$$\begin{Bmatrix} \breve{u} \\ \breve{v} \end{Bmatrix} = -\frac{1+\mu}{E} \begin{Bmatrix} \bar{b}_x x_2^2 \\ \bar{b}_y x_1^2 \end{Bmatrix} \tag{1.51}$$

The frame field defined on the element boundary is required to conform across the inter-element boundary. For a particular side, say side 1-2, of the element in Figure 1.2, Eq. (1.46) becomes

$$\begin{Bmatrix} \tilde{u} \\ \tilde{v} \end{Bmatrix}_{12} = \mathbf{N}_{12}\mathbf{d}_{12} = \frac{1}{2} \begin{bmatrix} 1+\xi & 1-\xi & 0 & 0 \\ 0 & 0 & 1+\xi & 1-\xi \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ v_1 \\ v_2 \end{Bmatrix} \tag{1.52}$$

The frame field for the other sides can be obtained similarly.

Unlike the conventional FE model, it is not possible to provide an explicit relation between $\mathbf{c}$ and $\mathbf{d}$, as these two unknown vectors are independent at this stage. There are several ways to establish their relationship, such as the variational or the least square approach. If the variational approach is used, we may first write the corresponding variational functional in terms of $\mathbf{c}$ and $\mathbf{d}$ in the form [61]

$$\Psi_m = \frac{1}{2}\mathbf{c}^{\mathrm{T}}\mathbf{H}\mathbf{c} - \mathbf{c}^{\mathrm{T}}\mathbf{G}\mathbf{d} + \mathbf{c}^{\mathrm{T}}\mathbf{h} + \mathbf{d}^{\mathrm{T}}\mathbf{g} + \text{ terms without } \mathbf{c} \text{ or } \mathbf{d} \tag{1.53}$$

in which the matrices $\mathbf{H}$, $\mathbf{G}$ and the vectors $\mathbf{h}$, $\mathbf{g}$ are all defined in Eq. (1.25). The vanishing variation of Ψ_m with respect to $\mathbf{c}$ provides the relation between $\mathbf{c}$ and $\mathbf{d}$:

$$\frac{\partial \Psi_m}{\partial \mathbf{c}} = 0 \Rightarrow \mathbf{c} = \mathbf{H}^{-1}(\mathbf{G}\mathbf{d} - \mathbf{h}) \tag{1.54}$$

It can be seen from the above derivation that there is a fundamental difference between the two element models because of their different assumed fields. For each of the assumed fields, it is necessary to construct the related energy functionals in order to satisfy the special continuity conditions. We discuss these functionals below.

1.5.2 Variational functionals

The complementary energy functional (1.33) is generally used to formulate conventional hybrid FE. The summation of the functional (1.33) for all elements gives

$$\Psi = \sum_e \Psi_e = \frac{1}{2}\int_\Omega \Psi(\sigma_{ij})\,\mathrm{d}\Omega - \int_{\Gamma_u} \mathbf{t}\bar{\mathbf{v}}\mathrm{d}\Gamma \tag{1.55}$$

To use this complementary principle the stress components and corresponding surface tractions are expressed in terms of unknown parameters, and the stress functions must satisfy stress continuity between elements [i.e., Eq. (1.10)], the differential equations of equilibrium [i.e., Eq. (1.1)] and the stress boundary condition (1.8). Invoking the stationarity of the functional Ψ with respect to stress parameters, we obtain

$$\delta\Psi = \int_{\Omega} \delta\Psi(\sigma_{ij})\,\mathrm{d}\Omega - \int_{\Gamma_u} \delta\mathbf{t}\bar{\mathbf{v}}\mathrm{d}\Gamma = 0 \tag{1.56}$$

This leads to a hybrid finite element formulation.

In the Trefftz FEM, as mentioned in Section 1.2, the conditions (1.8) and (1.10) cannot be satisfied, *a priori*, due to the use of a truncated T-complete set of the governing equation as the shape function to model internal fields. In this case, the conditions (1.8) and (1.10) must be included in the variational functional. Thus, the Trefftz FE formulation can be obtained by extending the principle of stationarity of total complementary energy (1.56) to include the conditions of stress continuity between elements and the stress boundary conditions. In other words, these two conditions should be relaxed by adding some new terms into the standard functional (1.55). Inclusion of the conditions of stress continuity between elements can be implemented by adding the term

$$-\sum_{e}\int_{\Gamma_{Ie}} t_i\tilde{u}_i\mathrm{d}\Gamma \tag{1.57}$$

into the functional (1.55), while relaxation of the stress boundary conditions is completed by adding

$$\int_{\Gamma_t} (\bar{t}_i - t_i)\tilde{u}_i\mathrm{d}\Gamma \tag{1.58}$$

into the functional (1.55). Hence, the final results of the extended (or modified) functional are given in Eq. (1.32).

The above derivations clearly illustrate the relationship between the functionals used in the two FE models.

1.5.3 Assessment of the two techniques

An important consequence of replacing the domain integrals in the T-elements by the equivalent boundary integrals is that explicit knowledge of the distributions of the conforming shape functions is restricted to the element boundary. As a result, rather than deriving such distributions from $\tilde{\mathbf{N}} = \tilde{\mathbf{N}}(\mathbf{x})$ initially defined on Γ_e, the distribution, used here to interpolate the "frame function" (1.21), can now be defined directly on Γ_e, $\tilde{\mathbf{N}} = \tilde{\mathbf{N}}(\xi)$ in terms of the natural boundary coordinate ξ (Figure 1.2). In contrast to conventional models, this not only permits great liberty in element geometry (for example, allowing a polygonal element with an optional number of curved sides [61]), but also bypasses the difficulty encountered in the generation of conforming hierarchical shape functions for conventional C^1 conformity elements.

1.5.3.1 Geometry-induced singularities and stress concentrations

In the conventional FEM, the FE mesh should be strongly graded towards a singularity in order to achieve a faster algebraic convergence rate in the preasymptotic convergence range. Though the same approach may also be adopted in the alternative HT FEM, it is shown in Ref. [61] that special purpose functions, which in addition to satisfy the governing differential equations also fulfill the boundary conditions which are responsible for the singularity, can handle such singularities efficiently without mesh adjustment. In addition to various corner singularities (for which suitable functions have been presented by Williams [74] and successfully used by various authors [75]) such a library also includes special-purpose Trefftz functions for elements with circular [30] or elliptical holes [27], singular corners [25], etc. Many different problems with geometry-induced singularities can thus be handled without troublesome mesh refinement. Even more rewarding is the fact that the convergence rates remain the same as in the smooth case.

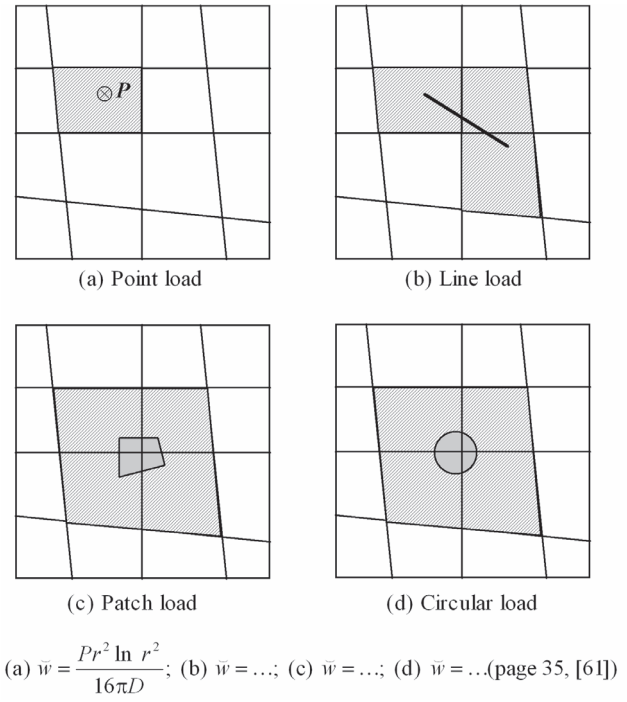


FIGURE 1.5
Handling of concentrated loads in Kirchhoff plate HT elements. The load term $\breve{w}$ can either extend over the whole element assembly or be confined to the closest elements (hatched elements)

1.5.3.2 Load-induced singularities and stress concentrations

Whereas in the conventional FEM concentrated loads must be handled in essentially the same manner as geometry-induced singularities, in the alternative HT FEM their effect is simply represented by the suitable load term $\breve{\mathbf{u}}$ in Eq. (1.11), for example (Figure 1.5) $\breve{w} = Pr^2 \ln r^2 \big/ (16\pi D)$ for an isolated load in HT plate elements. From the load term for an isolated load, load terms for other concentrated loadings of practical importance can effectively be obtained by integration [58]. Such load terms can either extend over the whole domain (global function) or be conveniently confined to the closest elements, automatically selected by the element subroutine (semi-global function). This feature is one of the most important advantages of the HT approach, since concentrated loads are accounted for with comparable accuracy to smooth ones (indeed, the accuracy depends very little on the load condition) and their location can be arbitrarily varied without any need for mesh adjustment, the mesh being totally load-independent.

1.5.3.3 Concluding remarks

The comparison above and further numerical experiments reveal that the advantage of the conventional FEM is that the underlying theoretical concept is largely known and a deep mathematical analysis is available for its convergence properties [59]. This approach has mostly been used for the solution of various C^0 conformity problems, whereas C^1 conformity shape functions are more difficult to generate [59]. In contrast, very general polygonal HT elements with an optional number of curved sides can be generated with equal ease for both second- and fourth-order equation governed problems. Another asset of the HT FEM is that load and/or geometry-induced singularities can be handled very efficiently without mesh refinement, and stress intensity factors of corner singularity problems are obtained directly in the process of the FE calculation without an elaborate extraction process. The major features and differences between conventional FEM and HT FEM are listed in Figure 1.6.

1.6 Comparison of T-elements with boundary elements

In Section 1.5 the main distinctions between T-elements and conventional FE and some relative advantages of T-elements were systematically shown. The purpose of this section is to provide a brief comparison between T-elements and boundary elements by way of the variational approach.

For the sake of simplicity we consider a two-dimensional Laplace equation:

$$\nabla^2 u = 0 \qquad \text{in } \Omega \tag{1.59}$$

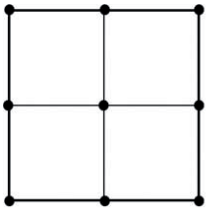
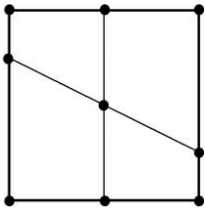
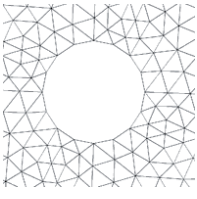
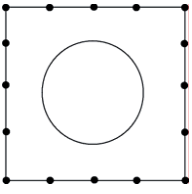
Conventional FEM	HT FEM
It is complex to generate curved, polygonal, or C^1-conformity elements as the same interpolation function is used to model both element domain and element boundary	It is easy to generate curved, polygonal, or C^1-conformity elements as two groups of independently assumed fields are used to model element domain and element boundary separately, and integrations are confined to the boundary only
It is not very sensitive to mesh distortion	It is insensitive to mesh distortion
 Regular mesh	 Irregular mesh
Mesh adjustment is required for an element with local effects	Mesh adjustment is not necessary as special purpose functions can be used as trial functions to efficiently handle local effects without mesh adjustment.
	
The stiffness matrix is symmetric and sparse	The stiffness matrix is symmetric and sparse
Much commercial software, such as ABAQUS and ANSYS, is available	No commercial software is available

FIGURE 1.6
Features of conventional FEM and HT FEM

with boundary conditions

$$u = \bar{u} \qquad \text{on } \Gamma_u \tag{1.60}$$

and

$$q_n = \frac{\partial u}{\partial n} = \bar{q} \qquad \text{on } \Gamma_q \tag{1.61}$$

where $\Gamma = \Gamma_u \cup \Gamma_q$ is the boundary of the domain Ω and n is the unit normal to the boundary.

1.6.1 Boundary elements

The standard boundary element formulation for the boundary value problem (BVP) (1.59) - (1.61) can most easily be obtained by first considering the following variational functional:

$$\Pi_m = \Pi + \int_{\Gamma_u} (\bar{u} - u)\, q_n \mathrm{d}\Gamma \tag{1.62}$$

where

$$\Pi = \int_{\Omega} U \mathrm{d}\Omega - \int_{\Gamma_q} \bar{q}_n u \mathrm{d}\Gamma \tag{1.63}$$

$$U = \frac{1}{2}\left(u_{,x_1}^2 + u_{,x_2}^2\right) \tag{1.64}$$

The vanishing variation of Π_m yields

$$\delta\Pi_m = \delta\Pi + \int_{\Gamma_u} \left[(\bar{u} - u)\, \delta q_n - q_n \delta u\right] \mathrm{d}\Gamma = 0 \tag{1.65}$$

where δu can be chosen arbitrarily, leading to different numerical techniques. The boundary integral equation (BIE) is obtained when δu is chosen as the fundamental solution of Eq. (1.59), that is [76],

$$\delta u = u^* \varepsilon = \frac{\varepsilon}{2\pi} \ln\left(\frac{1}{r}\right) \tag{1.66}$$

where ε is an infinitesimal and r is the distance from source point to observation point [77].

We approximate u and q_n through the shape function, say $\mathbf{F}$, of the boundary discretisation,

$$u = \mathbf{F}^{\mathrm{T}}\mathbf{u}, \quad q_n = \mathbf{F}^{\mathrm{T}}\mathbf{q} \tag{1.67}$$

where $\mathbf{u}$ and $\mathbf{q}$ are the nodal vectors of u and q_n, respectively. By substitution of Eqs. (1.66) and (1.67) into Eq. (1.65), we obtain

$$\mathbf{H}\mathbf{u} = \mathbf{G}\mathbf{q} \tag{1.68}$$

where

$$H_{ij} = \int_{\Gamma_j} q^* F_i \mathrm{d}\Gamma, \quad G_{ij} = \int_{\Gamma_j} u^* F_i \mathrm{d}\Gamma \tag{1.69}$$

with $q^* = \partial u^*/\partial n$.

When the boundary conditions are introduced in Eq. (1.68), a system of equations is obtained in which the unknowns are nodal values of both u and q_n.

1.6.2 T-elements

The T-element formulation for the problem given in Eqs. (1.59) - (1.61) can be derived by the following variational functional [36]:

$$\Psi_m = \Psi + \int_{\Gamma_q} (\bar{q}_n - q_n)\,\tilde{u}\mathrm{d}\Gamma - \sum_e \int_{\Gamma_{Ie}} q_n \tilde{u}\mathrm{d}\Gamma \tag{1.70}$$

where

$$\Psi = \int_\Omega U\mathrm{d}\Omega - \int_{\Gamma_u} q_n \bar{u}\mathrm{d}\Gamma \tag{1.71}$$

$$U = \frac{1}{2}\left(q_{x_1}^2 + q_{x_2}^2\right) \tag{1.72}$$

We approximate u, q_n and $\tilde{u}$ through the appropriate functions as [48],

$$u = \breve{u} + \mathbf{N}\mathbf{c},\ \ q_n = \breve{q}_n + \mathbf{Q}\mathbf{c},\ \ \tilde{u} = \tilde{\mathbf{N}}\mathbf{d} \tag{1.73}$$

where $\mathbf{N}$ and $\mathbf{Q}$ are regular functions obtained by a suitable truncated T-complete system of solutions for Eq. (1.59), while $\tilde{\mathbf{N}}$ is the usual shape function. Minimization of the functional (1.70) leads to

$$\mathbf{K}\mathbf{d} = \mathbf{P} \tag{1.74}$$

where

$$\mathbf{K} = \mathbf{G}^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{G},\quad \mathbf{P} = \mathbf{G}^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{h} + \mathbf{g} \tag{1.75}$$

$$\mathbf{G} = \int_{\Gamma_e} \mathbf{Q}^{\mathrm{T}}\tilde{\mathbf{N}}\mathrm{d}\Gamma,\quad \mathbf{H} = \int_{\Gamma_e} \mathbf{Q}^{\mathrm{T}}\mathbf{N}\mathrm{d}\Gamma \tag{1.76}$$

$$\mathbf{h} = \frac{1}{2}\int_{\Omega_e} \mathbf{N}^{\mathrm{T}}\mathbf{b}\mathrm{d}\Omega + \frac{1}{2}\int_{\Gamma_e} \left(\mathbf{Q}^{\mathrm{T}}\breve{\mathbf{u}} + \mathbf{N}^{\mathrm{T}}\breve{\mathbf{t}}\right)\mathrm{d}\Gamma \tag{1.77}$$

$$\mathbf{g} = \int_{\Gamma_{te}} \tilde{\mathbf{N}}^{\mathrm{T}}\bar{\mathbf{t}}\mathrm{d}\Gamma - \int_{\Gamma_e} \tilde{\mathbf{N}}^{\mathrm{T}}\breve{\mathbf{t}}\mathrm{d}\Gamma \tag{1.78}$$

It has been proved that the matrix $\mathbf{H}$ is symmetric [8]. Thus the element matrix $\mathbf{K}$ is symmetric, which can be seen from Eq. (1.75).

1.6.3 Assessment of the two numerical models

It can be seen from the above analysis that there is an essential distinction between T-elements and conventional boundary elements (BEs). This is attributable to the different variational principles and trial functions employed. The major difference between them is the calculation of the coefficient matrix and the trial functions used; namely, in the T-element approach, the coefficient matrix is evaluated through use of

a modified complementary energy principle, Π_m, and the regular T-complete functions, whereas calculation of the coefficient matrix in BEM is based on a modified potential principle, Γ_m, and the fundamental solutions which are either singular or hyper-singular. As a consequence, when integration is carried out in T-elements it is always simpler and more economical than in the standard BEM. In particular, computation of the coefficient matrix $\mathbf{K}$ (Eq. (1.75)) in T-elements is quite simple, since all kernel functions in Eq. (1.76) are regular. In contrast, standard BEM, where the kernel of the integrals in Eq. (1.69) is either singular or hyper-singular, requires special and expensive integration schemes to calculate the integrals accurately. Another important advantage of T-elements over standard BEM is that no boundary effect (inevitable in BEM calculation as the integral point approaches the boundary) is present, as no fundamental solutions of the differential equations are used. The T-element model has some additional advantages besides the above-mentioned properties. For example, the T-element form makes it possible to eliminate some of the well-known drawbacks of the BEM [8]: (a) the fully populated non-symmetric coefficient matrix of the resulting equation system is replaced by a symmetric banded form; (b) no complications arise if the governing differential equations are non-homogeneous; (c) there is no need for time-consuming boundary integration when the results are to be evaluated inside the domain. A comparison of major features of conventional BEM and HT FEM is provided in Figure 1.7.

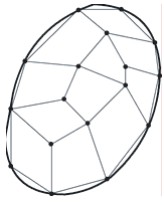
Conventional BEM	HT FEM
Calculation of the coefficient matrices **H** and **G** (Eq (1.69)) in BEM requires special and expensive integration schemes as all kernel functions in Eq (1.69) are either singular or hyper-singular	Computation of the coefficient matrix **K** (Eq (1.75)) in T-elements is quite simple, since all kernel functions in Eq (1.76) are regular.
The coefficient matrices **H** and **G** in Eq (1.69) are, in general, non-symmetric	The stiffness matrix **K** in Eq (1.75) is symmetric and sparse, like FEM
BEM requires only surface discretisation, not full-domain discretisation. As a result, the dimensionality of the problem is reduced by one and the system of equations is much smaller in size than that encountered in HT FEM	Full-domain discretisation is required in HT FEM. However, only the integral along the element boundary is used in the computation, so the dimensionality of the problem is reduced by one at the elemental level.
	
The fundamental solution in BEM is usually defined in the whole solution domain, and it is therefore inconvenient to analyse problems with different materials or heterogeneous materials.	As the material properties are defined in every element in HT-FEM, it is easy to apply to problems with different materials and heterogeneous materials.
It is time-consuming to calculate boundary and domain integrations when the results are to be evaluated inside the domain.	There is no need for time-consuming boundary integration when the results are to be evaluated inside the domain.
No commercial software is available	No commercial software is available

FIGURE 1.7
Features of conventional BEM and HT FEM

References

[1] Jirousek J, Leon N (1977), A powerful finite element for plate bending. *Comput Method Appl M*, **12**: 77-96.

[2] Herrera I (1977), General variational principles applicable to the hybrid element method. *Proc Natl Acad Sci USA*, **74**: 2595-2597.

[3] Herrera I (1980), Boundary methods. A criterion for completeness. *Proc Natl Acad Sci USA*, **77**: 4395-4398.

[4] Herrera I, Ewing RE, Celia MA, Russell T (1993), Eulerian-Lagrangian localized adjoint method: The theoretical framework. *Numer Meth Partial Dif Equa*, **9**: 431-457.

[5] Herrera I, Sabina FJ (1978), Connectivity as an alternative to boundary integral equations: construction of bases. *Proc Natl Acad Sci USA*, **75**: 2059-2063.

[6] Trefftz E (1926), Ein Gegenstück zum Ritzschen Verfahren,*in Proc 2nd Int Cong Appl Mech*, Zurich, 131-137.

[7] Jirousek J, N'Diaye M (1990), Solution of orthotropic plates based on *p*- extension of the hybrid-Trefftz finite element model, *Comput Struct* **34**: 51-62.

[8] Jirousek J, Wroblewski A (1996), T-elements: State of the art and future trends. *Arch Comput Method Eng*, **3**: 323-434.

[9] Jirousek J (1978), Basis for development of large finite elements locally satisfying all field equations. *Comput Meth Appl Mech Eng*, **14**: 65-92.

[10] Diab SA (2001), The natural boundary conditions as a variational basis for finite element methods. *Comput Assis Mech Eng Sci*, **8**: 213-226.

[11] Freitas JAT, Ji ZY (1996), Hybrid-Trefftz equilibrium model for crack problems. *Int J Numer Meth Eng*, **39**: 569-584.

[12] Herrera I (2001), On Jirousek method and its generalization. *Comput Assis Mech Eng Sci*, **8**: 325-342.

[13] Jirousek J, Venkatesh A (1989), A simple stress error estimator for hybrid-Trefftz *p*-version elements. *Int J Numer Meth Eng*, **28**: 211-236.

[14] Kita E, Kamiya N, Ikeda Y (1996), New boundary-type scheme for sensitivity analysis using Trefftz formulation. *Finite Elem Anal Des*, **21**: 301-317.

[15] Lifits S, Reutskiy S, Tirozzi B (1997), Trefftz spectral method for initial-boundary problems. *Comput Assis Mech Eng Sci*, **4**: 549-565.

[16] Markiewicz M, Mahrenholtz O (1997), Combined time-stepping and Trefftz approach for nonlinear wave-structure interaction. *Comput Assis Mech Eng Sci*, **4**: 567-586.

[17] Maunder EAW, Almeida JPM (1997), Hybrid-equilibrium elements with control of spurious kinematic modes. *Comput Assis Mech Eng Sci*, **4**: 587-605.

[18] Melenk JM, Babuska I (1997), Approximation with harmonic and generalized harmonic polynomials in the partition of unity method. *Comput Assis Mech Eng Sci*, **4**: 607-632.

[19] Ohnimus S, Rüter M, Stein E (2001), General aspects of Trefftz method and relations to error estimation of finite element approximations. *Comput Assis Mech Eng Sci*, **8**: 425-437.

[20] Peters K, Wagner W (1994), New boundary-type finite element for 2-D and 3-D elastic structures. *Int J Numer Meth Eng*, **37**: 1009-1025.

[21] Piltner R (1997), On the systematic construction of trial functions for hybrid Trefftz shell elements, *Comput Assis Mech Eng Sci*, **4**: 633-644.

[22] Qin QH (1994), Hybrid Trefftz finite element approach for plate bending on an elastic foundation. *Appl Math Model*, **18**: 334-339.

[23] Zheng XP, Yao ZH (1995), Some applications of the Trefftz method in linear elliptic boundary-value problems. *Adv Eng Softw*, **24**: 133-145.

[24] Jirousek J, Teodorescu P (1982), Large finite elements method for the solution of problems in the theory of elasticity. *Comput Struct*, **15**: 575-587.

[25] Jirousek J, Venkatesh A (1992), Hybrid-Trefftz plane elasticity elements with *p*-method capabilities. *Int J Numer Meth Eng*, **35**: 1443-1472.

[26] Leitao VMA (2001), Comparison of Galerkin and collocation Trefftz formulations for plane elasticity. *Comput Assis Mech Eng Sci*, **8**: 397-407.

[27] Piltner R (1985), Special finite elements with holes and internal cracks. *Int J Numer Meth Eng*, **21**: 1471-1485.

[28] Stein E, Peters K (1991), A new boundary-type finite element for 2D and 3D elastic solids. In: *The Finite Element Method in the 1990s, a book dedicated to OC Zienkiewicz*, E Onate, J Periaux and A Samuelson (eds), Springer, Berlin, p35-48.

[29] Jirousek J (1987), Hybrid-Trefftz plate bending elements with *p*-method capabilities. *Int J Numer Meth Eng*, **24**: 1367-1393.

[30] Jirousek J, Guex L (1986), The hybrid-Trefftz finite element model and its application to plate bending. *Int J Numer Meth Eng*, **23**: 651-693.

[31] Jirousek J, Szybinski B, Zielinski AP (1997), Study of a family of hybrid-Trefftz folded plate p-version elements. *Comput Assis Mech Eng Sci*, **4**: 453-476.

[32] Jin FS, Qin QH (1995), A variational principle and hybrid Trefftz finite element for the analysis of Reissner plates. *Comput Struct*, **56**: 697-701.

[33] Jirousek J, Wroblewski A, Qin QH, He XQ (1995), A family of quadrilateral hybrid Trefftz *p*-element for thick plate analysis. *Comput Method Appl Mech Eng*, **127**: 315-344.

[34] Jirousek J, Wroblewski A, Szybinski B (1995), A new 12 DOF quadrilateral element for analysis of thick and thin plates. *Int J Numer Meth Eng*, **38**: 2619-2638.

[35] Maunder EAW (2001), A Trefftz patch recovery method for smooth stress resultants and applications to Reissner-Mindlin equilibrium plate models. *Comput Assis Mech Eng Sci*, **8**: 409-424.

[36] Qin QH (1995), Hybrid Trefftz finite element method for Reissner plates on an elastic foundation. *Comput Method Appl Mech Eng*, **122**: 379-392.

[37] Petrolito J (1990), Hybrid-Trefftz quadrilateral elements for thick plate analysis. *Comput Method Appl Mech Eng*, **78**: 331-351.

[38] Petrolito J (1996), Triangular thick plate elements based on a Hybrid-Trefftz approach. *Comput Struct*, **60**: 883-894.

[39] Piltner R (1992), A quadrilateral hybrid-Trefftz plate bending element for the inclusion of warping based on a three-dimensional plate formulation, *Int J Numer Meth Eng*. **33**: 387-408.

[40] Piltner R (1989), On the representation of three-dimensional elasticity solutions with the aid of complex value functions, *J Elasticity* **22**: 45-55.

[41] Wroblewski A, Zielinski AP, Jirousek J (1992), Hybrid-Trefftz p-element for 3-D axisymmetric problems of elasticity, In: *Numerical methods in Engineering'92, Proc First Europ Conf On Numer Meth in Eng*, C Hirsch, OC Zienkiewicz, and E Oñate (eds), Brussels, Elsevier, 803-810.

[42] Jirousek J, Stojek M (1995), Numerical assessment of a new T-element approach, *Comput Struct* **57**: 367-378.

[43] Zielinski AP, Zienkiewicz OC (1985), Generalized finite element analysis with T-complete boundary solution functions, *Int J Numer Meth Eng*. **21**: 509-528.

[44] Vörös GM, Jirousek J (1991), Application of the hybrid-Trefftz finite element model to thin shell analysis. *Proc Europ Conf on new Advances in Comp Struc Mech*, P Ladeveze and OC Zienkiewicz (eds), Giens, France, Elsevier, p547-554.

[45] Freitas JAT (1997), Hybrid-Trefftz displacement and stress elements for elastodynamic analysis in the frequency domain. *Comput Assis Mech Eng Sci* **4**: 345-368.

[46] Jirousek J (1997), A T-element approach to forced vibration analysis and its application to thin plates. *LSC Internal Report 97/03*, Swiss Federal Institute of Technology, Lausanne.

[47] Qin QH (1996), Transient plate bending analysis by hybrid Trefftz element approach. *Commun Numer Meth Eng*, **12**: 609-616.

[48] Jirousek J, Qin QH (1996), Application of Hybrid-Trefftz element approach to transient heat conduction analysis. *Comput Struct*, **58**: 195-201.

[49] Qin QH (1995), Postbuckling analysis of thin plates by a hybrid Trefftz finite element method. *Comput Method Appl Mech Eng*, **128**: 123-136.

[50] Qin QH (1996), Nonlinear analysis of thick plates by HT FE approach. *Comput Struct*, **61**: 271-281.

[51] Qin QH (1997), Postbuckling analysis of thin plates on an elastic foundation by HT FE approach. *Appl Math Model*, **21**: 547-556.

[52] Qin QH, Diao S (1996), Nonlinear analysis of thick plates on an elastic foundation by HT FE with *p*-extension capabilities. *Int J Solids Struct*, **33**: 4583-4604.

[53] Qin QH (2005), Formulation of hybrid Trefftz finite element method for elastoplasticity. *Appl Math Model*, **29**: 235-252.

[54] Qin QH (2005), Trefftz plane elements of elastoplasticity with *p*-extension capabilities. *J Mech Eng*, **56**: 40-59.

[55] Zielinski AP (1988), Trefftz method: elastic and elastoplastic problems. *Comput Method Appl M*, **69**: 185-204.

[56] Qin QH (2003), Variational formulations for TFEM of piezoelectricity. *Int J Solids Struct*, **40**: 6335-6346.

[57] Qin QH (2003), Solving anti-plane problems of piezoelectric materials by the Trefftz finite element approach. *Comput Mech*, **31**: 461-468.

[58] Venkatesh A, Jirousek J (1995), Accurate representation of local effect due to concentrated and discontinuous loads in hybrid-Trefftz plate bending elements. *Comput Struct*, **57**: 863-870.

[59] Jirousek J, Venkatesh A, Zielinski AP, and Rabemanantsoo H (1993), Comparative study of *p*-extensions based on conventional assumed displacement and hybrid-Trefftz FE models. *Comput Struct*, **46**: 261-278.

[60] Huang HT, Li ZC (2003), Global superconvergence of Adini's elements coupled with Trefftz method for singular problems. *Eng Anal Bound Elem*, **27**: 227-240.

[61] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*, Southampton: WIT Press.

[62] Herrera I (1997), Trefftz-Herrera method. *Comput Assis Mech Eng Sci*, **4**: 369-382.

[63] Jirousek J (1991), New trends in HT-p element approach, *The Finite Element Method in the 1990s*, E Onate, J Periaux, and A Samuelson (eds), Springer, Berlin.

[64] Jirousek J, Wroblewski A (1995), T-elements: a finite element approach with advantages of boundary solution methods. *Adv Eng Softw*: **24**: 71-88.

[65] Jirousek J, Zielinski AP (1997), Survey of Trefftz-type element formulations. *Comput Struct*, **63**: 225-242.

[66] Kita E, Kamiya N (1995), Trefftz method: an overview. *Adv Eng Softw*: **24**: 3-12.

[67] Qin QH (1993), A brief introduction and prospect to hybrid -Trefftz finite elements. *Mech Pract*, **15**: 20-22 (in Chinese).

[68] Qin QH (1998), The study and prospect of Hybrid-Trefftz finite element method. *Adv Mech*, **28**: 71-80 (in Chinese).

[69] Qin QH (2005), Trefftz finite element method and its applications. *Appl Mech Rev*, **58**: 316-337.

[70] Zienkiewicz OC (1997), Trefftz type approximation and the generalised finite element method—history and development. *Comput Assis Mech Eng Sci*, **4**: 305-316.

[71] Herrera I (2000), Trefftz method: A general theory. *Numer Meth Partial Diff Equa*, **16**: 561-580.

[72] Washizu K (1982) *Variational Methods in Elasticity and Plasticity*. Oxford: Pergamon Press.

[73] Qin QH (2004), Dual variational formulation for Trefftz finite element method of Elastic materials. *Mech Res Commun*, **31**: 321-330.

[74] Williams ML (1952), Stress singularities resulting from various boundary conditions in angular corners of plates in tension. *J Appl Mech*, **19**: 526-528.

[75] Lin KY, Tong P (1980), Singular finite elements for the fracture analysis of V-notched plate. *Int J Numer Meth Eng*, **15**: 1343-1354.

[76] He XQ, Qin QH (1993), Nonlinear analysis of Reissner's plate by the variational approaches and boundary element methods. *Appl Math Model*, **17**: 149-155.

[77] Brebbia CA, Dominguez J (1992), *Boundary Elements: An introductory course*. Southampton: Computational Mechanics Publication/McGraw-Hill Inc.

2

Foundation of MATLAB programming

2.1 Introduction

MATrix**LAB**oratory, MATLAB for short, was invented in the late 1970s by Cleve Moler and further developed through The MathWorks Inc. MATLAB provides a high-level language and development tools that let you quickly develop and analyse your algorithms and applications.

As an interpretive programming language, the most attractive feature of MATLAB is that it incorporates a broad range of utility commands and intrinsic functions, such as mathematical functions for linear algebra and numerical integration, 2-D and 3-D graphics functions for visualising data, and functions for integrating MATLAB-based algorithms with external applications and languages, as compared to traditional upper-level compiled programming languages such as FORTRAN, C, and C++. With the MATLAB language, we can program and develop algorithms faster than with traditional languages because we do not need to perform low-level administrative tasks such as declaring variables, specifying data types, and allocating memory. It should be mentioned, however, that MATLAB run time may be greater than that of compiled programming languages like FORTRAN and C, due to the fact that each row of code is first interpreted by the MATLAB command interpreter and then executed. In this chapter, some basic elements of MATLAB programming used in the following chapters are reviewed; the presentation follows the results given in Refs. [1 - 5].

2.2 Basic data types in MATLAB

2.2.1 Array and variable

In MATLAB, the fundamental unit of data is the array or matrix. An array is a collection of data values organised into rows and columns, and defined by a user-specified name. Specially, a scalar is treated as an array with only one row and one column.

A MATLAB variable is a region of memory containing an array which does not use a type definition or a dimension statement. The contents of the array can be used

or modified at any time by including its name in an appropriate MATLAB command. Regarding variable names, there are two important aspects to be noted:

- Variable names must begin with an alphanumeric letter. Following that, any number of letters, digits and underscores can be added, but only the first 31 characters are retained. For example, variables `ntype`, `Lnods`, `coord` and `coord_n` are legal, whereas `3nnode` is illegal in MATLAB.

- MATLAB variable names are case-sensitive. Thus `coord` and `Coord` are different variables.

2.2.2 Types of variables

The most common types of MATLAB variables used in the following chapters are double-precision (or double for short) and characters. MATLAB constructs the `double` data type according to IEEE Standard 754 for double precision. Any value stored as a double requires 64 bits. A double variable can store a real, imaginary or complex number in the range of 10^{-308} to 10^{308} and with 15 to 16 significant decimal digits of accuracy. A variable of type `double` is automatically created whenever a numerical value is assigned to a variable name. For example, a complex variable with double-type real and imaginary components can be created

```
complex_var = 3 + 4i;
```

On the other hand, `character` variables consist of characters or arrays of 16-bit values, each representing a single character. Arrays of this type are used to store character strings. They are automatically created whenever a single character or a character string is assigned to a variable name. For example, a character string named as `string_var` with 5 characters is created

```
string_var = 'china';
```

2.2.3 Built-in variables

MATLAB has its own built-in variables* including `ans`, `pi`, `eps`, `flops`, `inf`, `Nan` or `nan`, `i`, `j`, `nargin`, `nargout`, `realmin`, `realmax`, which are partly listed in Table 2.1 for our programming application in the following chapter.

*These built-in variables are stored in the computer and can be overwritten or modified by users. However, in programming it is suggested not to change them.

TABLE 2.1
Some built-in variables in MATLAB

Function	Value
`pi`	3.1415926535897...
`i,j`	Imaginary unit ($\sqrt{-1}$)
`inf`	Infinity
`Nan`	Not a number, an invalid numerical value
`eps`	Floating-point relative accuracy. For example, eps = 2.2204e-16
`ans`	A special value is designed to store the result of an expression, if the result is not assigned to another variable
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number

2.3 Matrix manipulations

Since the basic unit in MATLAB is a matrix, any single number, character, or array can be viewed as a one by one or n by one (or m by n) matrix, respectively. In this section, we will review some rules for matrix manipulation in MATLAB.

2.3.1 Initialising matrix variable

In general, a matrix should be initialised before it is used. In MATLAB there are some built-in functions which can be used to perform initialisation of a matrix variable. Among these, the function `zeros` is a popular one used to create a zero matrix of any desired size. For example, the command

```
a=zeros(m,n);
```

can be used to create an `m` by `n` matrix `a`, whose elements are all zero.

It is useful to increase the efficiency of m-file functions in MATLAB programming because this command can pre-allocate memory for a matrix variable before assigning values to its elements, instead of increasing the size of the matrix variable step by step, which is time consuming.

2.3.2 Matrix indexing

Flexible matrix variable indexing provided by MATLAB can increase the efficiency of m-file functions. For example, for a typical two-dimensional matrix variable A

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

the usual operations for matrix indexing are shown in Table 2.2.

TABLE 2.2
Matrix indexing

Command	Result
`size(A)`	[m, n]
`A(i,j)`	a_{ij}
`A(p:q,j)`	$\begin{bmatrix} a_{pj} \\ \vdots \\ a_{qj} \end{bmatrix}$
`A(i,p:q)`	$\begin{bmatrix} a_{ip} & \cdots & a_{iq} \end{bmatrix}$
`A(i:j,p:q)`	$\begin{bmatrix} a_{ip} & \cdots & a_{iq} \\ \vdots & \ddots & \vdots \\ a_{jp} & \cdots & a_{jq} \end{bmatrix}$

2.3.3 Common array and matrix operations

In general, MATLAB supports two types of operation between arrays, known as array operation and matrix operation. Array operation is the operation performed between matrices on an element-by-element basis, that is, the operation is performed on corresponding elements in the two matrices. In contrast, matrix operations follow the normal rules of linear algebra such as matrix multiplication[†].

MATLAB uses a special symbol to distinguish array operations from matrix operations. Usually, MATLAB uses a dot before the symbol to indicate an array operation and produce element-by-element results, for example, "`.*`". A list of common array and matrix operations is given in Table 2.3.

[†]In a matrix, individual elements within a row are separated by blank spaces or commas, and the rows themselves are generally separated by semicolons. A semicolon can also be used to control the result displayed in the Command window by adding or discarding it at the end of each expression.

TABLE 2.3
Common array and matrix operations

	Array operation	**Matrix operation**
Addition	a±b	a±b
Subtraction	a-b	a-b
Multiplication	a.*b	a*b
Right division	a./b	a/b (=a*inv(b))
Left division	a.\b	a\b (=inv(a)*b)
Exponentiation	a.^b	a^b

As an illustration, the following expressions firstly initialise two matrices `A` and `b`, and then some array operations and matrix operations are compared, including some legal and illegal operations [6].

```
>> A=[2, -1, 3, 0; 1, 5, -2, 4; 2, 0, 3, -2; 1, 2, 3, 4]

A =

     2    -1     3     0
     1     5    -2     4
     2     0     3    -2
     1     2     3     4

>> b=[3; 1; -2; 2]

b =

     3
     1
    -2
     2

>> A.^2

ans =

     4     1     9     0
     1    25     4    16
     4     0     9     4
     1     4     9    16
```

```
>> A^2

ans =

     9    -7    17   -10
     7    32    -1    40
     8    -6     9   -14
    14    17    20    18

>> x=A/b
??? Error using ==> mrdivide Matrix dimensions must agree.

>> x=A\b

x =

    1.9259
   -1.8148
   -0.8889
    1.5926

>> x=inv(A)*b

x =

    1.9259
   -1.8148
   -0.8889
    1.5926

>> x=A.\b
??? Error using ==> ldivide Matrix dimensions must agree.

>> x=A./2

x =

    1.0000   -0.5000    1.5000         0
    0.5000    2.5000   -1.0000    2.0000
    1.0000         0    1.5000   -1.0000
    0.5000    1.0000    1.5000    2.0000

>> x=2.\A
```

```
x =

    1.0000   -0.5000    1.5000         0
    0.5000    2.5000   -1.0000    2.0000
    1.0000         0    1.5000   -1.0000
    0.5000    1.0000    1.5000    2.0000
```

From above procedure, we can see that to make array and matrix operations successfully, the requirement of dimension should be paid attention to carefully. Detailed explanations can be found in books [1 - 5].

2.3.4 Hierarchy of operations

In the above procedure, only one arithmetic operation is involved. However, in practice, two or more than two arithmetic operations on arrays and matrices are often combined into a single expression. In this case, to make the evaluation of the expressions unambiguous, MATLAB has established a series of rules governing the hierarchy, in which operations are evaluated in proper order within an expression. Table 2.4 below displays the hierarchy of arithmetic operations, and we can observe that the rules generally follow the normal rules of algebra.

TABLE 2.4
Hierarchy of arithmetic operations

Hierarchy	Arithmetic operator	Operation
1	()	The contents of all parentheses are evaluated, starting from the innermost parentheses and working outward
2	^	All exponentials are evaluated, working from left to right
3	*, /	All multiplications and divisions are evaluated, working from left to right
4	+, -	All additions and subtractions are evaluated, working from left to right

The following is a simple example to demonstrate the order of arithmetic operations:

```
>> (3+2)^2+2^3*4
```

```
ans =

     57
```

2.4 Control structures

During programming, control structures are usually necessary to control the flow of commands. MATLAB supplies programmers writing codes with two categories of control statements: branches and loops. Branches select specific sections of the code (called blocks) to execute, and loops cause specific sections of the code to be repeated according to conditions or expressions defined by programmers, which are usually related to rational and logical operators. In this section, the relational and logical operators are introduced and then two branches (the `if` construct and the `switch` construct) and two loops (the `for` loop and the `while` loop) are reviewed.

TABLE 2.5
Common relational and logical operators in MATLAB

Type	Operator	Operation
Rational	==	Equal to
	~=	Not equal to
	>	Greater than
	>=	Greater than or equal to
	<	Less than
	<=	Less than or equal to
Logic	&	AND
	|	OR
	~	NOT

2.4.1 Relational and logical operators

Relational operators are operators with two numerical or string operands that yield either a true (1) or a false (0) result, depending on the relationship between the two operands, while the logical operators in MATLAB are operators with one or two operands that yield a logical result, which are used to combine logical expressions with "AND" or "OR", or to change a logical value with "NOT". The commonly used relational and logical operators are shown in Table 2.5.

2.4.2 The `if` construct

The syntax of the `if` construct has a general form as

```
if expression 1
    Block 1;
elseif expression 2
    Block 2;
.
.
else
    Block n;
end
```

where the control expressions control the operation of the `if` construct. If `expression 1` is true, then the program executes Block 1 and skips to the first executable statement following the `end`. Otherwise, the program checks for the status of `expression 2`. If `expression 2` is true, then the program executes Block 2, and skips to the first executable statement following the `end`. If all control expressions are false, then the program executes the statements in Block 3 associated with the `else` clause.

For example, the following `if` structure gives the main function of selection:

```
if n>0
    x=2;
elseif n<0
    x=-3;
else
    x=0;
end
```

2.4.3 The `switch` construct

The `switch` construct is another form of branching construct. The general form of a `switch` construct is

```
switch expression
    case value 1
        Block 1;
    case value 2
        Block 2;
    .
    .
    otherwise
        Block n;
end
```

The `switch` construct permits a programmer to select a particular code block to execute if the value of expression, which may be a single integer, character, or logical expression, is equal to a certain value behind the `case` clause.

For instance,

```
switch n
    case 1
        x=pi/2;
    case 2
        x=pi/3;
    case 3
        x=pi/6;
    otherwise
        disp('Invalid n will cause wrong result');
end
```

2.4.4 The `for` loop

The `for` loop has the form

```
for index=initial value:increment:last value
    Block;
end
```

The block of commands between the `for` and `end` are executed for all values in the range of expression, that is to say, the number of times of execution is specified. In addition, it is worth pointing out that the `for-end` loop can be nested. For example, the following double `for` loop is designed to perform the summation and conversion of results:

```
for k=1:5
    temp=0;
    for m=1:3
        temp=temp+exp(2*m);
    end
    y(k)=temp;
end
```

2.4.5 The `while` loop

The `while` loop is often used when an iteration is repeated until some termination criterion is met. The syntax of the `while` loop is

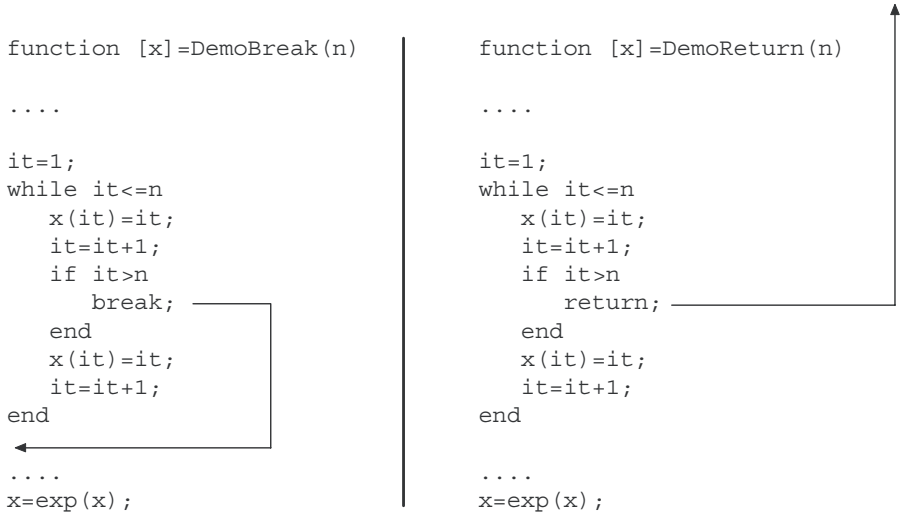
```
while expression
    Block;
end
```

As long as the expression is true, the block continues to be executed, so when using a `while` loop a programmer must think carefully to confirm that the loop is not repeated infinitely. It is always a good idea to put a limit on the number of iterations to be performed by a `while` loop. For example, in the following `while` loop, the variable `maxite` is used to control the iteration.

```
nt=3;
while nt<7
    RZ(nt)=10*sin(k*x);
    nt=nt+1;
end
```

2.4.6 Jump commands for loop control

In the loop structures such as `for` and `while` described above, it is sometimes convenient to introduce two jump control commands, `break` and `return`, enabling an early exit from loop structures and subroutines. Usually, the `break` command is used to escape the current loop, while the `return` command is used to escape the current function. For example, the usage of `break` and `return` in `while` loop shown in Figure 2.1 can give different results:



Break: Jump to while loop and execute the following commands

Return: Jump to while loop and return to calling function

FIGURE 2.1
Comparison of break and return commands in loop structure

2.5 M-file functions

Generally, files containing codes of MATLAB commands are called m-files. There are two kinds of m-file: script files and function files, which are created in the Edit/Debug Window shown in Figure 2.2. Script files can be viewed as a simple list of commands and have no input and output parameters, so they are seldom used in practical programming. In contrast, function files can solve problems with arbitrary input and output parameters and have many advantages over script files. In this section, therefore, only function files are introduced.

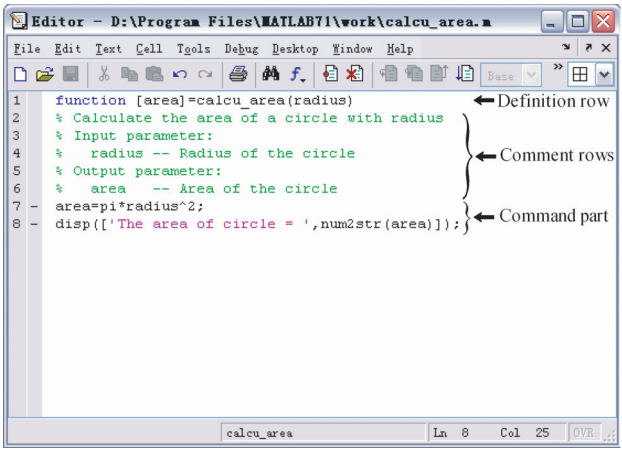


FIGURE 2.2
An m-file function created in the Edit/Debug Window

2.5.1 M-file function structure

A typical m-file function generally includes the following three parts:

(1) M-file definition row in the form

```
function [outvar1,outvar2,...]=functionname(invar1,invar2,...)
```

function statement | output parameters | function name | input parameters

The definition line provides the definition of the m-file function with a function statement using the command "`function`", output parameters (optional) enclosed in square brackets `[ ]`, function name and input argument list (optional) enclosed in round brackets `( )`. Without this line, the file becomes a script file. The syntax of the function definition line also requires that the function name must be the same as the filename (with extension .m) in which the function is written. For example, if the name of the function is "`projectile`", it must be written and saved in a file with the name `projectile.m`. Note that the first word in the function definition line must be typed in lowercase. A common mistake is to type it as `Function` (or something similar).

(2) Comment lines of m-file function. The comment lines next to the definition line in a function belong to the description part. The first commented line before any executable statement and immediately following the function definition line in a function file is called the H1 line. It is the line that is searched by the `lookfor` command. The `lookfor` command is used to look for m-files with keywords in their description. So we should put keywords in the H1 lines of all m-files that we create. The remaining comment lines from the H1 line until the first blank line or the first executable statement are displayed by the `help` command to give a detailed summary of how to use the function. All `help` command lines start with the symbol "*%*".

(3) Command part. The third part is the command block, which includes some commands or executable codes to perform the specified task. The execution ends when either a `return` statement or the end of the function is reached. Because execution stops at the end of a function anyway, the `return` statement is not actually required in most functions, and rarely appears.

When the execution reaches the end of the function, the values stored in the output argument list are returned to the caller and can be used in further calculations.

In summary, a typical m-file function has several features:

- The first function is the main program and other functions invoked by the main function are regarded as subroutines.
- M-file functions use input and output parameters to communicate with other functions and the command window. Meanwhile, [*outvar1, outvar2, . . .*] and (*invar1, invar2, . . .*) are optional in the process of programming.
- M-file functions can call other functions written by users, which are required to have the same path as the former.
- Input parameters allow the same calculation procedure to be applied to different data. Thus, m-file functions are reusable and are useful for developing structured solutions to complex problems.
- Both comment lines and explanation rows start with the symbol "%".
- If an expression in a line is too long, the symbol "`. . .`" can be used to continue writing the code on a new line. However, to avoid wrong usage, it is advisable to add a continue symbol "..." after an algorithm operator. For example,

```
>> 1+2+3+4+...
8

ans =

    18
```

whereas the following expression will cause an error

```
>> 1+2+3+4...
??? 1+2+3+4...
Error: Unexpected MATLAB operator.
```

For example, we can create an simple m-file function to calculate the area of a circle in the Edit/Debug Window (see Figure 2.2). The corresponding definition row of an m-file function, comment lines, and command part are also shown clearly in Figure 2.2. In this m-file function, the name is `calcu_area`, the input augment has the variable `radius` only, and the output augment is the variable `area`.

2.5.2 Global and local variables

Generally, each m-file function has its own local variables, which exist only while the function is executing, and are distinct from variables of the same name in other functions or in the base workspace. However, if a variable is declared to be global in a function, then it will be placed in the global memory instead of the local workspace. If other functions also declare the same variable by the `global` command, they all share a single copy of that variable.

A global variable is declared with the global statement before using it, which has the form:

```
global var1 var2 var3 ...;
```

where `var1, var2, var3` are variables to be placed in global memory. For example, the m-file function created in Figure 2.2 has two local variables; one is `radius`, and the other is `area`. No global variable is defined.

2.5.3 Executing an m-file function

When an m-file is created and saved, it may be executed by typing the name of the m-file in the Command Window. For example, for the m-file function called `calcu_area` with output augment `area` and input augment `radius`, we just need to type an expression in the Command Window and then press 'Enter' to produce the results displayed in the same window (see Figure 2.3).

In addition, in order to execute an m-file function successfully, the right path must be set in the MATLAB Work Window, because MATLAB uses a search path to find m-file functions which are stored in a user-specified file system. For example, the m-file function `calcu_area` created above is stored in the directory

```
D:\Program Files\MATLAB71\work
```

Thus, before executing the m-file function, the user must make MATLAB refer to this path by changing the search path shown in Figure 2.3.

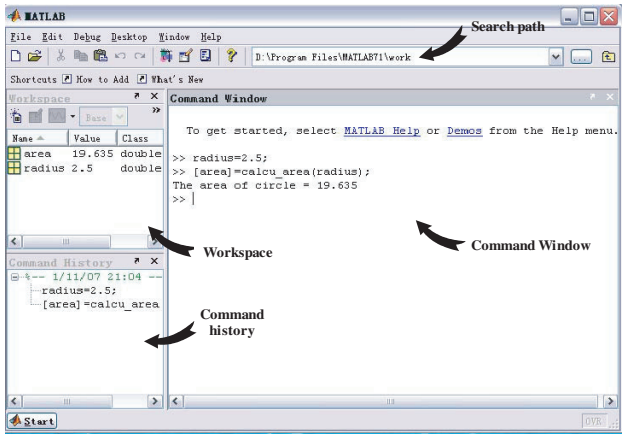


FIGURE 2.3
Execute an m-file function in the Command Window

2.6 I/O file manipulation

In engineering programming, it is usually desirable to read data from the keyboard or a file and print results to the screen or to a file. In this section, some I/O file manipulations are reviewed.

2.6.1 Open and close a file

Before introducing I/O file manipulation, we now discuss how to open and close MATLAB files.

- Open a file for reading or writing

The expression

```
fp = fopen('filename', mode);
```

opens a file named `filename` in the specified mode and returns a scalar MATLAB integer, called a file identifier, `fp`. If `fopen` cannot open the file, it returns -1 to

fp. The mode arguments used in MATLAB are shown in Table 2.6. For example, we open a file named input.txt for reading data in it by means of this command

```
fp=fopen('input.txt','r');
```

TABLE 2.6
Basic modes in the fopen function

Mode	Meaning
'r'	Open file for reading (default)
'w'	Open file for writing; discard existing contents, if any
'a'	Open file for writing; append data to the end of the file
'r+'	Open file for reading and writing
'w+'	Open file for reading and writing; discard existing contents, if any
'a+'	Open file for reading and writing; append data to the end of the file

- Close a file after reading or writing

The expression

```
status = fclose(fp);
```

is used to close the specified file if it is in open status, returning 0 if successful and -1 if unsuccessful.

2.6.2 Input manipulation

- The input function

The input function can be used to prompt the user for numerical or string input from the keyboard. For instance,

```
>> x = input('Enter a value for variable x = ')
Enter a value for variable x = 3

x =
```

```
3
```

It is worth pointing out that prompting via the `input` command may make automation of computing tasks impossible, so it is suggested to avoid using it.

- The `fgets` function

The `fgets` function reads a line from a specified file. In a practical data file, there are some text rows that are used to provide explanation on data or the file and do not affect the values of variables. In this case, the `fgets` function is useful to read these text rows. For example,

```
dummy=fgets(fp);
```

where the returned string `dummy` includes the line terminators associated with the text lines.

- The `deal` function

The `deal` function is used to distribute inputs to outputs. It is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions

```
[Y1, Y2, Y3, ...]=deal(X1, X2, X3, ...)

[Y(1:m)]=deal(X(1:m))
```

As an example, we read the data `NPOIN`, `NDIME` and coordinates array `COORD`, whose size is `NPOIN` by `NDIME`, from a file named input.txt (see Figure 2.4):

The detailed process is expressed as

```
fp=fopen('input.txt','r');
% Fill with ASCII zeros
dummy=char(zeros(1,100));
% Number of points and dimensions
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NPOIN,NDIME]=deal(TMP(1),TMP(2));
% Read coordinates
COORD=zeros(NPOIN,NDIME);
dummy=fgets(fp);
dummy=fgets(fp);
for iPOIN=1:NPOIN
    TMP=str2num(fgets(fp));
    [N,COORD(iPOIN,:)]=deal(TMP(1),TMP(2:1+NDIME));
end
fclose(fp);
```

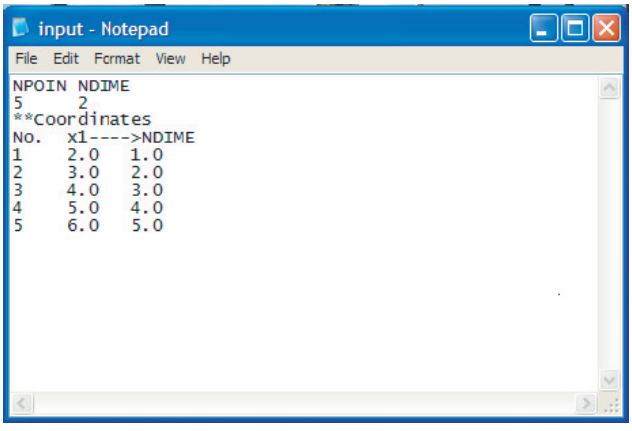


FIGURE 2.4
Configuration of the data file input.txt

2.6.3 Output manipulation

- The `disp` function

The `disp` function is a simple command to display results on screen. It can output a numerical variable, string variable, or a combination of the two. The common forms are

```
disp(x);
disp('string');
disp(['x = ',num2str(x)]);
```

where the `num2str` function is often used in the `disp` function to create a labelled output of a numerical value or a row vector.

- The `fprintf` function

The `fprintf` function is used to write formatted data to a specified file referred to by the file identifier `fp`, or to screen without `fp`. The typical form is

```
fprintf(fp, outformat, outvariables, ...)
```

where the `outformat` string specifies how the `outvariables` are converted and displayed. The `outformat` string can contain any text characters and a conversion code for each of the `outvariables`. Table 2.7 shows the basic conversion codes in MATLAB for the formatted output of results.

For example,

TABLE 2.7
Basic MATLAB conversion codes

Code	Conversion instruction
`%s`	String
`%5d`	Integer with specified field width
`%12.5f`	Floating-point value with specified field width and precision
`%12.5e`	Floating-point value in scientific notation with specified field width and precision
`%g`	Most compact form of either %f or %e. Insignificant zeros do not print
`\n`	Insert new line

```
>> x=[1, 2, 3];
>> fprintf('%5.3f %7.5f\n', [x; exp(x)])
1.000  2.71828
2.000  7.38906
3.000  20.08554
```

2.7 Vectorization programming with MATLAB

Due to the important feature that the basic data unit of MATLAB is the matrix, the vectorisation operation can be used to provide simpler expressions by vector operations. Properly vectorised expressions are equivalent to looping over the elements of the matrices being operated upon, but a vectorised expression is more compact and leads to codes that execute more quickly than looping procession [3].

For example, the computation procedures shown in Figure 2.5 are mutually equivalent.

Although vectorisation is excellent in execution efficiency, non-vectorised codes, also called scalar codes, are easily understood and can reduce errors in programming. Therefore, keeping codes clean and correct should be a matter of careful consideration [3].

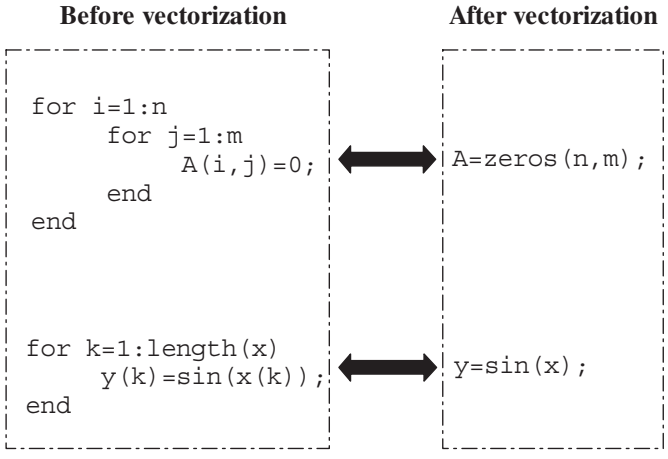


FIGURE 2.5
Illustration of vectorization programming in MATLAB

2.8 Common built-in MATLAB functions

There are many built-in functions in MATLAB [1]; Table 2.8 lists those used in subsequent chapters.

TABLE 2.8
Commonly used built-in MATLAB functions

Function	Description
Mathematical Functions	
`abs(x)`	Calculates absolute value and complex magnitude of an argument `x`
`cos(x)`	Calculates cosine of an argument `x` in radians
`sin(x)`	Calculates sine of an argument `x` in radians
`tan(x)`	Calculates tangent of an argument `x` in radians
`acos(x)`	Calculates inverse cosine of an argument `x`, result in radians

Continued.....

Function	Description
`asin(x)`	Calculates inverse sine of an argument `x`, result in radians
`atan2(y,x)`	Calculates inverse tangent $\tan^{-1}(y/x)$ over four-quadrant, result in radians in the range $[-\pi, \pi]$
`log(x)`	Calculates natural logarithm of an argument `x`
`exp(x)`	Calculates exponential of an argument `x`
`max(x)`	Calculates maximum elements of an array `x` and its location index
`min(x)`	Calculates minimum elements of an array `x` and its location index
`sqrt(x)`	Calculates square root of an argument `x`
`real(x)`	Calculates real part of complex number `x`
`imag(x)`	Calculates imaginary part of complex number `x`
Rounding Functions	
`ceil(x)`	Rounds the argument `x` to the nearest integer greater than or equal to `x`
`floor(x)`	Rounds the argument `x` to the nearest integer less than or equal to `x`
`fix(x)`	Rounds the argument `x` to the nearest integer towards zero
`round(x)`	Rounds the argument `x` to the nearest integer
`rem(x,y)`	Returns `x-n*y` where `n=fix(x/y)`
Matrix Functions	
`zeros(m,n)`	Creates an `m`-by-`n` matrix of zeros
`eye(m,n)`	Creates an `m`-by-`n` matrix with 1 on the diagonal and 0 elsewhere
`inv(A)`	Calculates the inverse of a square matrix `A`
`cond(A)`	Calculates the condition number of a square matrix `A`
`sum(A)`	Returns a row vector of the sums of each column of a matrix `A`
`A'`	Calculates the transposition of a matrix `A`
`det(A)`	Calculates the determinant of a square matrix `A`
`[m,n]=size(A)`	Calculates the size of each dimension of a matrix `A`

Continued.....

Function	Description
String Conversion Functions	
`num2str(x)`	Converts `x` into a character string representing a number with a decimal point
`str2num(x)`	Converts character string `x` into a number
I/O Functions	
`fopen`	Opens a file for reading or writing
`fclose`	Closes an opened file
`fgets(fp)`	Returns the next line of the file associated with file identifier fp
`deal`	Copies the contents of input to all the requested outputs
`fprintf`	Writes data to the file according to specified format
`disp`	Displays a variable in the Command window
Coordinate Transformation Functions	
`[s,r]=cart2pol(x,y)`	Transforms two-dimensional Cartesian coordinates stored in `x` and `y` into polar coordinates stored in `s` in radians and `r`
`[x,y]=pol2cart(s,r)`	Transforms the polar coordinate data stored in `s` in radians and `r` to two-dimensional Cartesian stored in `x` and `y`
Debug Function	
`error('message')`	Displays specified error message and returns control to the keyboard

References

[1] Chapman SJ (2002), *MATLAB Programming for Engineers* (2nd edition). Pacific Grove, CA: Brooks/Cole-Thomson Learning.

[2] Sigmon K, Davis TA (2002), *MATLAB Primer* (6th edition). Boca Raton: Chapman & Hall/CRC.

[3] Kiusalaas J (2005), *Numerical Methods in Engineering with MATLAB*. New York: Cambridge University Press.

[4] Lyshevski SE (2003), *Engineering and Scientific Computations Using MATLAB*. New Jersey:Wiley-Interscience.

[5] Wilson HB, Turcotte LH, Halpern D (2003), *Advanced Mathematics and Mechanics Applications Using MATLAB* (3th edition). Boca Raton: Chapman & Hall/CRC.

[6] Kattan PI (2003), *MATLAB Guide to Finite Elements*. Berlin: Springer-Verlag.

3

C programming

In the previous chapter some elements of MATLAB were discussed. As an alternative to MATLAB, we now deal with fundamentals of C programming. Chapters 2 and 3 provide a combined source of computer programming for reference in later chapters. C is a general-purpose, block structured computer programming language. Although it was first developed in 1972 as a system programming language for writing compilers and operating systems, it has been used equally well to write major programs in many different domains including numerical computation. C has been around for several decades and has won widespread acceptance because it gives programmers maximum control and efficiency [1 - 4].

In contrast to MATLAB, C is a relatively "low-level" language. This characterisation is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines. As a result, compiled C code runs in a stripped down run-time model and makes C faster than MATLAB.

In a word, the simple and effective syntax lets programmers express what they want in the minimum time by staying out of their way.

In this chapter, the basic features and functions of C language are discussed in order to provide basic programming knowledge for later chapters. The discussion focuses on the development in Refs. [1 - 4].

3.1 Data types, variable declaration and operators

3.1.1 Data types

There are four basic data types in the C language system: floating point number, double, integer, and character. The notion of a data type reflects the possibility of the 1s and 0s stored in a computer memory location being interpreted in different ways. For full details, an appropriate text on computer architecture should be consulted. Actually, C provides a standard, minimal set of basic data types. Sometimes the four types above are called "primitive" types (see Table 3.1). More complex data structures can be built up from these basic types.

- Four basic data types

  - `char`: a single byte, 8 bits, capable of storing one ASCII character
  - `int`: an integer, 16 bits, typically reflecting the natural size of integers on the host machine
  - `float`: a single-precision floating point, 32 bits
  - `double`: a double-precision floating point, 64 bits

- Four extension types
  - `signed`: signed integer and char have the same amount of storage as an `int` and a `char`, respectively.
  - `unsigned`: unsigned integer and char are always positive or zero
  - `long`: provides extended lengths of integers or extended-precision floating point
  - `short`: provides small integers which have less than or the same amount of storage as an `int`

TABLE 3.1
Common data types in C

	Type	Width (bits)	Value range
Basic	char	8	-128 to 127
	int	16	-32768 to 32767
	float	32	six-digit precision
	double	64	ten-digit precision
Extension	unsigned char	8	0 to 255
	signed char	8	same as `char`
	unsigned int	16	0 to 65535
	signed int	16	same as `int`
	short int	16	same as `int`
	long int	32	-2147483648 to 2147483647
	unsigned long int	32	0 to 4294967295
	signed long int	32	same as `long int`

3.1.2 Variable declaration

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable. For example, the following commands

```
int lower;
char upper[10];
double step[5];
```

are used to define the `int`-type variable `lower`, `char`-type array `upper[10]` including 10 characters and `double`-type array `step[5]` with 5 elements, respectively.

Regarding variable declaration, there are several important aspects to be noted:

- variables must be declared before they are used
- variables declared inside a block are local to that block and cannot be accessed from outside the block
- variables can be initialised after they are declared
- explicit type conversions can be forced in any variable by the construction:

```
(typename)variable;
```

The following example executes an explicit type conversion between variable `x` and `k`. As a result, the final value of int-type variable `k` is 2, while the value of double-type variable `x` is 2.5.

```
#include<stdio.h>
void main()
{
    double x;
    int k;
    x=5/2.0;
    k=(int)x;
    printf("x = %5.3f\n",x);
    printf("k = %d\n",k);
}
```

In addition to variable declaration as described above, variables usually have the following four storage classes:

- `auto`: variable is not required outside its block (default)
- `register`: variable will be allocated on a CPU register
- `static`: allows a local variable to retain its previous value upon reentry
- `extern`: global variable declared in another file.

For convenience of application, we focus on the usage of global variables. If a global variable defined outside a function is used in other functions, the `extern` declaration is necessary. For example, we design the following function to compute

the area of a circle with radius 2.5. There are two functions (one is the `main` function and the other is the subroutine `Calcu_Area`) in our project, which are discussed in detail in the following section. It is obvious that the global variable `PI` is defined outside the `main` function. Therefore it must be declared by `extern` in order to use it in the subroutine `Calcu_Area`.

```
void Calcu_Area(double radius,double *area)
{
    extern double PI;
    *area=PI*radius*radius;
    return;
}

#include<stdio.h>
double PI;
void main()
{
    void Calcu_Area();
    double radius,area;
    PI=3.1415926;
    radius=2.5;
    Calcu_Area(radius,&area);
    printf("The area of the circle = %5.3f\n",area);
    return;
}
```

3.1.3 Operators

The operators used in C include arithmetic operators, rational operators and logic operators, which are summarised in Table 3.2.

With the arithmetic operators listed in Table 3.2, it is worth mentioning that improper usage of division between two integers may cause a severe error. The following example shows the right way to execute division between two integers.

```
double k1,k2,k3;
k1=5/2;
k2=5/2.0;
k3=5.0/2;
```

The values of variables `k1`, `k2` and `k3` are 2.0, 2.5 and 2.5, respectively. In division calculation, one of the two integers must be converted into the form of a floating point number in order to obtain a correct result.

TABLE 3.2
Common operators in C

Type	Operator	Operation
Arithmetic	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
	%	Remainder (suitable only for integer division)
Rational	>	Greater than
	>=	Greater than or equal to
	<	Less than
	<=	Less than or equal to
	==	Equal to
	!=	Not equal to
Logic	&&	AND
	||	OR
	!	NOT

3.2 Control structures

The control-flow of a language specifies the order in which computations are performed. A detailed description of control structures in C used in later chapters is now presented.

3.2.1 `if-else` structure

The most general form of making multi-way decisions by the `if-else` structure is written as

```
if(expression 1)
{
    Block 1;
}
else if(expression 2)
{
    Block 2;
}
.
.
else
{
    Block 3;
}
```

In the structure above, `Block 3` handles the default case where none of the other conditions is satisfied. Sometimes it can be omitted if there is no explicit action for the default.

As an illustration, the following `if-else` structure is used to identify whether the variable `k` is negative, positive or zero.

```
if(k>0)
{
     printf("k is positive\n");
}
else if(k<0)
{
     printf("k is negative\n");
}
else
{
     printf("k is zero\n");
}
```

3.2.2 `switch-case` structure

The `switch-case` statement is an alternative form for multi-way decisions that identify whether an expression matches one of a number of `constant` integer values and branches accordingly. Typical syntax is

```
switch(expression)
{
    case value 1:
        Block 1;
        break;
    case value 2:
        Block 2;
        break;
    .
    .
    default:
        Block n;
        break;
}
```

In the `switch` statement, each case needs its own keyword and a trailing colon "`:`". In addition, the explicit break statements are necessary to exit the switch. For example,

```
switch(j)
{
    case 0:
        k=2;
        break;
    case 1:
        k=1;
        break;
    default:
        printf("unexpected value for variable j\n");
        exit(1);
}
```

It should be mentioned that omitting the `break` statement is a common error occurring in C programming. This error leads to an inadvertent behavior, although it can compile successfully, as is clearly displayed from the executing sequence of the `switch` structure [4] as shown in Figure 3.1.

3.2.3 `for` loop

The `for` loop in C is the most general loop construct. The loop header contains three parts: an initialisation, a continuation condition, and an increment, separated by semicolons "`;`", that is

```
for(initialisation;condition;increment)
{
    Block;
}
```

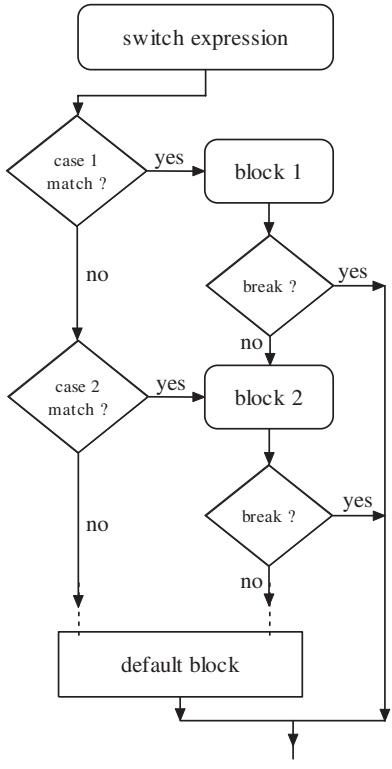


FIGURE 3.1
Executing sequence of a standard `switch` structure [4]

The initialisation is first executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true. After every execution of the loop, the increment process is executed. For example,

```
for(i=0;i<5;i++)
{
     printf("%d\n",b[i]);
}
```

3.2.4 `while` loop

The syntax of a `while` loop can be written as

```
while(expression)
{
     Block;
}
```

As an example, following `while` iteration structure

```
i=0;
while(i<5)
{
     printf("%d\n",b[i]);
     i=i+1;
}
```

produces the same result as that obtained using the `for` loop.

Comparing the two loops, we can see that unlike the `for` loop, the `while` loop evaluates the test expression before every loop. If it is non-zero, the block is executed and the expression re-evaluated. This cycle continues until the expression becomes zero, at which point execution resumes after the `while` structure. Obviously, the `while` loop can execute zero times if the condition is initially false.

3.2.5 Jump control commands

In C, there are several jump control commands, such as `goto`, `continue`, `break` and `return`. The usage of jump commands can achieve specified purposes in a program.

- `break` is used to jump out of the current loop and switch structure. It means that the `break` command can cause the innermost enclosing loop or switch to be exited immediately.

- `return` is used to jump and return to calling functions, which can be used outside the loop structure.

- `goto` is used to jump to specified locations. The feature of the `goto` command makes it easily used in multi-layer loop structures; however, this statement may decrease the readability of the program, so it is seldom used in C programming.

- `continue` statement is used to jump to the bottom of the current loop structure and start a new loop.

3.3 Advanced array and pointer action

Pointers are often used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than that obtained in other ways. Pointers and arrays are closely related; in this chapter we also explore this relationship and show how to exploit it.

3.3.1 Arrays

In C, an array is formed by laying out all the elements contiguously in memory. Square bracket syntax `[]` can be used to refer to the elements in the array. For example, the statement

```
double temp[5][2];
```

provides a definition of two-dimensional floating-point array named `temp`, whose size is 5 by 2. The simplest way to refer to elements in the array `temp` is to use syntax `[]`, such as `temp[3][2]` denotes the element at the third row and the second column.

In fact, all elements in an array are arranged continuously in memory by row order. For example, Figure 3.2 shows the order of elements in the array `temp` mentioned above.

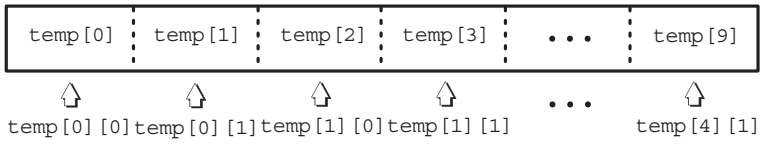


FIGURE 3.2
Configuration of elements alignment of array in C

Therefore we can also express a particular element `temp[i][j]` as

```
temp[i][j]=temp[i*2+j]
```

It should be pointed out that all indices will be equal to or greater than zero because all C arrays begin at position zero.

3.3.2 Pointers

A pointer is a variable that stores a memory address, and must be declared and initialised before it can be used. There are two operators related to a pointer: one is "`&`" which provides the address of an object, and the other is "`*`", which is used to make a pointer for accessing an object. For instance, the statement

```
int *p;
```

defines a pointer variable `p`, and the expression

```
p=&a;
```

provides the address of variable `a` to pointer `p`. As a result, we have

```
*p=10;
```

if the variable `a` has a value of `10`.

3.3.3 Pointers and arrays

In C, there is a close relationship between pointers and arrays which should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers, which gives programmers greater flexibility in using arrays. For example, the process for providing the address of the first element of the array `a` to a pointer `p` is shown in Figure 3.3.

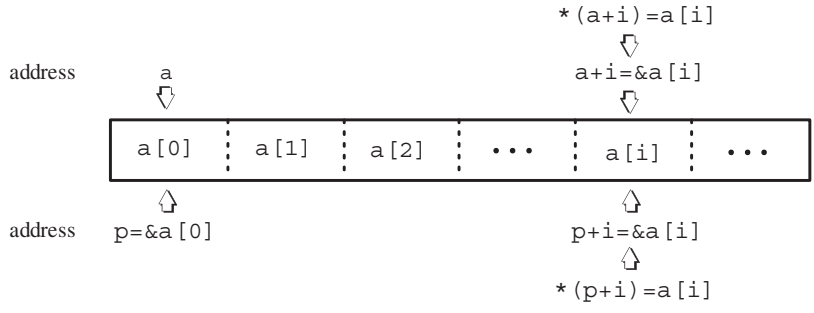


FIGURE 3.3
Relations of pointers and arrays in C

In fact, the name of an array also stores the location of the initial element in C, so in the above example, `p` can be written as based on the above example,

```
p=a
```

However, there is a difference between an array name and a pointer that must be kept carefully in mind. A pointer is a variable, so such expressions as `p=a` and `p++` are legal, whereas an array name is not a variable, so constructions like `a=p` and `a++` are illegal.

3.3.4 Initialisation of array and storage management

In C language, each array must be allocated space in memory before being used. The library function `calloc` can be used for this purpose. The typical syntax is

```
array=(type *)calloc(n,sizeof(type));
```

for example, the expression

```
COORD=(double *)calloc(n*m,sizeof(double));
```

arranges space in memory for the double-type array `COORD` with size `n*m`.

When an array or variable is not to be used again, it is better to free it and vacate the memory occupied. The library function `free` can be used for this task. For example,

```
free(COORD);
```

vacates the spaces occupied by the array or variable COORD.

3.4 Functions and parameter transfer

Functions can break large tasks into small ones and enable people to directly use what others have done instead of starting over from scratch. One or more functions can communicate each other in a subroutine/routine of C by means of parameter transfer [1 - 3].

3.4.1 Types of functions

In general, each C program contains at least one function, main, also called the main function of the program, which can call other functions, or subfunctions, to perform a task.

In C language there are two types of functions:

- Built-in library functions

Some built-in library functions are available in the standard library in C and can be used directly by programmers. However, the following statement

```
#include<header file name>
```

must be placed at the top of the main function or any subroutine so that you can access the corresponding library functions included in this header.

- User-defined functions

User-defined functions are defined and employed by users to achieve specified purposes. Before a user-defined function can be used, a statement must be provided naming it. Usually the typical form of a function definition in ANSI C is

```
type functionname(type1 param1, type2 param2, ...)
```

The easiest way to understand prototypes is by example. To do this here, we again list the function before it is used to calculate the area of a circle as our example.

```
void Calcu_Area(double radius,double *area)
{
    extern double PI;
    *area=PI*radius*radius;
    return;
```

```
}

#include<stdio.h>
double PI;
void main()
{
    void Calcu_Area();
    double radius,area;
    PI=3.1415926;
    radius=2.5;
    Calcu_Area(radius,&area);
    printf("The area of the circle = %5.3f\n",area);
    return;
}
```

In this example,

```
void main()
```

gives the definition of the main function, and the subfunction `Calcu_Area` is defined by

```
void Calcu_Area(double radius,double *area)
```

which is called by the main function. Meanwhile, the declaration of the subroutine `Calcu_Area`

```
void Calcu_Area();
```

occurs in the main function.

Apart from these two user-defined functions, the function `printf` used in the main function is a built-in library function in C, so the corresponding head file `stdio.h` must be added before the main function in the form

```
#include<stdio.h>
```

Regarding the types of functions defined by users, generally there are three common types in our C programming application, according to different types of returned value:

```
void functionname()
int functionname()
double functionname()
```

in which the `int` and `double` type function require the function to return an `int` or `double` value, while the `void` indicates that no value is required to be returned.

3.4.2 Function call and parameter transfer

- General parameter transfer

As an illustration, Figure 3.4 shows the way of parameter transfer in C program.

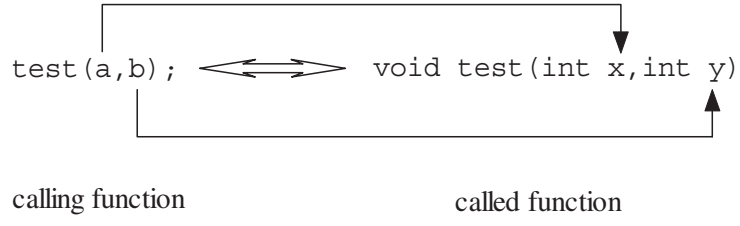


FIGURE 3.4
Illustration of simple parameter transfer

The above process is the most general and simple way to perform parameter transfer. The values of variables `a` and `b` in the calling function `test(a,b)` can be transferred to local variables `x` and `y` in the called function `void test(int x,int y)`, respectively. However, the changes of local variables `x` and `y` do not change the values of practical variables `a` and `b`.

- Pointer parameter transfer

Because C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function, as we see above. The effective way to change the value of a variable in the calling function is to pass pointer operations. For example, Figure 3.5 shows the procedure of parameter transfer between the calling function `test` and the called functions, also viewed as definition function of `test`.

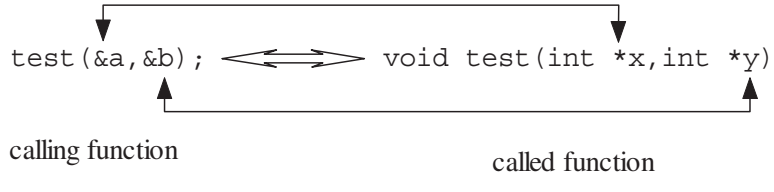


FIGURE 3.5
Illustration of pointer parameter transfer

Since the operator "`&`" produces the address of a variable, `&a` is a pointer to the

variable `a`. In the called function test itself, the parameters are declared as pointers, and the operands are accessed indirectly through them. Moreover, pointer arguments enable a function to access and change objects in the functions that call them.

The example above for calculating the area of a circle is a typical one to demonstrate the effect of pointer transfer; that is, the value of `area` is obtained by calling

```
Calcu_Area(radius,&area);
```

and it is declared in the subroutine

```
void Calcu_Area(double radius,double *area);
```

while the transfer of the variable radius is performed by a single-directional way (the value is transferred from the called function to calling function) as shown in Figure 3.4. It is evident that the value of a variable in the calling function is determined by the value of variable in the called function, instead of by a pointer.

- Array parameter transfer

Because the name of an array stores the location of the initial element in C, we can conveniently transfer and alter values of the array to a simpler form by using this feature. For example, the procedures in Figure 3.6 are all legal for array parameter transfer:

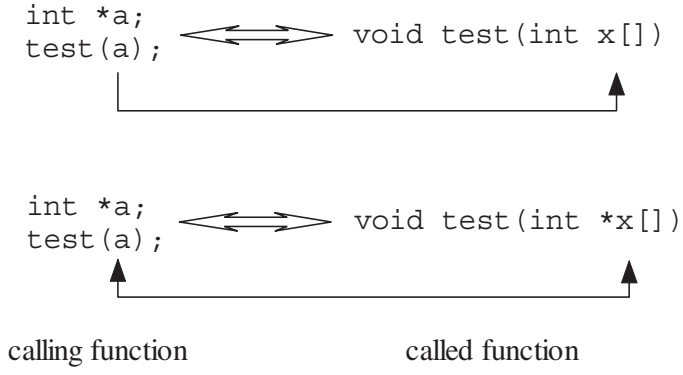


FIGURE 3.6
Illustration of array parameter transfer

3.5 File manipulation

An essential part of any useful computer program is the facility to manipulate enormous amounts of data stored externally. The file is the basic unit of storage for many operating systems, from Unix to Mac. Therefore, file manipulation is vital for engineers. In this section we provide some popular manipulations of files [1 - 3].

3.5.1 Open and close a file

Before it can be read or written, a file has to be opened by the library function `fopen`, which is used to open an external file such as input.dat or input.txt, and returns a pointer to be used in subsequent reading or writing of the file.

The definition of such pointer, called the file pointer, is

```
FILE *fp;
```

in which `fp` is a file pointer, and the built-in library function `FILE` can be obtained from the header file `stdio.h`.

Next, it is natural to open a file by the defined file pointer `fp` and the library function `fopen`. The proper syntax to open a file is

```
fp=fopen("filename",mode);
```

in which the second argument mode indicates how to use the file and the commonly used modes are listed in Table 3.3.

TABLE 3.3
Common modes in file manipulation

Mode	Meaning
`"r"`	Read data from a text file
`"w"`	Write to the file, overwriting existing data
`"a"`	Add data to the end of an existing text file

Regarding the three modes listed above, we must mention that trying to use mode `"r"` to read a file that does not exist will lead to an error, and `fopen` function will return `NULL`. Trying to write data using mode `"w"` to an existing file will cause

the old contents to be discarded, while appending mode `"a"` can prevent this and preserve the old contents.

When a file has been used and is no longer needed, it should be closed. This can be done by the library function:

```
fclose(fp);
```

in which the command `fclose` flushes any buffers maintained for the given file and closes the file. Buffers allocated by the standard I/O system are freed.

3.5.2 Input data from a file

- Read string data

Reading string data from an opened file can be performed by using the library function `fgets`, which can read a line from the file to a character array, for example,

```
fgets(title,n,fp);
```

where `title` is a character array which includes n-1 character elements.

- Read formatted data

The general library function used for reading formatted data from an opened file is `fscanf`. A typical example is

```
fscanf(fp,"%d, %lf, %s",&N,&F,&S);
```

where `%d`, `%lf`, `%s` is the list of formats, and `&N`, `&F`, `&S` are the addresses of variables `N`, `F`, `S` storing read data.

3.5.3 Output data to a file

Complementary to `fscanf`, the library function used for writing formatted data to an opened file is `fprintf`. For example, the expression

```
fprintf(fp,"Number of elements = %d",NE);
```

outputs the expression "`Number of elements = %d`" to a file whose pointer is `fp` and replaces the `%d` part by the value of variable `NE`.

3.6 Create and execute C codes in visual C++ platform

In subsequent chapters, all C codes are created and executed using the Visual C++ platform, which is more versatile than traditional platforms such as Turbo C. It is therefore necessary to introduce the basic operations including how to create a project in the Visual C++ 6.0 platform and then create C++ source files and add them to the project.

3.6.1 Creating a project

The Visual C++ Project Model programmatically exposes the functionality of the compiler, linker, and other build tools, so all programming in Visual C++ is required to create a project to enable all functions.

In Visual C++, the default new project is created by opening the "New" menu. The following two figures present a simple example of the complete process to set up a project named "Calculate Circular Area" located in the base directory " D: ". In our application, Win32 Console Application listed at the left of the first dialog box is always selected. Note that as you type the project's name, it is also automatically added to the basic directory defined by users in the directory test box. Finally, select "OK" to set up a project (see Figure 3.7).

Next, you are asked what kind of console the application is to create. Select "An empty project", which is also the default, and then press the "Finish" button to complete it. Another dialog box may appear; click "OK" immediately to close it, because it is just a summary for the created project (see Figure 3.8).

Thus, a project is created in Visual C++ platform.

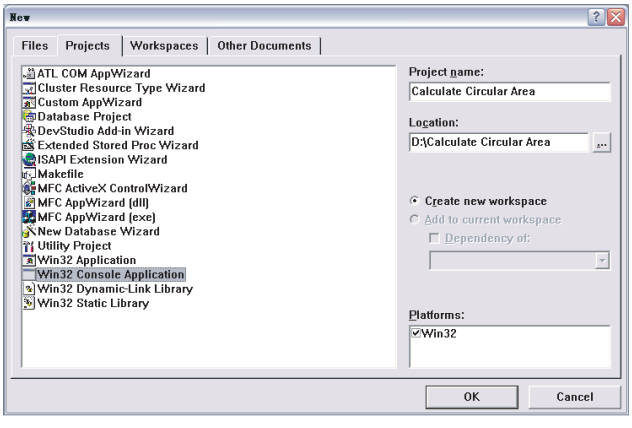


FIGURE 3.7
The first dialog box for creating a project

3.6.2 Creating a C source file

After setting up a project, Visual C++ returns to the startup screen. Reselecting the "New" menu will bring up the "New" dialog box again. In this case, the "Files" tab, instead of "Projects" tab, is selected as the default. Select "C++ Source File" as the type of source code and give a file name, for example, `Main.c` in Figure 3.9, at the right of the dialog box. Finally select "OK" to end the creation of a C function. Repeating the above procedure, more C++ source files can be created and added to

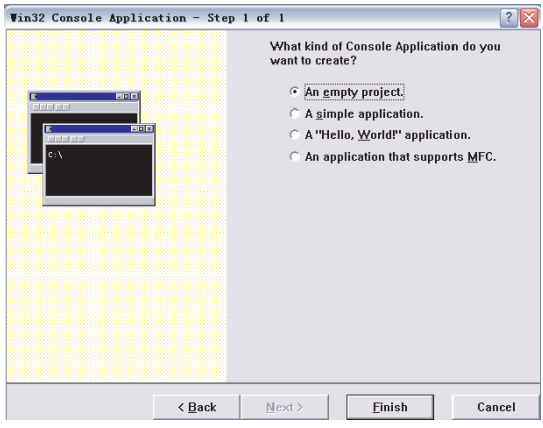


FIGURE 3.8
The second dialog box for creating a project

the same project, in which all source files can be called mutually.

For example, in the project "Calculate Circular Area" discussed in Section 3.4.1, two source files "`Main.c`" and "`Calcu_Area.c`" are created subsequently (see Figure 3.9 and Figure 3.10).

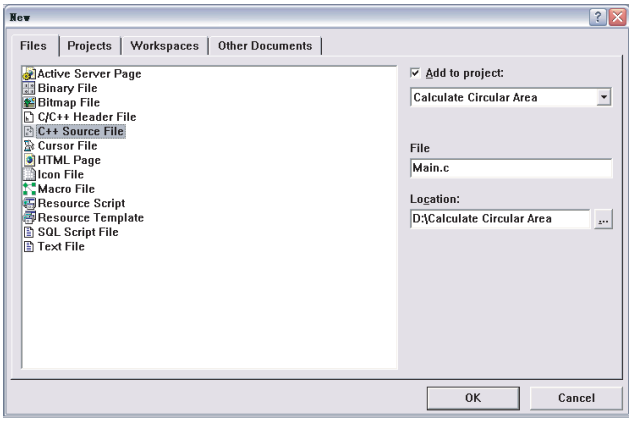


FIGURE 3.9
Creating the source file `Main.c` in the project Calculate Circular Area

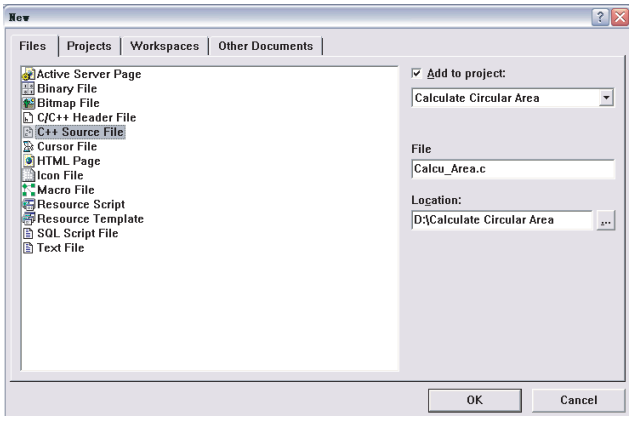


FIGURE 3.10
Creating the source file `Calcu_Area.c` in the project Calculate Circular Area

3.6.3 Compile, build and execute a C program

After writing the C codes in the work window, such as `Main.c` and `Calcu_Area.c` (see Figure 3.11 and Figure 3.12), we can compile, build and execute them. Compiling the program translates the C program into machine code, building the program combines the machine code with additional code needed to create an executable file, and executing refers actually to the executable file. The buttons for compile, build and execute are displayed in Figure 3.13. After you compile and build the project, any errors will appear at the bottom of the window.

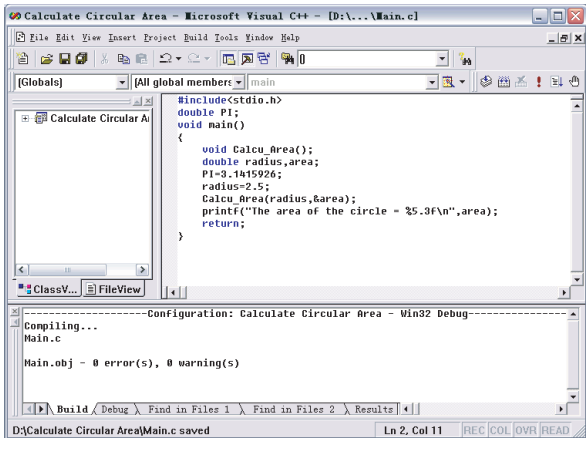


FIGURE 3.11
C code of source file `Main.c`

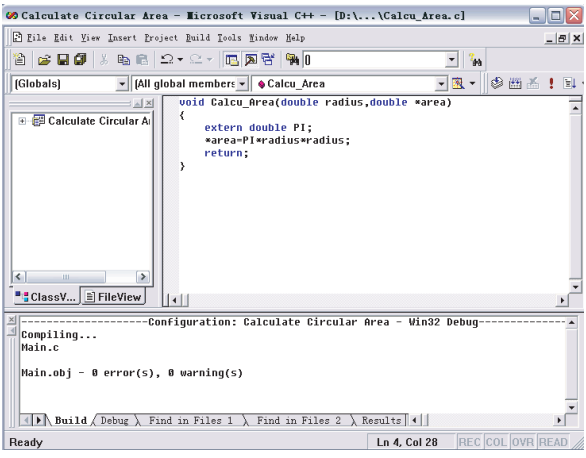


FIGURE 3.12
C code of source file `Calcu_Area.c`

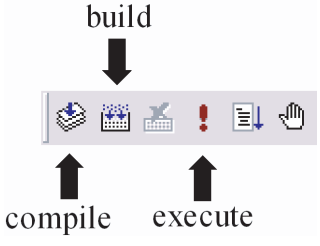


FIGURE 3.13
Compile, build and execute a program

3.6.4 Output result

In the end, the final result can be displayed to screen (Figure 3.14).

3.7 Common library functions and related head files

In C language, many basic built-in library functions are available to a C program in the form of standard library functions [1 - 3]. To call these, a program must include

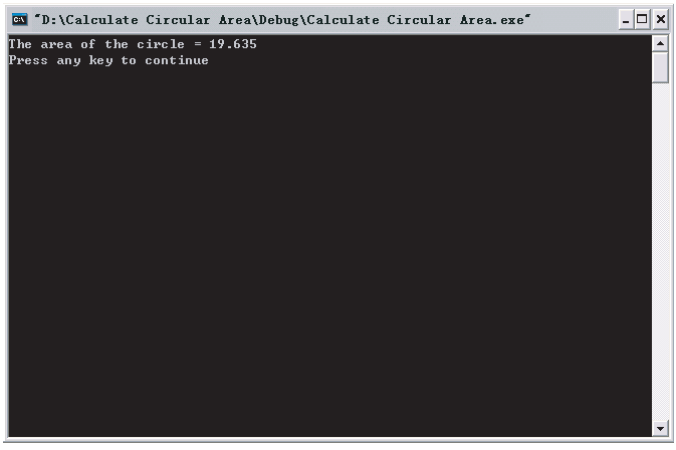


FIGURE 3.14
Results are displayed to screen

the appropriate head file. Some library functions and related head files commonly used in later chapters are listed in Table 3.4 for the sake of convenience.

TABLE 3.4
Common library functions and related head files in C

Function	Description
Mathematical Functions included in header <math.h>	
`fabs(x)`	Calculates the absolute value of argument x, returns *double* value
`abs(n)`	Calculates the absolute value of an *int* argument n, returning *int* value
`cos(x)`	Calculates cosine of argument x in radians, returning *double* value
`sin(x)`	Calculates sine of argument x in radians, returning *double* value
`tan(x)`	Calculates tangent of argument x in radians, returning *double* value
`acos(x)`	Calculates $\cos^{-1}(x)$, result in radians in the range $[0, \pi]$
`asin(x)`	Calculates $\sin^{-1}(x)$, result in radians in the range $[-\pi/2, \pi/2]$

Continued.....

Function	Description
`atan2(y,x)`	Calculates $\tan^{-1}(y/x)$ over four-quadrant, result in radians in the range $[-\pi, \pi]$
`pow(x,y)`	Calculates x^y, returning *double* value
`log(x)`	Calculates $\ln x$, returning *double* value
`exp(x)`	Calculates e^x, returning *double* value
`sqrt(x)`	Calculates $\sqrt{x}$, returning *double* value
`fmod(x,y)`	Calculates floating-point remainder of x/y
`ceil(x)`	Rounds argument x to the smallest integer not less than x, returning *double* value
`floor(x)`	Rounds argument x to the largest integer not greater than x, returning *double* value
I/O functions included in header <stdio.h>	
`FILE`	Defines a file pointer
`fopen`	Opens a file for reading or writing
`fclose`	Closes an opened file
`fgets`	Reads at most the next n-1 characters into array, stopping if a new line is encountered
`fscanf`	Reads data from a file under control of *format*
`fprintf`	Writes data to a file under control of *format*
Utility functions included in header <stdlib.h>	
`calloc`	Allocates memory to array
`free`	Frees memory space pointer
`exit`	Causes normal program termination

References

[1] Kernighan BW, Ritchie DM (1988), *The C Programming Language* (2nd edition). Englewood Cliffs, N.J.: Prentice Hall.

[2] Hancock L, Krieger M (1982), *The C Primer*. New York: McGraw-Hill.

[3] King KN (1996), *C Programming: A Modern Approach*. New York: Norton.

[4] William HP, Saul AT, William TV, Brian PF (1992), *Numerical Recipes in C:*

The Art of Scientific Computing (2nd edition). Cambridge: Cambridge University Press.

4

Commonly used subroutines

4.1 Introduction

In the previous two chapters, the foundations of MATLAB and C programming were described. This chapter illustrates the application of these two computer languages to both potential problems and plane elasticity, presenting some subroutines commonly used in later chapters. The subroutines discussed in this chapter include inputting data from a file and outputting results to a file, Gaussian integration, generation of shape functions, assembly of an elemental stiffness matrix, introduction of boundary conditions, and solution of global stiffness equations. These common modules are provided in both MATLAB and C codes.

For the sake of simplicity, we use the terminology "generalised displacements" to represent both potential field and displacement fields, and "generalised stresses" to represent both potential flux and elastic stresses (see Figure 4.1). In consequence, most variables can be commonly used in both potential and plane elastic problems.

4.2 Input and output

Although input of necessary data and output of final computational results for any program are technically less demanding aspects, they are important to practical engineers. It is well known that preparation of data and interpretation of intermediate results in running a computer program are generally time-consuming in engineering analysis. Therefore, input and output subroutines are essential components in any computer program used for engineering computation. This section presents both input and output subroutines which can be used in either potential or plane elastic problems.

4.2.1 Input of data

To ensure generality, the subroutine `INPUTDT` in both C and MATLAB is designed to be able to open and read a file for either potential problems or plane elastic problems. The data required includes the following four major classifications:

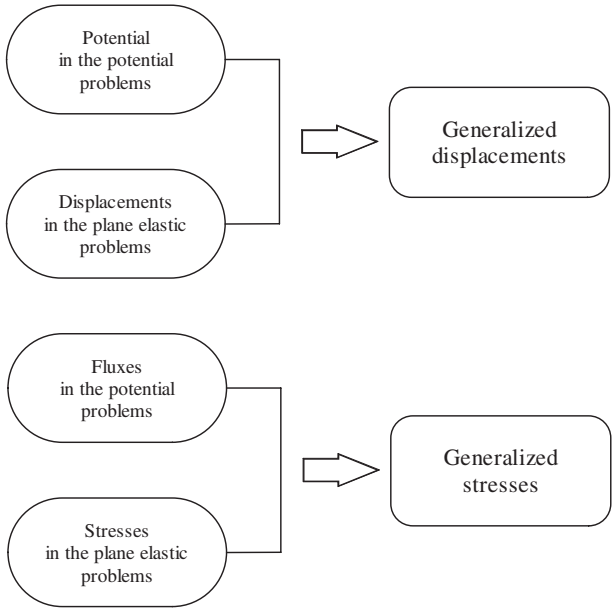


FIGURE 4.1
Definition of generalised displacement and stress components

- Basic control parameters

To reduce the total number of subroutines required, several subroutines are designed so that they can be utilised in more than one program. To this end, it is essential that some basic control parameters are supplied as input data. Examples of such parameters are number of Trefftz complete functions, number of dimensions, and number of degrees of freedom (DOF) per node. These numbers differ between potential problems and plane elastic problems. To enable the data input subroutine to apply for both potential and plane elastic problems, the parameters mentioned above must be specified.

A complete list of these control parameters is presented as follows.

`NTREF`	Number of T-complete functions
`NTYPE`	Type of problems = 0 potential problems = 1 plane stress problems = 2 plane strain problems
`NNODE`	Number of nodes per element
`NEDGE`	Number of edges per element
`NODEG`	Number of nodes per edge
`NDIME`	Number of coordinate components per node

NDOFN	Number of degrees of freedom per node
NSTRE	Number of generalised stress components
NMATS	Total number of different materials in the structure
NPROP	Number of material parameters required to define the characteristics of a material completely
NGAUS	Number of Gaussian points required in Gaussian numerical integration
NPOIN	Total number of nodal points
NELEM	Total number of elements
NVFIX	Total number of boundary nodes at which one or more degrees of freedom are restrained
NPLOD	Total number of concentrated loads
NDLEG	Total number of loaded edges

Among the parameters listed here, some parameters such as NTREF, NTYPE, NDIME, NDOFN, NSTRE, NNODE, NODEG, NEDGE are frequently used in programming. It is better to define them as global variables.

- Geometric data

Once a structure has been discretised into a number of finite elements, structural geometry including nodal coordinate and nodal number must then be used to define these elements. In the analysis, each node of the structure is assigned a unique identification number, and related coordinates are supplied through an input subroutine. Each element is then defined by the nodes appearing in the element. Generally, an element can be completely defined by specifying its nodal connection, material identification number and nodal coordinates. The corresponding arrays used for storing these data are:

MATNO(NELEM,1)	Array of material number of each element
LNODS(NELEM,NNODE)	Array of elemental node-connections
COORD(NPOIN,NNIME)	Array of nodal coordinates

- Boundary conditions

Having defined the geometry of a structure, it is now necessary to specify its boundary conditions. Basically there are two types of boundary condition: geometric (displacement in solid mechanics) and natural. If an appropriate formulation is used for the finite element equations on the basis of variational principles, then the natural boundary conditions are automatically satisfied. For the geometric boundary condition, displacements at a nodal point may be unconstrained (unknown) or constrained. One type of constraint is a single-point constraint, which is a known value

of the relevant degrees of freedom at a node. For the natural boundary condition, the values of the traction components are to be prescribed on any part of the structure. They can be input as generalised loads. In addition, concentrated loads are applied at nodes in our analysis. The information for all these conditions is stored in the following arrays:

`NOFIX(NVFIX,1)`	Array storing number index of constrained nodes
`IFPRE(NVFIX,NNOFN)`	Array storing types of constraints, which is introduced to specify which degrees of freedom at a node are to be restrained = 0 means no constraint = 1 means constraint
`PRESC(NVFIX,NNOFN)`	Array of specified values of any degree of freedom
`LODPT(NPLOD,1)`	Array of number index of nodes at which concentrated loads along any degree of freedom are applied
`POINT(NPLOD,1)`	Array of specified values of concentrated load
`NEASS(NDLEG,1)`	Array of number index of element with loaded edge
`NOPRS(NDLEG,NODEG)`	Array of nodal index along loaded edge
`PRESS(NDLEG,NVEDG)`	Array of specified load intensity at each node on the loaded edge (`NVEDG=NODEG*NDOFN` represents total number of degrees of freedom of each loaded edge)

- Material properties

The material properties required for solution of a problem differ for different engineering applications. But as far as input of the necessary data is concerned, the same array can be employed to store properties of different materials in terms of material identification number and the relevant properties. For example, the array `PROPS(NMATS,NPROP)` represents material properties for all materials.

`PROPS(NMATS,NPROP)`	Array of material properties definition

The detailed format of the file storing the data mentioned above is shown in Appendix A. It should be pointed out that the action of reading data from a file is included in the main function in C programming, instead of as a separate subroutine. Because many pointer operations are frequently employed in C programming to define and manipulate arrays, it is not convenient to put this action into a separate subroutine.

4.2.2 MATLAB codes for input of data

```
function [NPOIN,NELEM,COORD,MATNO,LNODS,NVFIX,NOFIX,...
    IFPRE,PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,NOPRS,...
    PRESS,PROPS]=INPUTDT
% ** Input data from a file
% Input parameters: No
% Output parameters:
%   NPOIN: Number of nodes in domain
%   NELEM: Number of elements in domain
%   COORD: Coordinates of nodes
%   MATNO: Material index of each element
%   LNODS: Element connectivity
%   NVFIX: Number of boundary nodes at which specified
%          DOF is restricted
%   NOFIX: Global index of nodes at which specified DOF
%          is restricted
%   IFPRE: Types of constraints of each DOF
%   PRESC: Specified values
%   NPLOD: Number of concentrated loads
%   NDLEG: Number of loaded edges
%   LODPT: Global index of nodes at which concentrated
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   PROPS: Properties of materials
% *********************************
global NTREF NTYPE NDIME NDOFN NSTRE NNODE NODEG NEDGE;
global NMATS NPROP NGAUS;

% Open the input file
FILE1=input('Input data file name: ','s');
fp=fopen(FILE1,'r');
if fp<0
    error('-- Error: Can't open data file!');
end

dummy=char(zeros(1,100));  % fill with ASCII zeros
TITLE=char(zeros(1,200));  % fill with ASCII zeros

% Description of problem
dummy = fgets(fp);
```

```
TITLE = fgets(fp);
dummy = fgets(fp);
% Number of m-set and type definition
dummy = fgets(fp);
TMP = str2num(fgets(fp));
[NTREF,NTYPE]=deal(TMP(1),TMP(2));
% Element properties
dummy = fgets(fp);
TMP = str2num(fgets(fp));
[NNODE,NEDGE,NODEG]=deal(TMP(1),TMP(2),TMP(3));
% Number of Dimensions, DOF and stress components
dummy = fgets(fp);
TMP = str2num(fgets(fp));
[NDIME,NDOFN,NSTRE]=deal(TMP(1),TMP(2),TMP(3));
% Number of materials, material properties and Gaussian
% points
dummy = fgets(fp);
TMP = str2num(fgets(fp));
[NMATS,NPROP,NGAUS]=deal(TMP(1),TMP(2),TMP(3));
% Number of nodes, elements and boundary conditions
dummy = fgets(fp);
TMP = str2num(fgets(fp));
[NPOIN,NELEM,NVFIX,NPLOD,NDLEG]=...
    deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5));

% Read element connectivity and material numbers
MATNO=zeros(NELEM,1);
LNODS=zeros(NELEM,NNODE);
dummy = fgets(fp);
dummy = fgets(fp);
for iELEM=1:NELEM
    TMP=str2num(fgets(fp));
    [N,MATNO(iELEM),LNODS(iELEM,:)]=...
        deal(TMP(1),TMP(2),TMP(3:2+NNODE));
end

% Read nodal coordinates
COORD=zeros(NPOIN,NDIME);
dummy = fgets(fp);
dummy = fgets(fp);
for iPOIN=1:NPOIN
    TMP=str2num(fgets(fp));
    [N,COORD(iPOIN,:)]=deal(TMP(1),TMP(2:1+NDIME));
end
% Read essential boundary conditions
```

```
NOFIX=zeros(NVFIX,1);
IFPRE=zeros(NVFIX,NDOFN);
PRESC=zeros(NVFIX,NDOFN);
dummy = fgets(fp);
dummy = fgets(fp);
for iVFIX=1:NVFIX
    TMP=str2num(fgets(fp));
    [N,NOFIX(iVFIX),IFPRE(iVFIX,:),PRESC(iVFIX,:)]=...
        deal(TMP(1),TMP(2),TMP(3:2+NDOFN),...
        TMP(3+NDOFN:2+2*NDOFN));
end
% Read natural boundary conditions
LODPT=zeros(NPLOD,1);
POINT=zeros(NPLOD,NDOFN);
NEASS=zeros(NDLEG,1);
NOPRS=zeros(NDLEG,NODEG);
PRESS=zeros(NDLEG,NODEG*NDOFN);
% Read concentrated loads
dummy = fgets(fp);
if NPLOD>0
    dummy = fgets(fp);
    for iPLOD=1:NPLOD
        TMP=str2num(fgets(fp));
        [N,LODPT(iPLOD),POINT(iPLOD,:)]=...
            deal(TMP(1),TMP(2),TMP(3:2+NDOFN));
    end
end
% Read distributed edge loads
dummy = fgets(fp);
if NDLEG>0
    dummy = fgets(fp);
    for iDLEG=1: NDLEG
        TMP=str2num(fgets(fp));
        [N,NEASS(iDLEG),NOPRS(iDLEG,:),PRESS(iDLEG,:)]...
            =deal(TMP(1),TMP(2),TMP(3:2+NODEG),...
            TMP(3+NODEG:2+NODEG+NODEG*NDOFN));
    end
end
% Read material properties
PROPS=zeros(NMATS,NPROP);
dummy = fgets(fp);
dummy = fgets(fp);
for iMATS=1:NMATS
    TMP=str2num(fgets(fp));
    [N,PROPS(iMATS,:)]=deal(TMP(1),TMP(2:1+NPROP));
```

```
end fclose(fp);
```

4.2.3 C codes for input of data

```
void main()
{
    ...
    ...
    /** Input data from file **/
    puts("Input file name < dir:fn.txt >: ");
    gets(file);
    if((fp=fopen(file,"r"))==NULL)
    {
        printf("Warning! Can't open input file\n");
        exit(0);
    }
    // basic parameters
    fgets(dummy,200,fp);
    fgets(TITLE,200,fp);
    fgets(dummy,200,fp);

    fgets(dummy,200,fp);
    fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

    fgets(dummy,200,fp);
    fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

    fgets(dummy,200,fp);
    fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

    fgets(dummy,200,fp);
    fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

    fgets(dummy,200,fp);
    fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
        &NPLOD,&NDLEG);

    // element connectivity
    MATNO=(int *)calloc(NELEM,sizeof(int));
    ITCLEAN(NELEM,1,MATNO);
    LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
    ITCLEAN(NELEM,NNODE,LNODS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NELEM;i++)
```

```
{
    fscanf(fp,"%d %d",&N,&n1);
    MATNO[i]=n1-1;
    for(j=0;j<NNODE;j++)
    {
        fscanf(fp,"%d",&n2);
        LNODS[i*NNODE+j]=n2-1;
    }
    fscanf(fp,"\n");
}
// nodal coordinates
COORD=(double *)calloc(NPOIN*NDIME,sizeof(double));
DUCLEAN(NPOIN,NDIME,COORD);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NPOIN;i++)
{
    fscanf(fp,"%d",&N);
    for(j=0;j<NDIME;j++)
    {
        fscanf(fp,"%lf",&COORD[i*NDIME+j]);
    }
    fscanf(fp,"\n");
}
// specified nodal potential/displacement
NOFIX=(int *)calloc(NVFIX,sizeof(int));
ITCLEAN(NVFIX,1,NOFIX);
IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
ITCLEAN(NVFIX,NDOFN,IFPRE);
PRESC=(double *)calloc(NVFIX*NDOFN,sizeof(double));
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    NOFIX[i]=n1-1;
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
    }
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
    }
```

```
        fscanf(fp,"\n");
    }
    // specified concentrated loads at nodes
    fgets(dummy,200,fp);
    if(NPLOD>0)
    {
        LODPT=(int *)calloc(NPLOD*1,sizeof(int));
        ITCLEAN(NPLOD,1,LODPT);
        POINT=(double *)calloc(NPLOD*NDOFN,
            sizeof(double));
        DUCLEAN(NPLOD,NDOFN,POINT);
        fgets(dummy,200,fp);
        for(i=0;i<NPLOD;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            LODPT[i]=n1-1;
            for(j=0;j<NDOFN;j++)
            {
                fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
            }
            fscanf(fp,"\n");
        }
    }
    // specified distributed edge loads
    fgets(dummy,200,fp);
    if(NDLEG>0)
    {
        NEASS=(int *)calloc(NDLEG*1,sizeof(int));
        ITCLEAN(NDLEG,1,NEASS);
        NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
        ITCLEAN(NDLEG,NODEG,NOPRS);
        TNFEG=NODEG*NDOFN;
        PRESS=(double *)calloc(NDLEG*TNFEG,sizeof(double));
        DUCLEAN(NDLEG,TNFEG,PRESS);
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
```

```
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NPROP;j++)
        {
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }
    ...
    ...
    return;
}
```

4.2.4 Output of results

With regard to the output of computational results, a subroutine named `OPRESUT` is designed for outputting numerical results at both nodal points and central points of each element to a file for further reference and analysis. Since all integrals in HT-FEM are defined on the element boundary and the nodes are generally assigned along the boundaries of an element only, the stress results at these boundary nodes and Gaussian sampling points might be different if the calculation is based on different elements, and special treatments would be needed to obtain meaningful results. Detailed description of these treatments is presented in Chapter 9. Moreover, stress values at points inside an element, such as the centroids of all elements, are required in most practical problems. The results output from the subroutine `OPRESUT` include the following three categories:

- Basic parameters

Basic control parameters discussed here are the same as those presented in Section 4.2.1, which reflect roughly the features of the problems under consideration and the elements employed.

- Generalised displacement field at nodes

Nodal displacements can be directly obtained from the solution of final stiffness equations. It is therefore easy to output nodal generalised displacement to a file. In the computer program, the array UPOIN is used to store the results of nodal displacements:

UPOIN(NPOIN,NDOFN)	Array of nodal generalised displacements

- Generalised displacement and stress fields at centroid of each element

Once the nodal displacement fields have been determined, the corresponding coefficients **c** of the intra-element field in each element can be calculated. The generalised displacements and stresses at the centroid of each element can then be determined. The arrays used to store the results of centroid displacements and stresses in this program are:

UCENP(NELEM,NDOFN)	Array of generalised displacements at centroid of each element
SCENP(NELEM,NSTRE)	Array of generalised stresses at centroid of each element

4.2.5 MATLAB codes for output of results

```
function OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,...
    SCENP,NVFIX,NPLOD,NDLEG)
% ** Output of results to a file
% Input parameters:
%   NPOIN: Number of nodes in domain
%   COORD: Coordinates of nodes
%   UPOIN: Nodal displacement field
%   NELEM: Number of elements in domain
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
%   NVFIX: Number of boundary nodes at which specified
%          DOF is restricted
%   NPLOD: Number of concentrated loads
%   NDLEG: Number of loaded edges
% Output parameters: No
% ********************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;

fp=fopen('Result.txt','wt');

fprintf(fp,'**Basic parameters\n');
```

```
fprintf(fp,'\n');
fprintf(fp,'NTREF=%5d  NTYPE=%5d\n',NTREF,NTYPE);
fprintf(fp,'\n');
fprintf(fp,'NDIME=%5d  NDOFN=%5d  NSTRE=%5d\n',...
    NDIME,NDOFN,NSTRE);
fprintf(fp,'\n');
fprintf(fp,'NNODE=%5d  NEDGE=%5d  NODEG=%5d\n',...
    NNODE,NEDGE,NODEG);
fprintf(fp,'\n');
fprintf(fp,'NMATS=%5d  NPROP=%5d  NGAUS=%5d\n',...
    NMATS,NPROP,NGAUS);
fprintf(fp,'\n');
fprintf(fp,'NPOIN=%5d  NELEM=%5d  NVFIX=%5d\n',...
    NPOIN,NELEM,NVFIX);
fprintf(fp,'\n');
fprintf(fp,'NPLOD=%5d  NDLEG=%5d\n',NPLOD,NDLEG);
fprintf(fp,'\n'); fprintf(fp,'\n'); fprintf(fp,'** Nodal generalis
displacements\n');
fprintf(fp,'----------------------------------\n'); fprintf(fp,'Nu
COD#1-->#NDIME DIS#1-->#NODFN\n');
fprintf(fp,'----------------------------------\n'); for
iPOIN=1:NPOIN
    xp=COORD(iPOIN,1);
    yp=COORD(iPOIN,2);
    fprintf(fp,'%5d  %9.4f  %9.4f',iPOIN,xp,yp);
    for iDOFN=1:NDOFN
        fprintf(fp,'  %9.4f',UPOIN(iPOIN,iDOFN));
    end
    fprintf(fp,'\n');
end fprintf(fp,'\n'); fprintf(fp,'\n'); fprintf(fp,'**Central
generalised displacements\n');
fprintf(fp,'----------------------------------\n');
fprintf(fp,'Elem# COD#1-->#NDIME DIS#1-->#NODFN\n');
fprintf(fp,'----------------------------------\n'); for
iELEM=1:NELEM
    xp=CECOD(iELEM,1);
    yp=CECOD(iELEM,2);
    fprintf(fp,'%5d  %9.4f  %9.4f',iELEM,xp,yp);
    for iDOFN=1:NDOFN
        fprintf(fp,'  %9.4f',UCENP(iELEM,iDOFN));
    end
    fprintf(fp,'\n');
end fprintf(fp,'\n'); fprintf(fp,'** Central generalised
stresses\n'); fprintf(fp,'------------------------\n');
fprintf(fp,'Elem#    STRES#1-->#NSTRE\n');
```

```
fprintf(fp,'-------------------------\n'); for iELEM=1:NELEM
    fprintf(fp,'%5d',iELEM);
    for iSTRE=1:NSTRE
        fprintf(fp,'  %9.4f',SCENP(iELEM,iSTRE));
    end
    fprintf(fp,'\n');
end
fclose(fp);
```

4.2.6 C codes for output of results

```
/*
*********************************************************
* Subroutine OPRESUT                                    *
* - Output of numerical results to a file               *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void OPRESUT(int NPOIN,double COORD[],double UPOIN[],
             int NELEM,double CECOD[],double UCENP[],
             double SCENP[],int NVFIX,int NPLOD,
             int NDLEG)
{
    FILE *fp;
    extern int NTREF,NTYPE,NDIME,NDOFN,NSTRE,NNODE,
        NEDGE,NODEG,NMATS,NPROP,NGAUS;
    int iPOIN,iELEM,iDOFN,iSTRE,iDIME,n1;

    fp=fopen("Result.txt","w");
    fprintf(fp,"** Basic parameters\n");
    fprintf(fp,"\n");
    fprintf(fp,"NTREF=%5d  NTYPE=%5d\n",NTREF,NTYPE);
    fprintf(fp,"\n");
    fprintf(fp,"NDIME=%5d  NDOFN=%5d  NSTRE=%5d\n",
        NDIME,NDOFN,NSTRE);
    fprintf(fp,"\n");
    fprintf(fp,"NNODE=%5d  NEDGE=%5d  NODEG=%5d\n",
        NNODE,NEDGE,NODEG);
    fprintf(fp,"\n");
    fprintf(fp,"NMATS=%5d  NPROP=%5d  NGAUS=%5d\n",
        NMATS,NPROP,NGAUS);
    fprintf(fp,"\n");
    fprintf(fp,"NPOIN=%5d  NELEM=%5d  NVFIX=%5d\n",
```

```
    NPOIN,NELEM,NVFIX);
fprintf(fp,"\n");
fprintf(fp,"NPLOD=%5d  NDLEG=%5d\n",NPLOD,NDLEG);
fprintf(fp,"\n");
fprintf(fp,"** Nodal generalised displacements\n");
fprintf(fp,"----------------------------------\n");
fprintf(fp,"Num# COD#1-->#NDIME DIS#1-->#NDOFN\n");
fprintf(fp,"----------------------------------\n");
for(iPOIN=0;iPOIN<NPOIN;iPOIN++)
{
    fprintf(fp,"%5d",iPOIN);
    for(iDIME=0;iDIME<NDIME;iDIME++)
    {
        n1=iPOIN*NDIME+iDIME;
        fprintf(fp,"  %9.4f",COORD[n1]);
    }
    for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
    {
        n1=iPOIN*NDOFN+iDOFN;
        fprintf(fp,"  %9.4f",UPOIN[n1]);
    }
    fprintf(fp,"\n");
}
fprintf(fp,"\n");
fprintf(fp,"**Central generalised displacements\n");
fprintf(fp,"-----------------------------------\n");
fprintf(fp,"Elem# COD#1-->#NDIME DIS#1-->#NODFN\n");
fprintf(fp,"-----------------------------------\n");
for(iELEM=0;iELEM<NELEM;iELEM++)
{
    fprintf(fp,"%5d",iELEM);
    for(iDIME=0;iDIME<NDIME;iDIME++)
    {
        n1=iELEM*NDIME+iDIME;
        fprintf(fp,"  %9.4f",CECOD[n1]);
    }
    for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
    {
        n1=iELEM*NDOFN+iDOFN;
        fprintf(fp,"  %9.4f",UCENP[n1]);
    }
    fprintf(fp,"\n");
}
fprintf(fp,"\n");
fprintf(fp,"** Central generalised stresses\n");
```

```
    fprintf(fp,"----------------------\n");
    fprintf(fp,"Ele#  STRES#1-->#NSTRE\n");
    fprintf(fp,"----------------------\n");
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        fprintf(fp,"%5d",iELEM);
        for(iSTRE=0;iSTRE<NSTRE;iSTRE++)
        {
            n1=iELEM*NSTRE+iSTRE;
            fprintf(fp,"  %9.4f",SCENP[n1]);
        }
        fprintf(fp,"\n");
    }
    fclose(fp);
    return;
}
```

4.3 Numerical integration over element edges

Unlike in conventional FEM, evaluation of elemental stiffness matrix in HT-FEM is based only on the numerical integration along element edges. Therefore, it is useful to provide a commonly used subroutine for conducting numerical boundary integration. For this purpose, a guideline for numerical computation of element boundary integrals is presented in this and next sections. Since Gaussian numerical integration formulae are easy to use and very accurate in comparison with other methods, they are adopted in our program [1].

Because only two-dimensional problems are involved in this book, integration along the boundaries of a two-dimensional element has a one-dimensional feature. Therefore only the algorithm of one-dimensional Gaussian quadrature is considered, which is mathematically defined as

$$I = \int_{-1}^{1} f(\xi)\,\mathrm{d}\xi \approx \sum_{i=1}^{n} f(\xi_i)\, w_i \tag{4.1}$$

where ξ_i is the coordinate of the ith integration point, w_i is the associated weighting factor, and n is the total number of integration points ($1 < n < 9$ in our program).

In the following, the subroutine GAUSSQU is presented in both C and MATLAB for providing the coordinates of Gaussian sampling points and the corresponding weight coefficients.

4.3.1 MATLAB codes

```
function [POSGP,WEIGP]=GAUSSQU(NGAUS)
% ** Set up the Gauss-Legendre integration constants
% Input parameters:
%   NGAUS: Number of Gaussian samples
% Output parameters:
%   POSGP: Coordinates of Gaussian sample
%   WEIGP: Weight coefficient of Gaussian sample
% *********************************

POSGP=zeros(1,NGAUS);
WEIGP=zeros(1,NGAUS);

if (NGAUS<2)|(NGAUS>8)
    error('NGAUS exceeds the range [2,8]!');
end

switch NGAUS
    case 2
        POSGP(1) =-0.577350269189626;
        WEIGP(1) = 1.0;
    case 3
        POSGP(1) =-0.774596669241483;
        POSGP(2) = 0.0;
        WEIGP(1) = 0.555555555555556;
        WEIGP(2) = 0.888888888888889;
    case 4
        POSGP(1) =-0.861136311594053;
        POSGP(2) =-0.339981043584856;

        WEIGP(1) = 0.347854845137454;
        WEIGP(2) = 0.652145154862546;
    case 5
        POSGP(1) =-0.906179845938664;
        POSGP(2) =-0.538469310105683;
        POSGP(3) = 0.0;

        WEIGP(1) = 0.236926885056189;
        WEIGP(2) = 0.478628670499366;
        WEIGP(3) = 0.568888888888889;
    case 6
        POSGP(1) =-0.932469514203152;
        POSGP(2) =-0.661209386466265;
        POSGP(3) =-0.238619186083197;
```

```
        WEIGP(1) = 0.171324492379170;
        WEIGP(2) = 0.360761573048139;
        WEIGP(3) = 0.467913934572691;
    case 7
        POSGP(1) =-0.949107912342759;
        POSGP(2) =-0.741531185599394;
        POSGP(3) =-0.405845151377397;
        POSGP(4) = 0.0;

        WEIGP(1) = 0.129484966168870;
        WEIGP(2) = 0.279705391489277;
        WEIGP(3) = 0.381830050505119;
        WEIGP(4) = 0.417959183673469;
    case 8
        POSGP(1) =-0.960289856497536;
        POSGP(2) =-0.796666477413627;
        POSGP(3) =-0.525532409916329;
        POSGP(4) =-0.183434642495650;

        WEIGP(1) = 0.101228536290376;
        WEIGP(2) = 0.222381034453374;
        WEIGP(3) = 0.313706645877887;
        WEIGP(4) = 0.362683783378362;
end

kGAUS=fix(NGAUS/2);
for iGAUS=1:kGAUS
    jGAUS=NGAUS+1-iGAUS;
    POSGP(jGAUS)=-POSGP(iGAUS);
    WEIGP(jGAUS)= WEIGP(iGAUS);
end
```

4.3.2 C codes

```
/*
*********************************************************
* Subroutine GAUSSQU                                    *
* - Set up Gaussian integration constants               *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void GAUSSQU(double POSGP[],double WEIGP[])
```

```
{
    extern int NGAUS;
    int iGAUS,jGAUS,kGAUS;

    if((NGAUS<2)||(NGAUS>8))
    {
        printf("** NGAUS exceeds the range [2,8]!\n");
        exit(0);
    }
    switch(NGAUS)
    {
    case 2:
        POSGP[0] =-0.577350269189626;
        WEIGP[0] = 1.0;
        break;
    case 3:
        POSGP[0] =-0.774596669241483;
        POSGP[1] = 0.0;
        WEIGP[0] = 0.555555555555556;
        WEIGP[1] = 0.888888888888889;
        break;
    case 4:
        POSGP[0] =-0.861136311594053;
        POSGP[1] =-0.339981043584856;

        WEIGP[0] = 0.347854845137454;
        WEIGP[1] = 0.652145154862546;
        break;
    case 5:
        POSGP[0] =-0.906179845938664;
        POSGP[1] =-0.538469310105683;
        POSGP[2] = 0.0;

        WEIGP[0] = 0.236926885056189;
        WEIGP[1] = 0.478628670499366;
        WEIGP[2] = 0.568888888888889;
        break;
    case 6:
        POSGP[0] =-0.932469514203152;
        POSGP[1] =-0.661209386466265;
        POSGP[2] =-0.238619186083197;

        WEIGP[0] = 0.171324492379170;
        WEIGP[1] = 0.360761573048139;
        WEIGP[2] = 0.467913934572691;
```

```
        break;
    case 7:
        POSGP[0] =-0.949107912342759;
        POSGP[1] =-0.741531185599394;
        POSGP[2] =-0.405845151377397;
        POSGP[3] = 0.0;

        WEIGP[0] = 0.129484966168870;
        WEIGP[1] = 0.279705391489277;
        WEIGP[2] = 0.381830050505119;
        WEIGP[3] = 0.417959183673469;
        break;
    case 8:
        POSGP[0] =-0.960289856497536;
        POSGP[1] =-0.796666477413627;
        POSGP[2] =-0.525532409916329;
        POSGP[3] =-0.183434642495650;

        WEIGP[0] = 0.101228536290376;
        WEIGP[1] = 0.222381034453374;
        WEIGP[2] = 0.313706645877887;
        WEIGP[3] = 0.362683783378362;
        break;
    }

    // rounds the element toward zero
    kGAUS=(int)floor(NGAUS/2.0);
    for(iGAUS=0;iGAUS<kGAUS;iGAUS++)
    {
        jGAUS=NGAUS-1-iGAUS;
        POSGP[jGAUS]=-POSGP[iGAUS];
        WEIGP[jGAUS]= WEIGP[iGAUS];
    }
    return;
}
```

4.4 Shape functions along element edge

One of the advantages of HT-FEM over the conventional FEM is that arbitrary shaped elements can be created easily as the frame field is defined on an elemental boundary and the integration is also performed along the element boundary. As a result, the

shape functions defined on the element edge are easy to construct.

For two-dimensional problems, the shape functions can be expressed in terms of one independent variable only, say ξ. In this case, the interpolation equations of coordinates $\mathbf{x} = (x_1, x_2)$ and any field variable f at an edge of the element can be expressed as [2]

$$\mathbf{x} = \sum_{j=1}^{n} N_j(\xi)\,\mathbf{x}_j \tag{4.2}$$

$$f = \sum_{j=1}^{n} N_j(\xi)\,f_j \tag{4.3}$$

where $\mathbf{x}_j$ represents the coordinates of j^{th} node, $N_j(\xi)$ is the shape function, and n is the number of nodes on the edge.

Making use of the properties of the shape functions for a one-dimensional line element:

$$N_i(\xi_i) = 1 \tag{4.4}$$

$$N_j(\xi_i) = 0 \quad \text{for } i \neq j \tag{4.5}$$

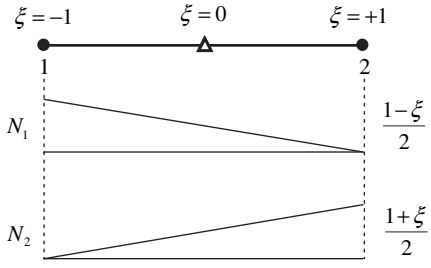
$$\sum_{j=1}^{n} N_j(\xi) = 1 \tag{4.6}$$

a series of shape functions with different nodes can be constructed. Figure 4.2 shows shape functions typical for an edge with 2, 3, and 4 nodes.

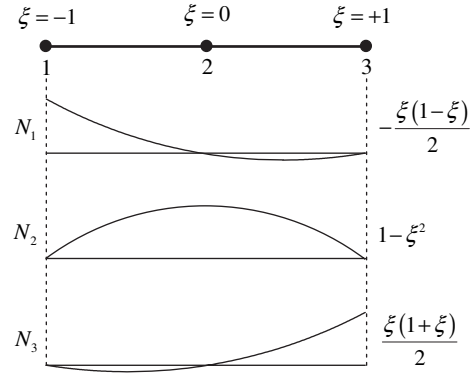
Although only edges with 2 and 3 nodes in a typical HT-FEM element are considered in our MATLAB and C programs, users can add their own modules straightforwardly with the subfunction `SHAPFUN` in 4.4.1 (or 4.4.2) to meet their special requirements.

4.4.1 MATLAB codes

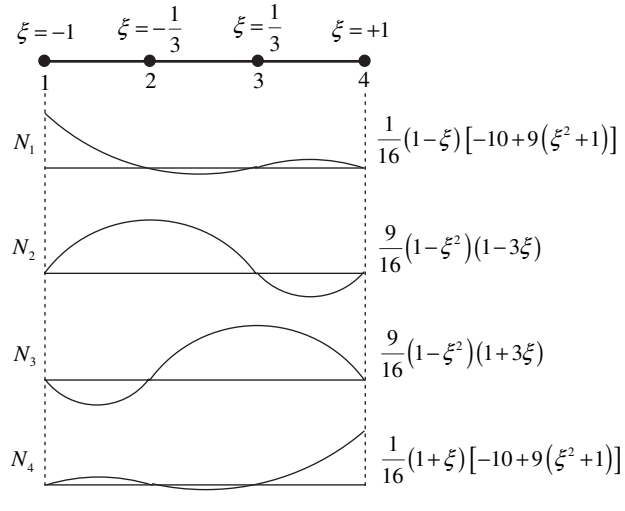
```
function [SHAPE,DSHAP]=SHAPFUN(EXISP)
% ** Shape functions and its derivatives for 1D line
%    element
% Input parameters:
%   EXISP: coordinates at Gaussian samples
% Output parameters:
%   SHAPE: vector of shape functions
%   DSHAP: vector of derivatives of shape functions
%          dN/ds
% *********************************
global NODEG;

if NODEG==2
    SH1 =(1-EXISP)/2;
    SH2 =(1+EXISP)/2;
```



(a) Edge with 2 nodes



(b) Edge with 3 nodes



(c) Edge with 4 nodes

FIGURE 4.2
Shape functions of an elemental edge with 2, 3, and 4 nodes

```
    DSH1=-0.5;
    DSH2= 0.5;

    SHAPE=[SH1,SH2];
    DSHAP=[DSH1,DSH2];
elseif NODEG==3
    SH1 =-EXISP*(1-EXISP)/2;
    SH2 = 1-EXISP^2;
    SH3 = EXISP*(1+EXISP)/2;
    DSH1=-0.5+EXISP;
    DSH2=-2*EXISP;
    DSH3= 0.5+EXISP;

    SHAPE=[SH1,SH2,SH3];
    DSHAP=[DSH1,DSH2,DSH3];
end
```

4.4.2 C codes

```
/*
*******************************************************
* Subroutine SHAPFUN                                   *
* - Shape functions and its derivatives for 1D line    *
*   element                                            *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void SHAPFUN(double EXISP,double SHAPE[],double DSHAP[])
{
    extern int NODEG;
    if(NODEG==2)
    {
        SHAPE[0]=(1-EXISP)/2.0;
        SHAPE[1]=(1+EXISP)/2.0;
        DSHAP[0]=-0.5;
        DSHAP[1]= 0.5;
    }
    else if(NODEG==3)
    {
        SHAPE[0]=-EXISP*(1-EXISP)/2.0;
        SHAPE[1]= 1.0-EXISP*EXISP;
        SHAPE[2]= EXISP*(1+EXISP)/2.0;
        DSHAP[0]=-0.5+EXISP;
```

```
        DSHAP[1]=-2.0*EXISP;
        DSHAP[2]= 0.5+EXISP;
    }
    else
    {
        printf("Exceed the maximum range of NODEG!");
        exit(0);
    }
    return;
}
```

4.5 Assembly of elements

The process of assembling elements to form global stiffness matrix in HT-FEM is just the same as that in conventional FEM [3]. The main idea of assembling an element matrix from a local into a global one is displayed in Figure 4.3, from which we can see that the entire process is independent of types of element. Thus it is suitable for any type of element. Consequently, we can design the corresponding programs in both MATLAB and C, as shown below.

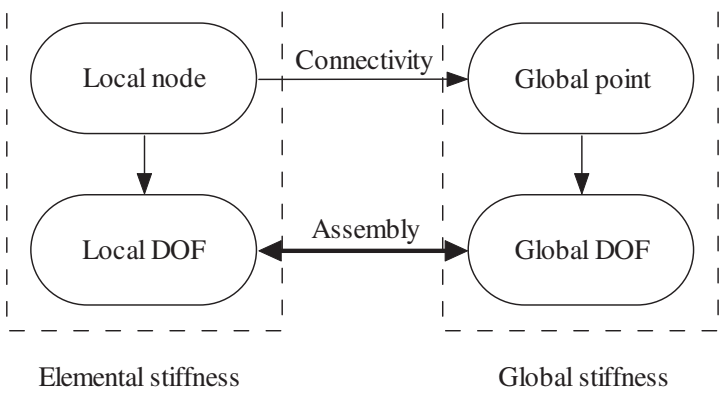


FIGURE 4.3
Illustration of assembly of elements

4.5.1 MATLAB codes

```
function [GSTIF]=ASMSTIF(iELEM,LNODS,ESTIF,GSTIF)
% ** Assemble element stiffness to global stiffness
% Input parameters:
%   iELEM: Element index
%   LNODS: Elementary connectivity
%   ESTIF: Element stiffness
%   GSTIF: Global stiffness
% Output parameters:
%   GSTIF: Global stiffness
% Note:
%   1. Assemble process can be used to other problems
%   directly
%   2. Full matrix is stored instead of bandwidth form
% *********************************
global NDOFN NNODE;

for iNODE=1:NNODE
    iPOIN=LNODS(iELEM,iNODE);
    for iDOFN=1:NDOFN
        %global row location
        iGR=NDOFN*(iPOIN-1)+iDOFN;
        %local row location
        iLR=NDOFN*(iNODE-1)+iDOFN;
        for jNODE=1:NNODE
            jPOIN=LNODS(iELEM,jNODE);
            for jDOFN=1:NDOFN
                % global column location
                jGC=NDOFN*(jPOIN-1)+jDOFN;
                % local column location
                jLC=NDOFN*(jNODE-1)+jDOFN;
                GSTIF(iGR,jGC)=GSTIF(iGR,jGC)+...
                    ESTIF(iLR,jLC);
            end
        end
    end
end
clear ESTIF;
```

4.5.2 C codes

```
/*
*******************************************************
* Subroutine ASMSTIF                                  *
```

```
* - Element stiffness assemble                          *
*********************************************************
Note: 1. Assemble process can be used to other problems
   directly
2. Full matrix is stored rather than bandwidth form
*/
void ASMSTIF(int iELEM,int NEQNS,int LNODS[],
             double ESTIF[],double GSTIF[])
{
    extern int NDOFN,NNODE;
    int ii,jj,it,jt,n1,n2,iGR,iLR,jGC,jLC,ip,jp,NEVAB;
    NEVAB=NNODE*NDOFN;

    for(ii=0;ii<NNODE;ii++)
    {
        ip=LNODS[iELEM*NNODE+ii];
        for(it=0;it<NDOFN;it++)
        {
            iGR=NDOFN*ip+it;
            iLR=NDOFN*ii+it;
            for(jj=0;jj<NNODE;jj++)
            {
                jp=LNODS[iELEM*NNODE+jj];
                for(jt=0;jt<NDOFN;jt++)
                {
                    jGC=NDOFN*jp+jt;
                    jLC=NDOFN*jj+jt;
                    n1=iGR*NEQNS+jGC;
                    n2=iLR*NEVAB+jLC;
                    GSTIF[n1]=GSTIF[n1]+ESTIF[n2];
                }
            }
        }
    }
    return;
}
```

4.6 Introduction of essential boundary conditions

In general, after the process of assembly of element stiffness matrix and evaluating the equivalent nodal loads in HT-FEM, the subsequent global stiffness equations are

obtained as

$$\mathbf{Kd} = \mathbf{P} \tag{4.7}$$

where $\mathbf{K}$ denotes the global stiffness matrix with $n \times n$ components, $\mathbf{d}$ represents the global degree of freedom vector, and $\mathbf{P}$ is the equivalent nodal load vector.

It is well known that for any physical problem, appropriate boundary conditions must be specified; otherwise, the structure will be free to experience an amount of rigid body motion. Mathematically speaking, if boundary constraints are not imposed, then the global stiffness matrix $\mathbf{K}$ will be singular; that is, it cannot be inverted.

Several approaches to introduce specified boundary conditions have been presented in the literature. They include the elimination procedure and the penalty approach. Among these two approaches, the simplest is the penalty approach, which is achieved by adding a large number (or penalty term), say 10^{20}, to the leading diagonal of the stiffness matrix in the row in which the prescribed value is required, while the term in the same row of the right-hand side equivalent nodal load vector is then set to be the prescribed value multiplied by the augmented stiffness coefficient [4]. For example, suppose the condition at the degree of freedom d_5 is known to be $\bar{d}_5$, then the unconstrained set of stiffness equations (4.7) is modified in such a way that the term $K_{5,5}$ augmented by adding 10^{20} (sometimes we call it a penalty parameter). In the subsequent solution, there would be an equation as follows:

$$K_{5,1}d_1 + \cdots + \left(K_{5,5} + 10^{20}\right) d_5 + \cdots + K_{5,n}d_n = \bar{d}_5 \left(K_{5,5} + 10^{20}\right) \tag{4.8}$$

which would have the approximate effect of making $d_5 = \bar{d}_5$, since the terms $\sum_{i=1, i\neq 5}^{n} K_{5,i}d_i$ are sufficiently small relative to the larger diagonal term $\left(K_{5,5} + 10^{20}\right) d_5$ and can be ignored in practical computation.

There is another penalty approach for dealing with the constraint [3]. With this method, Eq. (4.8) is modified as

$$K_{5,1}d_1 + \cdots + \left(K_{5,5} + 10^{20}\right) d_5 + \cdots + K_{5,n}d_n = p_5 + 10^{20}\bar{d}_5 \tag{4.9}$$

in which the implied equilibrium relation

$$\sum_{i=1}^{n} K_{5,i}d_i = p_5 \tag{4.10}$$

is used to simplify Eq. (4.9) as

$$10^{20}d_5 = 10^{20}\bar{d}_5 \tag{4.11}$$

Compared to the penalty approach shown in Eq. (4.8), the approach (4.9) seems more resonable and accurate. Additionally, the large number 10^{20} can be replaced by a relatively smaller number in most practical problems. In our application, the penalty parameter is chosen to be

$$\max_{i,j=1\rightarrow n} \left(\left|\mathbf{K}_{i,j}\right|\right) \times 10^6 \tag{4.12}$$

and the subroutine INDISBC in MATLAB and C is designed to fulfill the treatment of geometrical boundary conditions.

4.6.1 MATLAB codes

```
function [GSTIF,GLOAD]=INDISBC(NEQNS,NVFIX,NOFIX,...
    IFPRE,PRESC,GSTIF,GLOAD)
% ** Modify global stiffness matrix GSTIF and
%    equivalent load vector GLOAD by the penalty
%    approach
% Input parameters:
%   NEQNS: Total number of equations
%   NVFIX: Number of boundary nodes at which specified
%          DOF is restricted
%   NOFIX: Global index of nodes at which specified DOF
%          is restricted
%   IFPRE: Types of constraints of each DOF
%   PRESC: Specified values
%   GSTIF: Global stiffness matrix
%   GLOAD: Global equivalent nodal load vector
% Output parameters:
%   GSTIF: Modified global stiffness matrix
%   GLOAD: Modified global equivalent nodal load vector
% *********************************
global NDOFN;

% Determine penalty parameter CNST
CNST=max(max(abs(GSTIF)));
if CNST+1==1
    error('**Singular stiffness matrix GSTIF');
end
CNST=CNST*1000000;
% Modify GSTIF and GLOAD for specified nodal
% displacements
for iVFIX=1:NVFIX
    kPOIN=NOFIX(iVFIX);
    for iDOFN=1:NDOFN
        iGR=(kPOIN-1)*NDOFN+iDOFN;
        ii=IFPRE(iVFIX,iDOFN);
        % 1: a constrained DOF
        % 0: no constrained DOF
        if ii==1
            disv=PRESC(iVFIX,iDOFN);
            GSTIF(iGR,iGR)=GSTIF(iGR,iGR)+CNST;
            GLOAD(iGR)=GLOAD(iGR)+CNST*disv;
```

```
        end
    end
end
```

4.6.2 C codes

```
/*
*********************************************************
* Subroutine INDISBC                                    *
* - Introduce displacement constraints by the penalty *
*   approach                                            *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void INDISBC(int NEQNS,int NVFIX,int NOFIX[],int IFPRE[],
             double PRESC[],double GSTIF[],double GLOAD[])
{
    extern int NDOFN;
    int ii,jj,kp,iGR,n1;
    double temp,CNST,disv;
    // Decide penalty parameter CNST
    CNST=0.0;
    for(ii=0;ii<NEQNS;ii++)
    {
        for(jj=0;jj<NEQNS;jj++)
        {
            temp=fabs(GSTIF[ii*NEQNS+jj]);
            if(temp>CNST)
            {
                CNST=temp;
            }
        }
    }
    if((CNST+1)==1)
    {
        printf("Singular coefficient matrix GSTIF!");
        exit(0);
    }
    CNST=CNST*1000000.0;
    // Modify GSTIF and GLOAD for specified nodal
    // displacements
    for(ii=0;ii<NVFIX;ii++)
    {
```

```
        kp=NOFIX[ii];
        for(jj=0;jj<NDOFN;jj++)
        {
            iGR=kp*NDOFN+jj;
            // 1 indicates a constrained DOF
            if(IFPRE[ii*NDOFN+jj]==1)
            {
                disv=PRESC[ii*NDOFN+jj];
                n1=iGR*NEQNS+iGR;
                GSTIF[n1]=GSTIF[n1]+CNST;
                GLOAD[iGR]=GLOAD[iGR]+CNST*disv;
            }
        }
    }
    return;
}
```

4.7 Solution of global stiffness equation

After introduction of the geometrical boundary conditions, the global stiffness matrix becomes nonsingular and the related linear stiffness equations can now be solved directly. In this section, the solution technique for the stiffness matrix equation is discussed.

Generally, the final form of HT-FEM linear algebraic equations is written as

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.13}$$

or

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{4.14}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{x}$ the unknown vector and $\mathbf{b}$ the right-hand side vector. n is the number of equations, which is also the number of unknowns. How to exactly solve the linear system of Eq. (4.13) for obtaining all unknowns is important in HT-FEM.

From the literature [1, 5, 6], the available methods for solving Eq. (4.13) are Gaussian elimination, LU decomposition, and the conjugate gradient method. In our program, the Gaussian elimination technique is used. It should be mentioned that there are plentiful built-in functions in MATLAB to deal with matrix operations. In the MATLAB system, the solution process is completed by simply using the command-left division "\" [5], whereas in C, we must write a long list of computer code to

solve Eq. (4.13). Here, the useful Gaussian elimination with backsubstitution [1, 3] is introduced for C programming.

In the procedure of Gaussian elimination, the key idea is to convert the augmented matrix

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} & b_1 \\ A_{21} & A_{22} & \cdots & A_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} & b_n \end{bmatrix} \tag{4.15}$$

into an upper triangular matrix

$$\begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} & \cdots & \tilde{A}_{1n} & \tilde{b}_1 \\ 0 & \tilde{A}_{22} & \cdots & \tilde{A}_{2n} & \tilde{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \tilde{A}_{nn} & \tilde{b}_n \end{bmatrix} \tag{4.16}$$

by performing elementary row operations with full pivoting.

As a consequence, the system (4.14) can be rewritten as

$$\begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} & \cdots & \tilde{A}_{1n} \\ 0 & \tilde{A}_{22} & \cdots & \tilde{A}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \tilde{A}_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \vdots \\ \tilde{b}_n \end{bmatrix} \tag{4.17}$$

Then all unknowns can be solved in inverse order by a procedure called backsubstitution, defined by

$$x_i = \frac{1}{\tilde{A}_{ii}} \left[\tilde{b}_i - \sum_{j=i+1}^{n} \tilde{A}_{ij} x_j \right] \tag{4.18}$$

In addition, before solving Eq. (4.13) the treatment of equilibration [6] is employed to optimise the matrix of the system. The main idea of this treatment is that every row of the system is divided by a maximum absolute value of the elements in this row. As an example, considering the *i*th row of the system (4.13)

$$A_{i,1}x_1 + A_{i,2}x_2 + \cdots + A_{i,n}x_n = b_i \tag{4.19}$$

If the maximum number is chosen as $A_{i,n}$, we can change Eq. (4.19) to

$$\frac{A_{i,1}}{A_{i,n}}x_1 + \frac{A_{i,2}}{A_{i,n}}x_2 + \cdots + x_n = \frac{b_i}{A_{i,n}} \tag{4.20}$$

In this section, the common subroutine LSESOVR is designed to solve the final linear system of equations (4.13) including the modules of equilibration treatment and final solution by means of Gaussian elimination (in C) and built-in function (in MATLAB).

4.7.1 MATLAB codes

```
function [x]=LSSOLVR(A,b,n)
% ** Solve Linear system of equations Ax=b
% Input parameters:
%   A: coefficient matrix
%   b: right-handed vector
%   n: the number of equations
% Output parameters:
%   x: solution of Ax=b
% ********************************

% Equilibration treatment
[A,b]=EQUIL(A,b,n);
% Condition number
condnum=cond(A);
% MATLAB Command
x=A\b;
% Residual norm
resnorm=norm(A*x-b);
disp(['Condition number = ',num2str(condnum)]);
disp(['Residual  norm   = ',num2str(resnorm)]);


% ------------------------------------------
function [A,b]=EQUIL(A,b,n)
% Equilibration treatment of Ax=b
% ********************************
% Determine maximum elements in each row and store them
% in vector temp
temp=(max(abs(A')))'; if min(temp)+1==1
    error('**Warning: Matrix is singular, check it!');
end
% Elements in each row divide the maximum element
for ii=1:n
    A(ii,:)=A(ii,:)/temp(ii);
end b=b./temp;
```

4.7.2 C codes

```
/*
*******************************************************
* Subroutine LSSOLVR, EQUIL                           *
* - Gaussian Elimination with column pivoting for     *
*   solving Ax=b                                      *
```

```
**********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void LSSOLVR(double a[],double b[],int n)
{
    void EQUIL();
    int i,j,k,io,p;
    double d,t;
    // Equilibrium treatment
    EQUIL(a,b,n);
    // Forward Elimination
    for(k=0;k<=n-2;k++)
    {
        d=0.0;
        //select column pivot
        for(i=k;i<n;i++)
        {
            t=a[i*n+k];
            if(fabs(t)>d)
            {
                d=t;
                io=i;
            }
        }
        if(fabs(d)+1==1)
        {
            printf("%dth column is zero,Fail!\n",k);
            exit(0);
        }
        if(io!=k)
        {
            for(j=k;j<n;j++)
            {
                t=a[k*n+j];
                a[k*n+j]=a[io*n+j];
                a[io*n+j]=t;
            }
            t=b[k];
            b[k]=b[io];
            b[io]=t;
        }
        d=a[k*n+k];
        for(j=k+1;j<=n-1;j++)
```

```
        {
            p=k*n+j;
            a[p]=a[p]/d;
        }
        b[k]=b[k]/d;
        for(i=k+1;i<=n-1;i++)
        {
            for(j=k+1;j<=n-1;j++)
            {
                p=i*n+j;
                a[p]=a[p]-a[i*n+k]*a[k*n+j];
            }
            b[i]=b[i]-a[i*n+k]*b[k];
        }
    }
    d=a[(n-1)*n+n-1];
    if(fabs(d)+1==1)
    {
        printf("Matrix is singular,Fail!\n");
        exit(0);
    }
    b[n-1]=b[n-1]/d;
    for(i=n-2;i>=0;i--)
    {
        t=0.0;
        for(j=i+1;j<=n-1;j++)
        {
            t=t+a[i*n+j]*b[j];
        }
        b[i]=b[i]-t;
    }
    return;
}

//-------------------------------------
// Equilibrium treatment
void EQUIL(double a[],double b[],int n)
{
    int i,j;
    double temp;
    for(i=0;i<n;i++)
    {
        temp=0.0;
        for(j=0;j<n;j++)
        {
```

```
            if(temp<fabs(a[i*n+j]))
            {
                temp=fabs(a[i*n+j]);
            }
        }
        for(j=0;j<n;j++)
        {
            a[i*n+j]=a[i*n+j]/temp;
        }
        b[i]=b[i]/temp;
    }
    return;
}
```

References

[1] William HP, Saul AT, William TV, Brian PF (1992), *Numerical Recipes in C: the art of scientific computing* (2nd edition). Cambridge: Cambridge University Press.

[2] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*. Southampton: WIT Press.

[3] Chandrupatla TR and Belegundu AD (2002), *Introduction to Finite Elements in Engineering* (3rd edition). N. J.: Prentice Hall.

[4] Smith IM, Griffiths DV (1988), *Programming the Finite Element Method* (2nd edition). John Wiley & Sons Inc.

[5] Kiusalaas J (2005), *Numerical Methods in Engineering with MATLAB*. New York: Cambridge University Press.

[6] Mathews JH, Fink KD (2004), *Numerical Methods Using MATLAB* (4th edition). N. J.: Pearson Prentice Hall.

5

Potential problems

5.1 Introduction

The Laplace or Poisson's equation and related boundary conditions have a wide range of applications to problems in practical engineering. In fact, many problems in mechanics and physics can finally be reduced to boundary value problems (BVPs) governed by the Laplace or Poisson's equation. For example, the equilibrium of an elastic membrane can be reduced to the standard Poisson equation; stable heat conduction in isotropic and homogeneous media is governed by the standard Laplace or Poisson's equation. Moreover, incompressible potential flow, static electromagnetic fields, and so on, can also be governed by the standard Laplace or Poisson equations. These problems have totally different physical backgrounds, but have the same mathematical formulation [1 - 3]. Generally, a solution which satisfies the Laplace equation is said to be harmonic and is a scalar field.

Historically, the application of integral equations to formulate the fundamental boundary-value problems of potential theory dates back to 1903 when Fredholm [4] demonstrated the existence of solutions to such equations, on the basis of a discretisation procedure. Due to the difficulty of finding analytical solutions, the use of integral equations has, to a great extent, been limited to theoretical investigations of existence and uniqueness in solutions of problems of mathematical physics. However, the appearance of computers has made it possible to implement discretisation procedures analytically and has enabled numerical solutions to be readily achieved.

Applying T-complete solution functions, Zielinski and Zienkiewicz [5] presented a solution technique in which the boundary solution over subdomains is linked by least square procedures without an auxiliary frame. Cheung et al. [6, 7] developed a set of indirect and direct formulations using T-complete functions for Poisson and Helmholtz problems. Jirousek and his co-workers [8, 9] studied an alternative method, the "frameless" T-element approach based on the application of a suitably truncated T-complete set of Trefftz functions, over individual subdomains linked by means of a least square procedure, and applied it to the Poisson's equation. Stojek [10] extended his work to the case of the Helmholtz equation. Recently, Wang et al. [11] applied the HT-FEM to nonlinear minimal surface problems by combination use of HT-FEM, radial basis functions and the analog equation method (AEM).

In the present chapter, the application is developed of HT-FEM to the solution of classical Laplace problems, whose equations are briefly reviewed in Section 5.2,

and the derivation of formulation and programming implementation are discussed in detail in Sections 5.3 - 5.6. The corresponding MATLAB and C codes are provided in Sections 5.7 and 5.8. Finally, some numerical examples are considered, to demonstrate the accuracy and performance of the HT FE models presented in this chapter.

5.2 Basic equations of potential problems

Consider that we are seeking the solution of a Laplace equation defined in a plane domain Ω (Figure 5.1)

$$\nabla^2 u(\mathbf{x}) = 0 \qquad \forall \mathbf{x} \in \Omega \tag{5.1}$$

and with the following boundary conditions:

-Dirichlet boundary condition related to unknown potential field

$$u = \bar{u} \qquad \text{on } \Gamma_u \tag{5.2}$$

-Neumann boundary condition for the boundary normal gradient

$$q = \frac{\partial u}{\partial n} = \bar{q} \qquad \text{on } \Gamma_q \tag{5.3}$$

where

$$\nabla^2 = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} \tag{5.4}$$

denotes Laplace operator, u is the field variable sought and q represents the boundary flux. n is the outward normal to the boundary $\Gamma = \Gamma_u \cup \Gamma_q$, and $\bar{u}$ and $\bar{q}$ are specified functions on the related boundaries, respectively.

For the sake of convenience, the expression (5.3) is rewritten in matrix form as

$$q = \mathbf{A} \begin{bmatrix} \dfrac{\partial u}{\partial x_1} \\ \dfrac{\partial u}{\partial x_2} \end{bmatrix} = \bar{q} \tag{5.5}$$

with

$$\mathbf{A} = [n_1 \quad n_2] \tag{5.6}$$

It should be mentioned that Eq. (5.1) is a homogeneous equation. For the case of an inhomogeneous problem with a nonzero right-hand side in Eq. (5.1), known as a Poisson problem, the treatment of inhomogeneous terms will be discussed in Chapter 7.

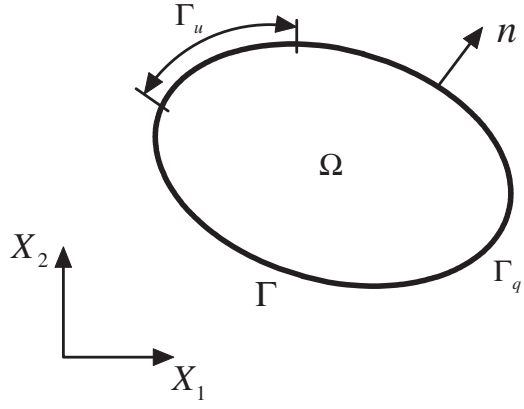


FIGURE 5.1
Geometrical definitions for the Laplace problem

5.3 Trefftz FE formulation

To perform FE analysis, as in the conventional FEM, the domain under consideration is divided into a series of elements. For each element in the HT-FEM, two independent fields are assumed in the following way [12] as discussed in the next two subsections.

5.3.1 Non-conforming intra-element field

The intra-element potential

$$u_e(\mathbf{x}) = \sum_{j=1}^{m} N_j(\mathbf{x})\, c_{ej} = \mathbf{N}_e(\mathbf{x})\, \mathbf{c}_e \tag{5.7}$$

is defined in Ω_e (see Figure 5.2), where $\mathbf{c}_e = \{c_1 \;\; c_2 \;\; \cdots \;\; c_m\}^{\mathrm{T}}$ is a vector of undetermined coefficients and m is its number of components, which is also the number of truncated T-complete functions N_j consisting of the row vector $\mathbf{N}_e(\mathbf{x}) = \{N_1 \;\; N_2 \;\; \cdots \;\; N_m\}$ satisfying

$$\nabla^2 N_j = 0 \tag{5.8}$$

The corresponding outward normal derivative of u_e on Γ_e is

$$q_e = \frac{\partial u_e}{\partial n} = \mathbf{Q}_e \mathbf{c}_e \tag{5.9}$$

where

$$\mathbf{Q}_e = \frac{\partial \mathbf{N}_e}{\partial n} = \mathbf{A}\mathbf{T}_e \tag{5.10}$$

with

$$\mathbf{T}_e = \begin{bmatrix} \dfrac{\partial \mathbf{N}_e}{\partial x_1} \\ \dfrac{\partial \mathbf{N}_e}{\partial x_2} \end{bmatrix} \tag{5.11}$$

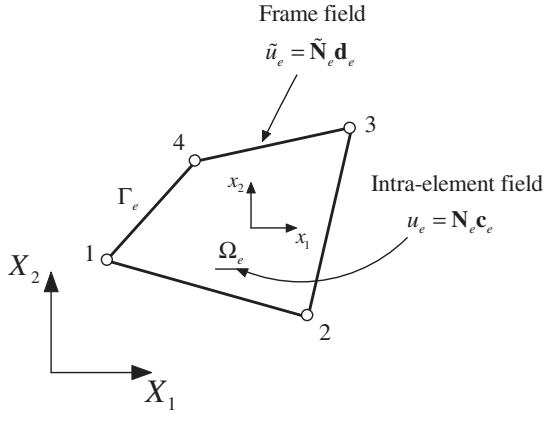


FIGURE 5.2
Intra-element field and frame field in a particular element e

5.3.2 Auxiliary conforming frame field

To enforce the inter-element conformity on u, that is, to ensure $u_e = u_f$ on $\Gamma_e \cap \Gamma_f$ of any two neighboring elements, we use an auxiliary inter-element frame field $\tilde{u}$ expressed in terms of the same degrees of freedom (DOF), $\mathbf{d}$, as used with conventional elements (see Figure 5.2). In this case, $\tilde{u}$ is confined to the whole element boundary only, that is,

$$\tilde{u}_e(\mathbf{x}) = \tilde{\mathbf{N}}_e(\mathbf{x})\,\mathbf{d}_e = \begin{bmatrix} \tilde{N}_{e1} & \tilde{N}_{e2} & \cdots & \tilde{N}_{en} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \tag{5.12}$$

which is independently assumed along the element boundary in terms of nodal DOF $\mathbf{d}_e$, where $\tilde{\mathbf{N}}_e$ represents the conventional finite element interpolating shape functions. For example, a simple interpolation of the frame field on the side 2-3 of a

4-node linear element (Figure 5.3) can be given in the form

$$\tilde{u}_{23} = \hat{N}_1 u_2 + \hat{N}_2 u_3 = \begin{bmatrix} 0 & \hat{N}_1 & \hat{N}_2 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \tag{5.13}$$

where

$$\hat{N}_1 = \frac{1-\xi}{2}, \qquad \hat{N}_2 = \frac{1+\xi}{2} \tag{5.14}$$

This means that on the side 2-3 the shape function $\tilde{\mathbf{N}}_e$ can be expressed as

$$\tilde{\mathbf{N}}_e = \begin{bmatrix} 0 & \hat{N}_1 & \hat{N}_2 & 0 \end{bmatrix} \tag{5.15}$$

Similarly, the shape function $\tilde{\mathbf{N}}_e$ on other sides can be written as

$$\begin{aligned} \tilde{\mathbf{N}}_e\big|_{\text{side}12} &= \begin{bmatrix} \hat{N}_1 & \hat{N}_2 & 0 & 0 \end{bmatrix} \\ \tilde{\mathbf{N}}_e\big|_{\text{side}34} &= \begin{bmatrix} 0 & 0 & \hat{N}_1 & \hat{N}_2 \end{bmatrix} \\ \tilde{\mathbf{N}}_e\big|_{\text{side}41} &= \begin{bmatrix} \hat{N}_2 & 0 & 0 & \hat{N}_1 \end{bmatrix} \end{aligned} \tag{5.16}$$

To make the derivation tractable, introduce a new function $\beta_{ij}(\xi)$ such that

$$\beta_{ij}(\xi) = \begin{cases} 1 & \text{when } \xi \in \text{side}ij \\ 0 & \text{otherwise} \end{cases} \tag{5.17}$$

As a result, we can express the shape function $\tilde{\mathbf{N}}_e$ defined on the entire element boundary in a compact form:

$$\tilde{\mathbf{N}}_e = \begin{bmatrix} \hat{N}_1\beta_{12} + \hat{N}_2\beta_{41} & \hat{N}_1\beta_{23} + \hat{N}_2\beta_{12} & \hat{N}_1\beta_{34} + \hat{N}_2\beta_{23} & \hat{N}_1\beta_{41} + \hat{N}_2\beta_{34} \end{bmatrix} \tag{5.18}$$

It is worth pointing out that the vector $\tilde{\mathbf{N}}_e$ has a different form for different edges [see from Eq. (5.13)]. Understanding this fact is helpful for subsequent numerical integration.

5.3.3 Modified variational principle

Following the approach described in Ref. [12], the variational functional Ψ_{me} corresponding to a particular three-dimensional element e can be written as

$$\begin{aligned} \Psi_{me} = \frac{1}{2}\int_{V_e} \left(q_1^2 + q_2^2 + q_3^2\right) \mathrm{d}V - \int_{S_{eu}} q\bar{u}\mathrm{d}S \\ + \int_{S_{eq}} (\bar{q} - q)\,\tilde{u}\mathrm{d}S - \int_{S_{el}} q\tilde{u}\mathrm{d}S \end{aligned} \tag{5.19}$$

where $q_i = \frac{\partial u}{\partial x_i}$ is the potential flow in x_i direction, V_e represents the volume of the domain occupied by the element e, and $S_e = \partial V_e$ denotes the corresponding boundary.

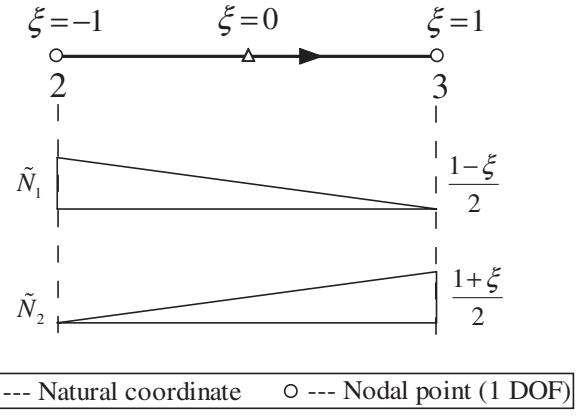


FIGURE 5.3
Typical linear interpolation for frame field

S_{eI} is the inter-element boundary of the element e. $S_e = S_{eu} + S_{eq} + S_{eI}$, $S_{eu} = S_e \cap S_u$, $S_{eq} = S_e \cap S_q$.

For two-dimensional problems considered in the remainder of this chapter, Eq. (5.19) is reduced to the following form

$$\Psi_{me} = \frac{1}{2}\int_{\Omega_e} \left(q_1^2 + q_2^2\right) t_e \mathrm{d}\Omega - \int_{\Gamma_{eu}} q\bar{u} t_e \mathrm{d}\Gamma + \int_{\Gamma_{eq}} (\bar{q} - q)\,\tilde{u} t_e \mathrm{d}\Gamma - \int_{\Gamma_{eI}} q\tilde{u} t_e \mathrm{d}\Gamma \quad (5.20)$$

where t_e denotes the thickness of the element e, Ω_e and Γ_e are shown in Figure 5.2, and Γ_{eI} is the inter-element boundary of the element e. $\Gamma_e = \Gamma_{eu} + \Gamma_{eq} + \Gamma_{eI}$, $\Gamma_{eu} = \Gamma_e \cap \Gamma_u$, $\Gamma_{eq} = \Gamma_e \cap \Gamma_q$.

Considering the fact that $\Gamma_e = \Gamma_{eu} + \Gamma_{eq} + \Gamma_{eI}$ in the above equation and $\bar{u} = \tilde{u}$ on Γ_{eu}, we rewrite Eq. (5.20) in a simpler form

$$\Psi_{me} = \frac{1}{2}\int_{\Omega_e} \left(q_1^2 + q_2^2\right) t_e \mathrm{d}\Omega - \int_{\Gamma_e} q\tilde{u} t_e \mathrm{d}\Gamma + \int_{\Gamma_{eq}} \bar{q}\tilde{u} t_e \mathrm{d}\Gamma \quad (5.21)$$

Integrating the domain integral in Eq. (5.21) by parts gives

$$\begin{aligned}\int_{\Omega_e} \left(q_1^2 + q_2^2\right) t_e \mathrm{d}\Omega &= \int_{\Gamma_e} \left[\left(u\frac{\partial u}{\partial x} n_x + u\frac{\partial u}{\partial y} n_y\right)\right] t_e \mathrm{d}\Gamma - \int_{\Omega_e} \left(\nabla^2 u\right) u t_e \mathrm{d}\Omega \\ &= \int_{\Gamma_e} u\frac{\partial u}{\partial n} t_e \mathrm{d}\Gamma = \int_{\Gamma_e} uqt_e \mathrm{d}\Gamma \end{aligned} \quad (5.22)$$

As a result, the functional (5.21) can convert into one containing boundary integrals only

$$\Psi_{me} = \frac{1}{2}\int_{\Gamma_e} qut_e \mathrm{d}\Gamma - \int_{\Gamma_e} q\tilde{u} t_e \mathrm{d}\Gamma + \int_{\Gamma_{eq}} \bar{q}\tilde{u} t_e \mathrm{d}\Gamma \quad (5.23)$$

Then, substituting Eqs. (5.7), (5.9) and (5.12) into the functional (5.23) produces

$$\Psi_{me} = \frac{1}{2}\mathbf{c}_e^{\mathrm{T}}\mathbf{H}_e\mathbf{c}_e - \mathbf{c}_e^{\mathrm{T}}\mathbf{G}_e\mathbf{d}_e + \mathbf{d}_e^{\mathrm{T}}\mathbf{g}_e \tag{5.24}$$

in which

$$\mathbf{H}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathrm{T}}\mathbf{N}_e t_e \mathrm{d}\Gamma \tag{5.25}$$

$$\mathbf{G}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathrm{T}}\tilde{\mathbf{N}}_e t_e \mathrm{d}\Gamma \tag{5.26}$$

$$\mathbf{g}_e = \int_{\Gamma_{eq}} \tilde{\mathbf{N}}_e^{\mathrm{T}}\bar{q}t_e \mathrm{d}\Gamma \tag{5.27}$$

To enforce inter-element continuity on the common element boundary, the unknown vector $\mathbf{c}_e$ should be expressed in terms of nodal DOF $\mathbf{d}_e$. Minimisation of the functional Ψ_{me} with respect to $\mathbf{c}$ and $\mathbf{d}$ yields

$$\frac{\partial \Psi_{me}}{\partial \mathbf{c}_e^{\mathrm{T}}} = \mathbf{H}_e\mathbf{c}_e - \mathbf{G}_e\mathbf{d}_e = \mathbf{0} \tag{5.28}$$

$$\frac{\partial \Psi_{me}}{\partial \mathbf{d}_e^{\mathrm{T}}} = -\mathbf{G}_e^{\mathrm{T}}\mathbf{c}_e + \mathbf{g}_e = \mathbf{0} \tag{5.29}$$

Eq. (5.28) provides the optional relationship between $\mathbf{c}_e$ and $\mathbf{d}_e$, that is

$$\mathbf{c}_e = \mathbf{H}_e^{-1}\mathbf{G}_e\mathbf{d}_e \tag{5.30}$$

Finally, substitution of Eq. (5.30) into Eq. (5.29) yields the element stiffness equation

$$\mathbf{K}_e\mathbf{d}_e = \mathbf{p}_e \tag{5.31}$$

where

$$\mathbf{K}_e = \mathbf{G}_e^{\mathrm{T}}\mathbf{H}_e^{-1}\mathbf{G}_e \tag{5.32}$$

$$\mathbf{p}_e = \mathbf{g}_e = \int_{\Gamma_{eq}} \tilde{\mathbf{N}}_e^{\mathrm{T}}\bar{q}t_e \mathrm{d}\Gamma \tag{5.33}$$

stand for the element stiffness matrix and the equivalent nodal load vector, respectively.

5.3.4 Recovery of rigid-body motion

By checking the functional (5.23) and Eq. (5.10), we know that the solution fails if any of the functions N_{ej} is a rigid-body motion mode associated with a vanishing boundary flux term of Q_{ej}. As a consequence, the matrix $\mathbf{H}_e$ is not of full rank and becomes singular for inversion. Therefore, special care should be taken to discard all rigid-body motion terms from u.

However, it is necessary to reintroduce the discarded modes in the internal field u_e of a particular element and then to calculate their undetermined coefficients by

requiring, for example, the least squares adjustment of u_e and $\tilde{u}_e$ [13]. In this case, these missing terms can easily be recovered by setting for the augmented internal field

$$u_e(\mathbf{x}) = \mathbf{N}_e(\mathbf{x})\,\mathbf{c}_e + c_0 \tag{5.34}$$

where the undetermined rigid-body motion parameter c_0 can be calculated using the least square matching of u_e and $\tilde{u}_e$ at element nodes

$$\sum_{i=1}^{n} (u_i - \tilde{u}_i)^2 = \min \tag{5.35}$$

which finally gives

$$c_0 = \frac{1}{n}\sum_{i=1}^{n} \Delta u_{ei} \tag{5.36}$$

in which n is the number of element nodes and

$$\Delta u_{ei} = \tilde{u}_{ei} - \hat{u}_{ei} \tag{5.37}$$

with $\hat{u}_e = \mathbf{N}_e\mathbf{c}_e$.

5.4 T-complete functions

T-complete functions are very important for constructing approximate fields in deriving Trefftz element formulation. For this reason it is necessary to know how to construct them and what is the suitable criterion for completeness. However, in the present book, our purpose focuses on the usage of T-complete functions, so the proof of completeness is not included (more detail can be found in Refs. [14, 15]).

It is well known that the solutions of the Laplace equation in polar coordinates

$$\nabla^2 u = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} = 0 \tag{5.38}$$

may be found by using the method of variable separation. By way of this method, the complete solutions are obtained as [5]

$$u(r,\theta) = \sum_{k=0}^{\infty} r^k (a_k \cos k\theta + b_k \sin k\theta) \tag{5.39}$$

in a two-dimensional bounded domain and

$$u(r,\theta) = d_0 + a_0 \ln r + \sum_{k=0}^{\infty} r^{-k} (a_k \cos k\theta + b_k \sin k\theta) \tag{5.40}$$

in a two-dimensional unbounded domain.

Thus, the associated T-complete sets can be expressed in the form

$$\bar{\mathrm{T}} = \left\{1,\ r^k \cos k\theta,\ r^k \sin k\theta \right\} = \{\bar{T}_i\} \tag{5.41}$$

$$\bar{\mathrm{T}} = \left\{1,\ lnr,\ r^{-k} \cos k\theta,\ r^{-k} \sin k\theta \right\} = \{\bar{T}_i\} \tag{5.42}$$

We note that $\bar{T}_1 = 1$ represents rigid-body motion and yields zero element stiffness, so in the expression of the approximated intra-element field, $\bar{T}_1 = 1$ should be abandoned. As a result, the final T-complete functions can be chosen as

$$N_1 = r\cos\theta,\ N_2 = r\sin\theta,\ \cdots,\ N_{2k-1} = r^k \cos k\theta,\ N_{2k} = r^k \sin k\theta,\ \cdots \tag{5.43}$$

Furthermore, the derivatives of T-complete functions with respect to spatial variables x_1 and x_2 can be deduced as*

$$T_{2k-1} = \left\{ \begin{array}{c} \dfrac{\partial N_{2k-1}}{\partial x_1} \\ \dfrac{\partial N_{2k-1}}{\partial x_2} \end{array} \right\} = \left\{ \begin{array}{c} kr^{k-1} \cos(k-1)\,\theta \\ -kr^{k-1} \sin(k-1)\,\theta \end{array} \right\} \tag{5.44}$$

$$T_{2k} = \left\{ \begin{array}{c} \dfrac{\partial N_{2k}}{\partial x_1} \\ \dfrac{\partial N_{2k}}{\partial x_2} \end{array} \right\} = \left\{ \begin{array}{c} kr^{k-1} \sin(k-1)\,\theta \\ kr^{k-1} \cos(k-1)\,\theta \end{array} \right\} \tag{5.45}$$

for the case of $k = 1, 2, \cdots$.

Furthermore, for a successful solution it is important to choose the proper number of trial functions for the element. The basic rule used to prevent spurious energy modes is analogous to that used in the hybrid stress element. The necessary (but not sufficient) condition for the matrix $\mathbf{H}_e$ to be of full rank is stated as [13]

$$m_{\min} = N_{dof} - 1 \tag{5.46}$$

where N_{dof} is the number of nodal degrees of freedom of the element. For example, in the case of a 4-node element $N_{dof} = 4$ and in the case of an 8-node element $N_{dof} = 8$. Although use of minimum number $m_{\min}$ does not always guarantee a stiffness matrix with full rank, full rank may always be achieved by suitably augmenting m.

Additionally, to keep the formulation symmetrical in terms of all coordinate directions, the terms $r^k \cos k\theta$ and $r^k \sin k\theta$ should be chosen simultaneously for every k. Since $\bar{T}_1 = 1$ is excluded in generating the sequence N_j, this symmetry rule implies that m should always be an even number in the case of two-dimensional bounded domains and an odd number in the case of two-dimensional unbounded domains.

---

* $r = \sqrt{x_1^2 + x_2^2},\ \frac{\partial r}{\partial x_1} = \frac{x_1}{r},\ \frac{\partial r}{\partial x_2} = \frac{x_2}{r}$
$\theta = \arctan\left(\frac{x_2}{x_1}\right),\ \frac{\partial \theta}{\partial x_1} = -\frac{x_2}{r^2},\ \frac{\partial \theta}{\partial x_2} = \frac{x_1}{r^2}$

5.5 Computation of H and G matrix

The computation of $\mathbf{H}_e$ and $\mathbf{G}_e$ matrices defined in Eqs. (5.25) and (5.26) is essential in HT-FEM and is performed along the element boundary. In the following analysis, we assume that there are n_e edges and n nodes at element e and on each edge there are n_o nodes.

5.5.1 Geometric characteristics of boundary edges

Using the shape functions defined on each edge (see Chapter 4), the coordinates of any point on the edge under consideration can be expressed as

$$\mathbf{x} = \sum_{i=1}^{n_o} N_i(\xi)\,\mathbf{x}_i \tag{5.47}$$

Further, the derivatives of space variables with respect to the coordinate ξ are written as

$$\begin{Bmatrix} \dfrac{\mathrm{d}x_1}{\mathrm{d}\xi} \\ \dfrac{\mathrm{d}x_2}{\mathrm{d}\xi} \end{Bmatrix} = \sum_{i=1}^{n_o} \frac{\mathrm{d}N_i(\xi)}{\mathrm{d}\xi} \begin{Bmatrix} x_{1i} \\ x_{2i} \end{Bmatrix} \tag{5.48}$$

while the outward normal vector at any point on the edge can be determined by

$$n_1 = \frac{\mathrm{d}x_2}{\mathrm{d}\Gamma} = \frac{\mathrm{d}x_2}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}\Gamma} = \left[\sum_{i=1}^{n_o} \frac{\mathrm{d}N_i(\xi)}{\mathrm{d}\xi} x_{2i}\right]\frac{1}{J} \tag{5.49}$$

$$n_2 = -\frac{\mathrm{d}x_1}{\mathrm{d}\Gamma} = -\frac{\mathrm{d}x_1}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}\Gamma} = -\left[\sum_{i=1}^{n_o} \frac{\mathrm{d}N_i(\xi)}{\mathrm{d}\xi} x_{1i}\right]\frac{1}{J} \tag{5.50}$$

where

$$\mathrm{d}\Gamma = \sqrt{(\mathrm{d}x_1)^2 + (\mathrm{d}x_2)^2} = \sqrt{\left(\frac{\mathrm{d}x_1}{\mathrm{d}\xi}\right)^2 + \left(\frac{\mathrm{d}x_2}{\mathrm{d}\xi}\right)^2}\,\mathrm{d}\xi = J\mathrm{d}\xi \tag{5.51}$$

and

$$J = \sqrt{\left(\frac{\mathrm{d}x_1}{\mathrm{d}\xi}\right)^2 + \left(\frac{\mathrm{d}x_2}{\mathrm{d}\xi}\right)^2} \tag{5.52}$$

5.5.2 Computation of H matrix

According to previous derivations, we rewrite Eq. (5.25) as

$$\mathbf{H}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathrm{T}} \mathbf{N}_e t_e \mathrm{d}\Gamma = \int_{\Gamma_e} \mathbf{T}_e^{\mathrm{T}} \mathbf{A}^{\mathrm{T}} \mathbf{N}_e t_e \mathrm{d}\Gamma \tag{5.53}$$

in which $\mathbf{A}$ is defined in Eq. (5.6), which represents the outward normal vector to the edge and is generally related to the coordinates of points on the edge.

For simplicity in the following writing, denote

$$\mathbf{F}(\mathbf{x}) = [F_{ij}(\mathbf{x})]_{m\times m} = \mathbf{T}_e^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{N}_e t_e \tag{5.54}$$

The matrix $\mathbf{H}_e$ can then be written as

$$H_{ij} = \int_{\Gamma_e} F_{ij}(\mathbf{x})\,\mathrm{d}\Gamma = \sum_{l=1}^{n_e}\int_{\Gamma_{el}} F_{ij}(\mathbf{x})\,\mathrm{d}\Gamma \tag{5.55}$$

Using Eqs. (5.47), (5.51) and Gaussian numerical integration as described in Chapter 4, Eq. (5.55) can be further written as

$$H_{ij} = \sum_{l=1}^{n_e}\left[\int_{-1}^{+1} F_{ij}(\mathbf{x}(\xi))J(\xi)\,\mathrm{d}\xi\right] \approx \sum_{l=1}^{n_e}\left[\sum_{k=1}^{n_s} w_k F_{ij}(\mathbf{x}(\xi_k))J(\xi_k)\right] \tag{5.56}$$

where n_s is the Gaussian sampling points employed in the Gaussian numerical integration.

5.5.3 Computation of G matrix

Substituting Eq. (5.10) into Eq. (5.26), we have

$$\mathbf{G}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathbf{T}}\tilde{\mathbf{N}}_e t_e \mathrm{d}\Gamma = \int_{\Gamma_e} \mathbf{T}_e^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\tilde{\mathbf{N}}_e t_e \mathrm{d}\Gamma \tag{5.57}$$

where $\mathbf{T}_e$, $\mathbf{A}$ and $\tilde{\mathbf{N}}_e$ are defined in Eqs. (5.11), (5.6) and (5.18). They are generally dependent on the spatial coordinates.

As was done for matrix $\mathbf{H}$, denote

$$\tilde{\mathbf{F}}(\mathbf{x}) = \left[\tilde{F}_{ij}(\mathbf{x})\right]_{m\times n} = \mathbf{T}_e^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\tilde{\mathbf{N}}_e t_e \tag{5.58}$$

As a result, the matrix $\mathbf{G}_e$ can be written as

$$G_{ij} = \int_{\Gamma_e} \tilde{F}_{ij}(\mathbf{x})\,\mathrm{d}\Gamma = \sum_{l=1}^{n_e}\int_{\Gamma_{el}} \tilde{F}_{ij}(\mathbf{x})\,\mathrm{d}\Gamma \tag{5.59}$$

Again, using Eqs. (5.47), (5.51) and Gaussian numerical integration as described in Chapter 4, we finally have

$$G_{ij} = \sum_{l=1}^{n_e}\left[\int_{-1}^{1} \tilde{F}_{ij}(\mathbf{x}(\xi))J(\xi)\,\mathrm{d}\xi\right] \approx \sum_{l=1}^{n_e}\left[\sum_{k=1}^{n_s} w_k \tilde{F}_{ij}(\mathbf{x}(\xi_k))J(\xi_k)\right] \tag{5.60}$$

5.6 Computation of equivalent nodal load

Since, in potential problems, a generalised load at a point is a scalar, evaluation of the equivalent nodal load is simple in contrast to that encountered in the next chapter. If on a particular edge of an element (see Figure 5.4) the normal flux q is prescribed, the equivalent nodal load can be calculated through the following equation (see Eq. (5.33))

$$\mathbf{p}_e = \int_{\Gamma_{eq}} \tilde{\mathbf{N}}_e^{\mathrm{T}} \bar{q} \mathrm{d}\Gamma \tag{5.61}$$

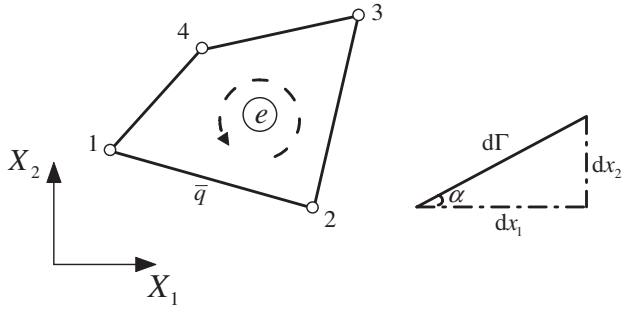


FIGURE 5.4
Computation of equivalent nodal load for potential problems

Although the vector $\tilde{\mathbf{N}}_e$ is defined on the entire element boundary, the nonzero equivalent nodal load exists on the loaded edge only. So we can rewrite Eq. (5.61) in component form as

$$p_{ie} = \int_{\Gamma_{eq}} \hat{N}_i(\xi) \bar{q} \mathrm{d}\Gamma \tag{5.62}$$

in which $\hat{N}_i(\xi)$ is the shape function at the ith node on the loaded edge, and p_{ie} is the corresponding component of equivalent nodal flux.

To make the following derivation more general, we assume that there are n nodes on the loaded edge. Then, the distribution of normal flux along the loaded edge is expressed as

$$\bar{q}(\xi) = \sum_{i=1}^{n} \hat{N}_i(\xi) \bar{q}_i \tag{5.63}$$

Similarly, the coordinates and their derivatives to the non-dimensional parameter ξ at any point along the loaded edge are given as

$$\begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \sum_{i=1}^{n} N_i(\xi) \begin{Bmatrix} x_{1i} \\ x_{2i} \end{Bmatrix} \tag{5.64}$$

$$\begin{Bmatrix} \frac{dx_1}{d\xi} \\ \frac{dx_2}{d\xi} \end{Bmatrix} = \sum_{i=1}^{n} \frac{dN_i(\xi)}{d\xi} \begin{Bmatrix} x_{1i} \\ x_{2i} \end{Bmatrix} \tag{5.65}$$

As a result, the arc length can be obtained

$$d\Gamma = \sqrt{(dx_1)^2 + (dx_2)^2} = \sqrt{\left(\frac{dx_1}{d\xi}\right)^2 + \left(\frac{dx_2}{d\xi}\right)^2} d\xi = J d\xi \tag{5.66}$$

Making use of Eqs. (5.63) and (5.66), we can convert the arc integral (5.62) into a standard one-dimensional integral equation, which can be evaluated using Gaussian quadrature:

$$\begin{aligned} p_{ie} &= \int_{\Gamma_e} N_i(\xi)\,\bar{q} d\Gamma = \int_{-1}^{+1} N_i(\xi)\,\bar{q}(\xi)\,J(\xi)\,d\xi \\ &\approx \sum_{k=1}^{n_s} w_k N_i(\xi_k)\,\bar{q}(\xi_k)\,J(\xi_k) \end{aligned} \tag{5.67}$$

where n_s is the number of Gaussian sampling points.

5.7 Program structure for HT-FEM

In the previous sections, the theory required by the computer programming in the following sections of this chapter is presented. To make the program developed more applicable and understandable, a modular approach is adopted for the programs presented in this book. Thus the various main HT-FE operations are performed by separate subroutines. Figure 5.5 shows the structure of all modules employed, particularly the sequence in which particular submodules are accessed and called. The basic HT-FE analysis is performed by primary modules which rely on auxiliary modules to carry out secondary operations. An auxiliary module may be required by more than one primary module, as shown in Figure 5.5, where it can be seen that the order of calling of the primary modules is controlled by a main function.

5.8 MATLAB programming for potential problems

```
function MAINFUN
% Main program using HTFEM for 2D Laplace problems
```

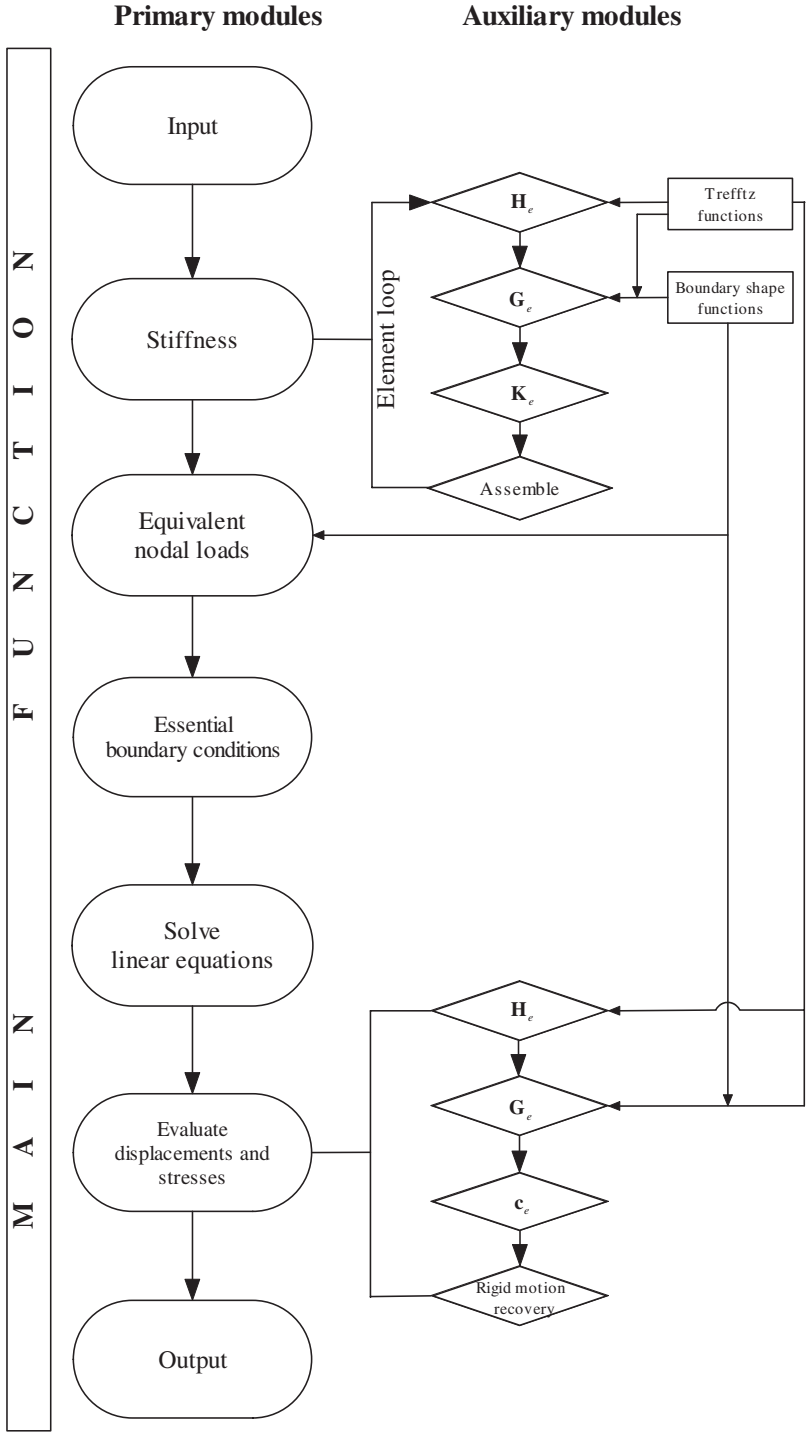


FIGURE 5.5
Program structure for HT-FEM

```
% ******************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;

disp('*******************************************');
disp('              Hybrid Trefftz FEM');
disp('           for 2D Laplace problems');
disp('*******************************************');

% Input data from file
[NPOIN,NELEM,COORD,MATNO,LNODS,NVFIX,NOFIX,IFPRE,...
    PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,NOPRS,PRESS,...
    PROPS]=INPUTDT;
% Generate local relations of nodes and edges
[ELNOD]=TYPELEM;
% Form stiffness matrix
NEQNS=NPOIN*NDOFN;
GSTIF=zeros(NEQNS,NEQNS);
GLOAD=zeros(NEQNS,1);
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute element stiffness matrix
    [ESTIF]=KMATRIX(EHMTX,EGMTX);
    % Assemble stiffness matrix
    [GSTIF]=ASMSTIF(iELEM,LNODS,ESTIF,GSTIF);
end
% Compute equivalent loads
[GLOAD]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,...
    NEASS,NOPRS,PRESS,NPLOD,LODPT,POINT,GLOAD);
% Introduce constrained displacements
[GSTIF,GLOAD]=INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,...
    GSTIF,GLOAD);
% Solve linear system of equations and store
% displacements of each node in the array ASDIS
[ASDIS]=LSSOLVR(GSTIF,GLOAD,NEQNS);
% Output nodal displacements
[UPOIN]=FIEDNOD(NPOIN,ASDIS);
% Compute displacement and stress components at central
% point of element
```

```
[CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,LNODS,COORD,...
    PROPS,ELNOD,ASDIS);
% Output results
OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,...
    NVFIX,NPLOD,NDLEG);
disp('--- All procedure are finished ---');


------------------------------------------------------
function [ELNOD]=TYPELEM
% Form the local relation of edge and nodes according
% to specified parameters in element: NNODE, NEDGE,
% NODEG
% Output parameters:
%   ELNOD: Local relation of edge and nodes
% ***************************************
global NNODE NEDGE NODEG;

if (NEDGE==4)&(NODEG==2)% 4-node element
    [ELNOD]=TELE442;
elseif (NEDGE==4)&(NODEG==3) % 8-node element
    [ELNOD]=TELE843;
end

if NNODE~=(NEDGE*(NODEG-1))
    error('Wrong element type, Check it!');
end

------------------------------------------------------
function [ELNOD]=TELE442
% Element properties of 4-node quadrature element
%               4             3
%               o******<******o
%               *             *
%               *             *
%               v             ^
%               *             *
%               *             *
%               *             *
%               o******>******o
%               1             2
global NEDGE NODEG;

% Local relation of edges and nodes
ELNOD=zeros(NEDGE,NODEG);
```

```
ELNOD(1,1)=1; % edge 1
ELNOD(1,2)=2;

ELNOD(2,1)=2; % edge 2
ELNOD(2,2)=3;

ELNOD(3,1)=3; % edge 3
ELNOD(3,2)=4;

ELNOD(4,1)=4; % edge 4
ELNOD(4,2)=1;

-----------------------------------------------------
function [ELNOD]=TELE843
% Element properties of 8-node quadrature element
%               7      6      5
%               o******o******o
%               *             *
%               *             *
%             8 o             o 4
%               *             *
%               *             *
%               *             *
%               o******o******o
%               1      2      3
global NEDGE NODEG;

% Local relation of edges and nodes
ELNOD=zeros(NEDGE,NODEG);
ELNOD(1,1)=1; % edge 1
ELNOD(1,2)=2; ELNOD(1,3)=3;

ELNOD(2,1)=3; % edge 2
ELNOD(2,2)=4; ELNOD(2,3)=5;

ELNOD(3,1)=5; % edge 3
ELNOD(3,2)=6; ELNOD(3,3)=7;

ELNOD(4,1)=7; % edge 4
ELNOD(4,2)=8; ELNOD(4,3)=1;


-----------------------------------------------------
function [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD)
% Generate coordinates of nodes and central point in
```

```
% specified element
% Input parameters:
%   iELEM: Specified element
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
% Output parameters:
%   ECOOD: Coordinates of nodes of specified element
%   CenCoord: Coordinates of centroid of specified
%             element
% *********************************
global NDIME NDOFN NNODE NEDGE NODEG;

% Compute nodal coordinates of the given element
ECOOD=zeros(NNODE,NDIME);
for iNODE=1:NNODE
    kPOIN=LNODS(iELEM,iNODE);
    ECOOD(iNODE,:)=COORD(kPOIN,:);
end
% Coordinates of central point
CenCoord=sum(ECOOD)/NNODE;
% Form local element coordinates system
ECOOD(:,1)=ECOOD(:,1)-CenCoord(1);
ECOOD(:,2)=ECOOD(:,2)-CenCoord(2);


-------------------------------------------------------
function [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS)
% Compute element matrix H
%   H = integral( QT * N) on element boundary
% Input parameters:
%   ECOOD: Coordinates of nodes of specified element
%   ELNOD: Local relation of edge and nodes
%   kMATS: Material number of specified element
%   PROPS: Properties of materials
% Output parameters:
%   EHMTX: Element H matrix
% *********************************
global NTREF NDIME NDOFN NNODE NEDGE NGAUS;

% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);
% Material properties
THICK=PROPS(kMATS,3);
% Compute H matrix of every edge of element
EHMTX=zeros(NTREF,NTREF);
```

```
for iEDGE=1:NEDGE
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
            EXISP,ECOOD,ELNOD);
        DVOLU=DVOLU*WEIGP(iGAUS);
        if THICK+1~=1
            DVOLU=DVOLU*THICK;
        end
        [N_SET,T_SET]=TREFFTZ(CORGS(1),CORGS(2));
        Q_SET=AMTRX*T_SET;
        QTN=Q_SET'*N_SET; % m by m
        for im=1:NTREF
            for jm=1:NTREF
                EHMTX(im,jm)=EHMTX(im,jm)+...
                    QTN(im,jm)*DVOLU;
            end
        end
    end
end
clear Q_SET QTN N_SET T_SET POSGP WEIGP SHMTX AMTRX;


------------------------------------------------------
function [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS)
% Compute element matrix G
%    G = integral( QT* ShapEge) along element boundary
% Input parameters:
%   ECOOD: Coordinates of nodes of specified element
%   ELNOD: Local relation of edge and nodes
%   kMATS: Material number of specified element
%   PROPS: Properties of materials
% Output parameters:
%   EGMTX: Element G matrix
% *********************************
global NTREF NDIME NDOFN NEDGE NNODE NGAUS;

% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);
% Material properties
THICK=PROPS(kMATS,3);
NEVAB=NNODE*NDOFN;
EGMTX=zeros(NTREF,NEVAB);
for iEDGE=1:NEDGE
    for iGAUS=1:NGAUS
```

```
        EXISP=POSGP(iGAUS);
        [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
            EXISP,ECOOD,ELNOD);
        DVOLU=DVOLU*WEIGP(iGAUS);
        if THICK+1~=1
            DVOLU=DVOLU*THICK;
        end
        [N_SET,T_SET]=TREFFTZ(CORGS(1),CORGS(2));
        Q_SET=AMTRX*T_SET;
        QTS=Q_SET'*SHMTX;
        for im=1:NTREF
            for jn=1:NEVAB
                EGMTX(im,jn)=EGMTX(im,jn)+...
                    QTS(im,jn)*DVOLU;
            end
        end
    end
end
clear Q_SET QTS N_SET T_SET POSGP WEIGP SHMTX AMTRX;


------------------------------------------------------
function [KE]=KMATRIX(HE, GE)
% Form element stiffness matrix
%     K = GT * inv(H) * G
% Input parameters:
%   HE: H matrix
%   GE: G matrix
% Output parameters:
%   KE: element stiffness matrix
% ******************************
KE=GE'*inv(HE)*GE;


------------------------------------------------------
function [GP]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,...
    NEASS,NOPRS,PRESS,NPLOD,LODPT,POINT,GP)
% Compute the effective nodal loads
% Input parameters:
%   MATNO: Material index of each element
%   PROPS: Properties of materials
%   LNODS: Element connections
%   COORD: Coordinates of nodes
%   NPLOD: Number of concentrated loads
%   LODPT: Global index of nodes at which concentrated
```

```
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NDLEG: Number of loaded edges
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   GP:    Global effective nodal loads
% Output parameters:
%   GP:    Global effective nodal loads
% ************************************
global NDIME NDOFN NNODE NEDGE NODEG NGAUS;

% Add the point loads to global load vector
if NPLOD>0
    for iPLOD=1:NPLOD
        NRT=(LODPT(iPLOD)-1)*NDOFN;
        for iDOFN=1: NDOFN
            NR=NRT+iDOFN;
            GP(NR,1)=GP(NR,1)+POINT(iPLOD,iDOFN);
        end
    end
end

% Total number of local DOF of each element
NEVAB=NNODE*NDOFN;
% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);

% Calculate equivalent nodal loads on the distributed
% loaded edge
for iDLEG=1:NDLEG
    kELEM=NEASS(iDLEG);
    % Material properties
    kMATS=MATNO(kELEM);
    THICK=PROPS(kMATS,3);
    % Determine coordinates of nodes on element edge
    ELCOD=zeros(NODEG,NDIME);
    for iODEG=1:NODEG
        kPOIN=NOPRS(iDLEG,iODEG);
        for iDIME=1:NDIME
            ELCOD(iODEG,iDIME)=COORD(kPOIN,iDIME);
        end
        %ELCOD(iODEG,:)=COORD(kPOIN,:)
    end
```

```
% Determine local nodal load intensity
EPRES=zeros(NODEG,NDOFN);
for iODEG=1:NODEG
    for iDOFN=1:NDOFN
        ii=(iODEG-1)*NDOFN+iDOFN;
        EPRES(iODEG,iDOFN)=PRESS(iDLEG,ii);
    end
end
% Integration along the loaded edge
%    P(iODEG)(iDOFN)=integral(N(iODEG)*p(iDOFN)*dS)
% dS is arc-length
PE=zeros(NEVAB,1);
for iGAUS=1:NGAUS
    EXISP=POSGP(iGAUS);
    % Shape function and its derivatives for 1D
    % line element
    SHAPE=zeros(1,NODEG);
    DSHAP=zeros(1,NODEG);
    [SHAPE,DSHAP]=SHAPFUN(EXISP);

    % Coordinates and derivatives of Gauss points
    % x=sum(Ni*xi) and y=sum(Ni*yi)
    % dx/dt=sum(dNi/dt*xi) and dy/dt=sum(dNi/dt*yi)
    CORGS=zeros(1,NDIME);
    DERGS=zeros(1,NDIME);
    for iDIME=1:NDIME
        for iODEG=1:NODEG
            CORGS(iDIME)=CORGS(iDIME)+...
                SHAPE(iODEG)*ELCOD(iODEG,iDIME);
            DERGS(iDIME)=DERGS(iDIME)+...
                DSHAP(iODEG)*ELCOD(iODEG,iDIME);
        end
    end
    % DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    DVOLU=sqrt(DERGS(1)^2+DERGS(2)^2);

    % Gauss integration factor
    DVOLU=DVOLU*WEIGP(iGAUS);
    if THICK+1~=1
        DVOLU=DVOLU*THICK;
    end

    % Load intensity at Gauss point
    %          p=sum(Ni*pi)
    % for potential problems, PGASH is scalar
```

```
        % denoting normal flux
        PGASH=zeros(NDOFN,1);
        for iDOFN=1:NDOFN
            for iODEG=1:NODEG
                PGASH(iDOFN)=PGASH(iDOFN)+...
                    SHAPE(iODEG)*EPRES(iODEG,iDOFN);
            end
        end
        % Compute equivalent nodal load vector PE
        for iNODE=1:NNODE
            kPOIN=LNODS(kELEM,iNODE);
            if kPOIN==NOPRS(iDLEG,1)
                % iNODE is start point of loaded edge
                for iODEG=1:NODEG
                    kNODE=iNODE+iODEG-1;
                    if kNODE>NNODE
                        kNODE=1;
                    end
                    for iDOFN=1:NDOFN
                        iEVAB=(kNODE-1)*NDOFN+iDOFN;
                        PE(iEVAB,1)=PE(iEVAB,1)+...
                            SHAPE(iODEG)*...
                            PGASH(iDOFN)*DVOLU;
                    end
                end
            end
        end
    end
    % Assemble PE into global load term
    for iNODE=1:NNODE
        kPOIN=LNODS(kELEM,iNODE);
        for iDOFN=1:NDOFN
            kEQNS=NDOFN*(kPOIN-1)+iDOFN; % global DOF
            iEVAB=NDOFN*(iNODE-1)+iDOFN; % local DOF
            GP(kEQNS,1)=GP(kEQNS,1)+PE(iEVAB,1);
        end
    end
end
clear ELCOD EPRES PE SHAPE DSHAP PGASH POSGP WEIGP;


------------------------------------------------------
function [UPOIN]=FIEDNOD(NPOIN,ASDIS)
% Generate nodal generalised displacement field
% Input parameters:
```

```
%   NPOIN: Number of nodes in domain
%   ASDIS: Nodal generalised displacement field in DOF
%          order
% Output parameters:
%   UNODE: Nodal generalised displacement field
% **********************************
global NDOFN NDIME;

UPOIN=zeros(NPOIN,NDOFN);
for iPOIN=1:NPOIN
    NRT=(iPOIN-1)*NDOFN;
    for iDOFN=1:NDOFN
        NR=NRT+iDOFN;
        UPOIN(iPOIN,iDOFN)=ASDIS(NR);
    end
end


--------------------------------------------------------
function [CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    LNODS,COORD,PROPS,ELNOD,ASDIS)
% Compute potential and flux at central point of each
% element
% Input parameters:
%   NELEM: Number of elements in domain
%   MATNO: Material index of each element
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
%   PROPS: Properties of materials
%   ELNOD: Local relation of edge and nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
% Output parameters:
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
% **********************************
global NDIME NDOFN NNODE NEDGE NODEG NSTRE NGAUS NMATS;

CECOD=zeros(NELEM,NDIME);
UCENP=zeros(NELEM,NDOFN);
SCENP=zeros(NELEM,NSTRE);
% Loop for all nodes
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
```

```
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Identify nodal field of the specified element
    [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Calculate the ce coefficients: m by 1
    [c_Ele]=CMATRIX(EHMTX,EGMTX,d_Ele);
    % Recover rigid displacement
    [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele);
    % Compute Trefftz internal fields at central point
    xp=0;
    yp=0;
    [N_SET,T_SET]=TREFFTZ(xp,yp);
    GDISP=N_SET*c_Ele;
    GSTRE=T_SET*c_Ele;
    UCENP(iELEM,1)=GDISP(1)+c0;
    SCENP(iELEM,1)=GSTRE(1);
    SCENP(iELEM,2)=GSTRE(2);
    % Coordinates of commputing point
    CECOD(iELEM,:)=[CenCoord(1)+xp,CenCoord(2)+yp];
end
clear EHMTX EGMTX c_Ele d_Ele N_SET T_SET GDISP GSTRE;


------------------------------------------------------
function [CORGS,DVOLU,AMTRX,SHMTX]=...
    QUANGAS(iEDGE,EXISP,ECOOD,ELNOD)
% Evaluate quantities at Gaussian points
% Input parameters:
%   iEDGE: Number index of element edge
%   EXISP: Local coordinate of Gaussian points
%   ECOOD: Coordinates of element nodes
%   ELNOD: Local relations of edge and nodes
% Output parameters:
%   CORGS: Coordinates of Gauss point
%   DVOLU: Functions of coordinate tranformation for
%          Gauss integral
%   AMTRX: Directional cosine at Gauss point
%   SHMTX: Shape function of frame field defined on
%          entire boundary
% ***********************************
```

```
global NDIME NDOFN NNODE NEDGE NODEG;

% Shape function and its derivatives for 1D line
% element
SHAPE=zeros(1,NODEG);
DSHAP=zeros(1,NODEG);
[SHAPE,DSHAP]=SHAPFUN(EXISP);

% Coordinates and derivatives of Gauss points
%  x=sum(Ni*xi) and y=sum(Ni*yi)
%  dx/dt=sum(dNi/dt*xi) and dy/dt=sum(dNi/dt*yi)
CORGS=zeros(1,NDIME);
DERGS=zeros(1,NDIME);
for iDIME=1:NDIME
    for iODEG=1:NODEG
        kNODE=ELNOD(iEDGE,iODEG);
        CORGS(iDIME)=CORGS(iDIME)+...
            SHAPE(iODEG)*ECOOD(kNODE,iDIME);
        DERGS(iDIME)=DERGS(iDIME)+...
            DSHAP(iODEG)*ECOOD(kNODE,iDIME);
    end
end
% Coordinate tranformation for Gauss integral
%  DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
DVOLU=sqrt(DERGS(1)^2+DERGS(2)^2);

% Directional cosine at Gauss point
%  nx = dy/dS   ny = - dx/dS
AMTRX=zeros(1,NDIME);
AMTRX(1)= DERGS(2)/DVOLU;
AMTRX(2)=-DERGS(1)/DVOLU;

% Form shape function matrix of frame function
SHMTX=zeros(1,NNODE);
iNODE=ELNOD(iEDGE,1); % start node of the edge
for iODEG=1:NODEG
    kNODE=iNODE+iODEG-1;
    if kNODE>NNODE
        kNODE=1;
    end
    SHMTX(kNODE)=SHAPE(iODEG);
end
```

```
function [N_SET,T_SET]=TREFFTZ(sp,tp)
% Compute the Trefftz functions with specified terms
% Input parameters:
%   sp,tp: coordinates
% Output parameters:
%   N_SET: Trefftz functions
%   T_SET: Derivatives of Trefftz functions
%             dN/ds and dN/dt
% ***********************************************
global NTREF NNODE NDOFN;

% Check the number of terms of Trefftz functions
temp=NNODE*NDOFN-1;
if NTREF<temp
    error('Too small terms of Trefftz functions!');
end

% Compute N_SET and T_SET
N_SET=zeros(1,NTREF);
T_SET=zeros(2,NTREF);

r=sqrt(sp^2+tp^2);
s=atan2(tp,sp);

for im=1:NTREF
    % Determine the order of Ni
    n=ceil(im/2);
    % Justify im is odd or even number
    remaind=rem(im,2);
    if 1+remaind==1 % im is even
        N_SET(1,im)=r^(n)*sin(n*s);
        T_SET(1,im)=n*r^(n-1)*sin((n-1)*s);
        T_SET(2,im)=n*r^(n-1)*cos((n-1)*s);
    else % im is odd
        N_SET(1,im)=r^(n)*cos(n*s);
        T_SET(1,im)=n*r^(n-1)*cos((n-1)*s);
        T_SET(2,im)=-n*r^(n-1)*sin((n-1)*s);
    end
end


------------------------------------------------------
function [c_Ele]=CMATRIX(H_Ele,G_Ele,d_Ele)
% Compute the unknown coefficients vector c in a
% particular element
```

```
% Input parameters:
%   H_Ele: Element H matrix
%   G_Ele: Element G matrix
%   d_Ele: Nodal generalised displacement field of
%   element
% Output parameters:
%   c_Ele: Coefficients of Trefftz interpolation
% *********************************
global NTREF;

% Compute the coefficients of Trefftz interpolation
% ce=inv(H)*G*d
c_Ele=zeros(NTREF,1);
c_Ele=inv(H_Ele)*G_Ele*d_Ele;


------------------------------------------------------
function [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele)
% Recovery of rigid body motion
% Input parameters:
%   ECOOD: Coordinates of element nodes
%   c_Ele: Coefficients of Trefftz interpolation
%   d_Ele: Displacement field at element nodes
% Output parameters:
%   c0: Rigid body motion term
% *********************************
global NNODE;

sum=0;
for iNODE=1:NNODE
    xp=ECOOD(iNODE,1);
    yp=ECOOD(iNODE,2);
    [N_SET,T_SET]=TREFFTZ(xp,yp);
    sum=sum+(d_Ele(iNODE)-N_SET*c_Ele);
end
c0=sum/NNODE;


-----------------------------------------------------
function [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS)
% Identify the nodal displacements of the specified
% element
% Input parameters:
%   iELEM: Specified element
%   LNODS: Element connectivity
```

```
%   ASDIS: Nodal generalized displacement field in DOF
%          order
% Output parameters:
%   d_Ele: Nodal generalised displacement field in
%          specified element
% ********************************
global NDIME NDOFN NNODE;

NEVAB=NNODE*NDOFN;
d_Ele=zeros(NEVAB,1);
for iNODE=1:NNODE
    kPOIN=LNODS(iELEM,iNODE);
    NR=(kPOIN-1)*NDOFN;
    nr=(iNODE-1)*NDOFN;
    for iDOFN=1:NDOFN
        NPOSN=NR+iDOFN;
        iEVAB=nr+iDOFN;
        d_Ele(iEVAB)=ASDIS(NPOSN);
    end
end
```

5.9 C computer programming

```
/*
*********************************************************
* Mainfunction MAINFUN                                  *
* - Call other subroutines for solving Laplace          *
*   problems using HTFEM                                *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int NTREF,NTYPE,NNODE,NEDGE,NODEG,NDIME,NDOFN,NSTRE,
NMATS,NPROP,NGAUS;
void main()
{
    void DUCLEAN(); void ITCLEAN(); void INPUTDT();
    void TYPELEM(); void CHECKDT(); void ELEPARS();
    void HMATRIX(); void GMATRIX(); void KMATRIX();
```

```
void ASMSTIF(); void PVECTOR(); void INDISBC();
void LSSOLVR(); void FIEDNOD(); void FIEDCEN();
void OPRESUT();
FILE *fp;
int NEQNS,NPOIN,NELEM,NVFIX,NPLOD,NDLEG,TNFEG,NEVAB;
int *MATNO,*LNODS,*NOFIX,*IFPRE,*LODPT,*NEASS,*NOPRS,
    *ELNOD;
double *COORD,*PRESC,*POINT,*PRESS,*PROPS;
double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*ESTIF,*GSTIF,
    *GLOAD,*UPOIN,*CECOD,*UCENP,*SCENP;
char dummy[201],TITLE[201],file[81];
int i,j,k,N,n1,n2,iELEM,kMATS;

printf("**********************************\n");
printf("        Hybrid Trefftz FEM\n");
printf("      for 2D Laplace problems\n");
printf("**********************************\n");
/** Input data from file **/
puts("Input file name < dir:fn.txt >: ");
gets(file);
if((fp=fopen(file,"r"))==NULL)
{
    printf("Warning! Can't open input file\n");
    exit(0);
}
// basic parameters
fgets(dummy,200,fp);
fgets(TITLE,200,fp);
fgets(dummy,200,fp);

fgets(dummy,200,fp);
fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
    &NPLOD,&NDLEG);
```

```
// element connectivity
MATNO=(int *)calloc(NELEM,sizeof(int));
ITCLEAN(NELEM,1,MATNO);
LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
ITCLEAN(NELEM,NNODE,LNODS);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NELEM;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    MATNO[i]=n1-1;
    for(j=0;j<NNODE;j++)
    {
        fscanf(fp,"%d",&n2);
        LNODS[i*NNODE+j]=n2-1;
    }
    fscanf(fp,"\n");
}
// nodal coordinates
COORD=(double *)calloc(NPOIN*NDIME,sizeof(double));
DUCLEAN(NPOIN,NDIME,COORD);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NPOIN;i++)
{
    fscanf(fp,"%d",&N);
    for(j=0;j<NDIME;j++)
    {
        fscanf(fp,"%lf",&COORD[i*NDIME+j]);
    }
    fscanf(fp,"\n");
}
// specified nodal potential/displacement
NOFIX=(int *)calloc(NVFIX,sizeof(int));
ITCLEAN(NVFIX,1,NOFIX);
IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
ITCLEAN(NVFIX,NDOFN,IFPRE);
PRESC=(double *)calloc(NVFIX*NDOFN,sizeof(double));
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
```

```
        NOFIX[i]=n1-1;
        for(j=0;j<NDOFN;j++)
        {
            fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
        }
        for(j=0;j<NDOFN;j++)
        {
            fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
        }
        fscanf(fp,"\n");
    }
    // specified concentrated loads at nodes
    fgets(dummy,200,fp);
    if(NPLOD>0)
    {
        LODPT=(int *)calloc(NPLOD*1,sizeof(int));
        ITCLEAN(NPLOD,1,LODPT);
        POINT=(double *)calloc(NPLOD*NDOFN,
            sizeof(double));
        DUCLEAN(NPLOD,NDOFN,POINT);
        fgets(dummy,200,fp);
        for(i=0;i<NPLOD;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            LODPT[i]=n1-1;
            for(j=0;j<NDOFN;j++)
            {
                fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
            }
            fscanf(fp,"\n");
        }
    }
    // specified distributed edge loads
    fgets(dummy,200,fp);
    if(NDLEG>0)
    {
        NEASS=(int *)calloc(NDLEG*1,sizeof(int));
        ITCLEAN(NDLEG,1,NEASS);
        NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
        ITCLEAN(NDLEG,NODEG,NOPRS);
        TNFEG=NODEG*NDOFN;
        PRESS=(double *)calloc(NDLEG*TNFEG,sizeof(double));
        DUCLEAN(NDLEG,TNFEG,PRESS);
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
```

```
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NPROP;j++)
        {
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }

    /** Check data **/
    CHECKDT(NPOIN,NELEM,COORD,MATNO,LNODS,NVFIX,NOFIX,
        IFPRE,PRESC,NPLOD,LODPT,POINT,NDLEG,NEASS,NOPRS,
        PRESS,PROPS);

    /** Establish local relations of nodes and edges **/
    ELNOD=(int *)calloc(NEDGE*NODEG,sizeof(int));
    ITCLEAN(NEDGE,NODEG,ELNOD);
    TYPELEM(ELNOD);

    /** Form stiffness matrix **/
    NEQNS=NPOIN*NDOFN;
    GSTIF=(double *)calloc(NEQNS*NEQNS,sizeof(double));
    DUCLEAN(NEQNS,NEQNS,GSTIF);
    for(iELEM=0;iELEM<NELEM;iELEM++)
```

```
{
    kMATS=MATNO[iELEM];
    // Compute some quantities related to each element
    ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
    DUCLEAN(NNODE,NDIME,ECOOD);
    CenCoord=(double *)calloc(1*NDIME,sizeof(double));
    DUCLEAN(1,NDIME,CenCoord);
    ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
    // Compute H matrix
    EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
    DUCLEAN(NTREF,NTREF,EHMTX);
    HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
    // Compute G matrix
    NEVAB=NNODE*NDOFN;
    EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
    DUCLEAN(NTREF,NEVAB,EGMTX);
    GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
    // Compute element stiffness matrix
    ESTIF=(double *)calloc(NEVAB*NEVAB,sizeof(double));
    DUCLEAN(NEVAB,NEVAB,ESTIF);
    KMATRIX(EHMTX,EGMTX,ESTIF);
    // Assemble stiffness matrix
    ASMSTIF(iELEM,NEQNS,LNODS,ESTIF,GSTIF);
    free(EHMTX); free(EGMTX); free(ESTIF);
}
// Compute equivalent loads
GLOAD=(double *)calloc(NEQNS*1,sizeof(double));
DUCLEAN(NEQNS,1,GLOAD);
PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,NOPRS,
    PRESS,NPLOD,LODPT,POINT,GLOAD);

// Introduce constrained displacements
INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,GSTIF,GLOAD);

// Solve linear system of equations
LSSOLVR(GSTIF,GLOAD,NEQNS);

// Output nodal displacement
UPOIN=(double *)calloc(NPOIN*NDOFN,sizeof(double));
DUCLEAN(NPOIN,NDOFN,UPOIN);
FIEDNOD(NPOIN,GLOAD,UPOIN);

// Compute quantities at centroid of each element
CECOD=(double *)calloc(NELEM*NDIME,sizeof(double));
DUCLEAN(NELEM,NDIME,CECOD);
```

```
    UCENP=(double *)calloc(NELEM*NDOFN,sizeof(double));
    DUCLEAN(NELEM,NDOFN,UCENP);
    SCENP=(double *)calloc(NELEM*NSTRE,sizeof(double));
    DUCLEAN(NELEM,NSTRE,SCENP);
    FIEDCEN(NELEM,MATNO,LNODS,COORD,PROPS,ELNOD,GLOAD,
        CECOD,UCENP,SCENP);

    // Output results
    OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,
        NVFIX,NPLOD,NDLEG);

    free(COORD); free(LNODS); free(MATNO);
    free(NOFIX); free(IFPRE); free(PRESC);
    free(PROPS); free(ECOOD); free(CenCoord);
    free(GSTIF); free(GLOAD); free(UPOIN);
    free(CECOD); free(UCENP); free(SCENP);
    printf("------------ Finished ------------\n");
    return;
}


/*
*******************************************************
* Subroutine TYPELEM                                  *
* - Local relations of edge and node number for       *
*   given element                                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void TYPELEM(int ELNOD[])
{
    void TELE442(); void TELE843();
    extern int NNODE,NEDGE,NODEG;

    if(NNODE!=NEDGE*(NODEG-1))
    {
        printf("Wrong element type, Check it!\n");
        exit(0);
    }
    if((NEDGE==4)&&(NODEG==2))
    {
        TELE442(ELNOD);
    }
```

```
    if((NEDGE==4)&&(NODEG==3))
    {
        TELE843(ELNOD);
    }
    return;
}

/*
      4-node quadrilateral element
               3             2
               o******<******o
               *             *
               v             ^
               *             *
               o******>******o
               0             1
*/
void TELE442(int ELNOD[])
{
    extern int NODEG;
    // Local relation of edges and nodes
    ELNOD[0*NODEG+0]=0; // edge 1
    ELNOD[0*NODEG+1]=1;

    ELNOD[1*NODEG+0]=1; // edge 2
    ELNOD[1*NODEG+1]=2;

    ELNOD[2*NODEG+0]=2; // edge 3
    ELNOD[2*NODEG+1]=3;

    ELNOD[3*NODEG+0]=3; // edge 4
    ELNOD[3*NODEG+1]=0;
    return;
}

/*
        8-node quadrilateral element
               6      5      4
               o******o******o
               *             *
             7 o             o 3
               *             *
               o******o******o
               0      1      2
*/
```

```
void TELE843(int ELNOD[])
{
    extern int NODEG;
    // Local relation of edges and nodes
    ELNOD[0*NODEG+0]=0; // edge 1
    ELNOD[0*NODEG+1]=1;
    ELNOD[0*NODEG+2]=2;

    ELNOD[1*NODEG+0]=2; // edge 2
    ELNOD[1*NODEG+1]=3;
    ELNOD[1*NODEG+2]=4;

    ELNOD[2*NODEG+0]=4; // edge 3
    ELNOD[2*NODEG+1]=5;
    ELNOD[2*NODEG+2]=6;

    ELNOD[3*NODEG+0]=6; // edge 4
    ELNOD[3*NODEG+1]=7;
    ELNOD[3*NODEG+2]=0;
    return;
}


/*
*******************************************************
* Subroutine ELEPARS                                  *
* - Compute geometric quantities related to element   *
*******************************************************
*/
void ELEPARS(int iELEM,int LNODS[],double COORD[],
             double ECOOD[],double CenCoord[])
{
    extern int NDIME,NDOFN,NNODE,NEDGE,NODEG;
    int ii,jj,kp,n1;

    for(ii=0;ii<NNODE;ii++)
    {
        kp=LNODS[iELEM*NNODE+ii];
        for(jj=0;jj<NDIME;jj++)
        {
            n1=ii*NDIME+jj;
            ECOOD[n1]=COORD[kp*NDIME+jj];
            CenCoord[jj]=CenCoord[jj]+ECOOD[n1];
        }
    }
```

```
    for(jj=0;jj<NDIME;jj++)
    {
        CenCoord[jj]=CenCoord[jj]/NNODE;
    }
    for(ii=0;ii<NNODE;ii++)
    {
        for(jj=0;jj<NDIME;jj++)
        {
            n1=ii*NDIME+jj;
            ECOOD[n1]=ECOOD[n1]-CenCoord[jj];
        }
    }
    return;
}


/*
*******************************************************
* Subroutine HMATRIX                                  *
* - Compute H matrix for each element                 *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void HMATRIX(double ECOOD[],int ELNOD[],int kMATS,
             double PROPS[],double EHMTX[])
{
    void DUCLEAN();
    void GAUSSQU();
    void QUANGAS();
    void TREFFTZ();
    void MATMULT();
    void MATTRAN();
    extern int NTREF,NDIME,NDOFN,NNODE,NEDGE,NGAUS,
        NPROP,NSTRE;
    double *POSGP,*WEIGP,THICK,EXISP,*CORGS,DVOLU,
        *AMTRX,*SHMTX,*N_SET,*T_SET,*Q_SET,*TQ_SET,*QTN;
    int iEDGE,iGAUS,im,jm,n1,NEVAB;

    NEVAB=NNODE*NDOFN;
    // Gaussian point and weight coefficients
    POSGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,POSGP);
    WEIGP=(double *)calloc(NGAUS,sizeof(double));
```

```
DUCLEAN(NGAUS,1,WEIGP);
GAUSSQU(POSGP,WEIGP);
// Material properties
THICK=PROPS[kMATS*NPROP+2];
// Compute H matrix
for(iEDGE=0;iEDGE<NEDGE;iEDGE++)
{
    for(iGAUS=0;iGAUS<NGAUS;iGAUS++)
    {
        EXISP=POSGP[iGAUS];
        //
        CORGS=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CORGS);
        AMTRX=(double *)calloc(NDOFN*NSTRE,
            sizeof(double));
        DUCLEAN(NDOFN,NSTRE,AMTRX);
        SHMTX=(double *)calloc(NDOFN*NEVAB,
            sizeof(double));
        DUCLEAN(NDOFN,NEVAB,SHMTX);
        QUANGAS(iEDGE,EXISP,ECOOD,ELNOD,CORGS,
            &DVOLU,AMTRX,SHMTX);
        DVOLU=DVOLU*WEIGP[iGAUS];
        if((THICK+1)!=1)
        {
            DVOLU=DVOLU*THICK;
        }
        // Trefftz functions
        N_SET=(double *)calloc(NDOFN*NTREF,
            sizeof(double));
        DUCLEAN(NDOFN,NTREF,N_SET);
        T_SET=(double *)calloc(NSTRE*NTREF,
            sizeof(double));
        DUCLEAN(NSTRE,NTREF,T_SET);
        TREFFTZ(CORGS[0],CORGS[1],N_SET,T_SET);
        //
        Q_SET=(double *)calloc(NDOFN*NTREF,
            sizeof(double));
        DUCLEAN(NDOFN,NTREF,Q_SET);
        MATMULT(AMTRX,T_SET,NDOFN,NSTRE,NTREF,Q_SET);
        //
        TQ_SET=(double *)calloc(NTREF*NDOFN,
            sizeof(double));
        DUCLEAN(NTREF,NDOFN,TQ_SET);
        MATTRAN(Q_SET,NDOFN,NTREF,TQ_SET);
```

```
            //
            QTN=(double *)calloc(NTREF*NTREF,
                sizeof(double));
            DUCLEAN(NTREF,NTREF,QTN);
            MATMULT(TQ_SET,N_SET,NTREF,NDOFN,NTREF,QTN);
            for(im=0;im<NTREF;im++)
            {
                for(jm=0;jm<NTREF;jm++)
                {
                    n1=im*NTREF+jm;
                    EHMTX[n1]=EHMTX[n1]+QTN[n1]*DVOLU;
                }
            }
            free(CORGS); free(AMTRX);
            free(SHMTX); free(N_SET);
            free(T_SET); free(Q_SET);
            free(TQ_SET); free(QTN);
        }
    }
    free(POSGP); free(WEIGP);
    return;
}


/*
*********************************************************
* Subroutine GMATRIX                                    *
* - Compute G matrix for each element                   *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void GMATRIX(double ECOOD[],int ELNOD[],int kMATS,
             double PROPS[],double EGMTX[])
{
    void DUCLEAN();
    void GAUSSQU();
    void QUANGAS();
    void TREFFTZ();
    void MATMULT();
    void MATTRAN();
    extern int NTREF,NDIME,NDOFN,NNODE,NEDGE,NGAUS,
        NPROP,NSTRE;
    double *POSGP,*WEIGP,THICK,EXISP,*CORGS,DVOLU,
```

```
    *AMTRX,*SHMTX,*N_SET,*T_SET,*Q_SET,*TQ_SET,*QTS;
int ii,jj,im,jn,n1,NEVAB;

NEVAB=NNODE*NDOFN;
// Gaussian point and weight coefficients
POSGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,POSGP);
WEIGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,WEIGP);
GAUSSQU(POSGP,WEIGP);
// Material properties
THICK=PROPS[kMATS*NPROP+2];
// Compute H matrix
for(ii=0;ii<NEDGE;ii++)
{
    for(jj=0;jj<NGAUS;jj++)
    {
        EXISP=POSGP[jj];
        // Related quantities at Gaussian point
        CORGS=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CORGS);
        AMTRX=(double *)calloc(NDOFN*NSTRE,
            sizeof(double));
        DUCLEAN(NDOFN,NSTRE,AMTRX);
        SHMTX=(double *)calloc(NDOFN*NEVAB,
            sizeof(double));
        DUCLEAN(NDOFN,NEVAB,SHMTX);
        QUANGAS(ii,EXISP,ECOOD,ELNOD,CORGS,&DVOLU,
            AMTRX,SHMTX);
        DVOLU=DVOLU*WEIGP[jj];
        if((THICK+1)!=1)
        {
            DVOLU=DVOLU*THICK;
        }
        // Trefftz functions
        N_SET=(double *)calloc(NDOFN*NTREF,
            sizeof(double));
        DUCLEAN(NDOFN,NTREF,N_SET);
        T_SET=(double *)calloc(NSTRE*NTREF,
            sizeof(double));
        DUCLEAN(NSTRE,NTREF,T_SET);
        TREFFTZ(CORGS[0],CORGS[1],N_SET,T_SET);
        // Q=A*T
        Q_SET=(double *)calloc(NDOFN*NTREF,
```

```
                sizeof(double));
            DUCLEAN(NDOFN,NTREF,Q_SET);
            MATMULT(AMTRX,T_SET,NDOFN,NSTRE,NTREF,Q_SET);
            // Q'
            TQ_SET=(double *)calloc(NTREF*NDOFN,
                sizeof(double));
            DUCLEAN(NTREF,NDOFN,TQ_SET);
            MATTRAN(Q_SET,NDOFN,NTREF,TQ_SET);
            // Q'*SHAPE
            QTS=(double *)calloc(NTREF*NEVAB,
                sizeof(double));
            DUCLEAN(NTREF,NEVAB,QTS);
            MATMULT(TQ_SET,SHMTX,NTREF,NDOFN,NEVAB,QTS);
            //
            for(im=0;im<NTREF;im++)
            {
                for(jn=0;jn<NEVAB;jn++)
                {
                    n1=im*NEVAB+jn;
                    EGMTX[n1]=EGMTX[n1]+QTS[n1]*DVOLU;
                }
            }
            free(CORGS); free(AMTRX);
            free(SHMTX); free(N_SET);
            free(T_SET); free(Q_SET);
            free(TQ_SET); free(QTS);
        }
    }
    free(POSGP); free(WEIGP);
    return;
}


/*
*******************************************************
* Subroutine KMATRIX                                  *
* - Compute element stiffness matrix                  *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void KMATRIX(double HE[],double GE[],double KE[])
{
    void MATINVE();
```

```
    void MATMULT();
    void MATTRAN();
    void DUCLEAN();
    extern int NTREF,NNODE,NDOFN;
    int NEVAB;
    double *TGE,*TGH;

    NEVAB=NNODE*NDOFN;
    // HE: NTREF*NTREF
    // GE: NTREF*NEVAB
    TGE=(double *)calloc(NEVAB*NTREF,sizeof(double));
    DUCLEAN(NEVAB,NTREF,TGE);
    TGH=(double *)calloc(NEVAB*NTREF,sizeof(double));
    DUCLEAN(NEVAB,NTREF,TGH);
    MATINVE(HE,NTREF);
    MATTRAN(GE,NTREF,NEVAB,TGE);
    MATMULT(TGE,HE,NEVAB,NTREF,NTREF,TGH);
    MATMULT(TGH,GE,NEVAB,NTREF,NEVAB,KE);
    free(TGE); free(TGH);
    return;
}


/*
*******************************************************
* Subroutine PVECTOR                                  *
* - Compute effective nodal force                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void PVECTOR(int MATNO[],double PROPS[],int LNODS[],
             double COORD[],int NDLEG,int NEASS[],
             int NOPRS[],double PRESS[],int NPLOD,
             int LODPT[],double POINT[],double GP[])
{
    void GAUSSQU();
    void SHAPFUN();
    void DUCLEAN();
    extern int NDIME,NDOFN,NGAUS,NPROP,NNODE,NEDGE,
        NODEG;
    double *POSGP,*WEIGP,*ELCOD,*EPRES,*PE,*SHAPE,
        *DSHAP,*CORGS,*DERGS,*PGASH;
    int iPLOD,iDLEG,iNODE,kNODE,iODEG,iDOFN,iDIME,
```

```
    jGAUS,kPOIN,kELEM,kMATS,kEQNS,iEVAB,n1,n2,
    NEVAB,TNFEG;
double EXISP,THICK,DVOLU;
// Add point loads at nodes to global load vector
if(NPLOD>0)
{
    for(iPLOD=0;iPLOD<NPLOD;iPLOD++)
    {
        for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
        {
            n1=LODPT[iPLOD]*NDOFN+iDOFN;
            n2=iPLOD*NDOFN+iDOFN;
            GP[n1]=GP[n1]+POINT[n2];
        }
    }
}
// Evaluate equivalent nodal force caused
// by distributed edge load
NEVAB=NNODE*NDOFN;
TNFEG=NODEG*NDOFN;
// Gaussian point and weight coefficients
POSGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,POSGP);
WEIGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,WEIGP);
GAUSSQU(POSGP,WEIGP);
for(iDLEG=0;iDLEG<NDLEG;iDLEG++)
{
    kELEM=NEASS[iDLEG];
    // Material properties
    kMATS=MATNO[kELEM];
    THICK=PROPS[kMATS*NPROP+2];
    // Determine coordinates of nodes on the edge
    ELCOD=(double *)calloc(NODEG*NDIME,
        sizeof(double));
    DUCLEAN(NODEG,NDIME,ELCOD);
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        kPOIN=NOPRS[iDLEG*NODEG+iODEG];
        for(iDIME=0;iDIME<NDIME;iDIME++)
        {
            n1=iODEG*NDIME+iDIME;
            n2=kPOIN*NDIME+iDIME;
            ELCOD[n1]=COORD[n2];
        }
```

```
}
// Determine local nodal load intensity
EPRES=(double *)calloc(NODEG*NDOFN,
    sizeof(double));
DUCLEAN(NODEG,NDOFN,EPRES);
for(iODEG=0;iODEG<NODEG;iODEG++)
{
    for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
    {
        n1=iODEG*NDOFN+iDOFN;
        n2=iDLEG*TNFEG+n1;
        EPRES[n1]=PRESS[n2];
    }
}
// Integration along the loaded edge
PE=(double *)calloc(NEVAB,sizeof(double));
DUCLEAN(NEVAB,1,PE);
for(jGAUS=0;jGAUS<NGAUS;jGAUS++)
{
    EXISP=POSGP[jGAUS];
    // shape function and its derivatives
    SHAPE=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,SHAPE);
    DSHAP=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,DSHAP);
    SHAPFUN(EXISP,SHAPE,DSHAP);
    // geometric quantities of Gaussian points
    // x=sum(Ni*xi) and y=sum(Ni*yi)
    // dx/dt=sum(dNi/dt*xi)
    // dy/dt=sum(dNi/dt*yi)
    // DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    CORGS=(double *)calloc(NDIME,
        sizeof(double));
    DUCLEAN(1,NDIME,CORGS);
    DERGS=(double *)calloc(NDIME,
        sizeof(double));
    DUCLEAN(1,NDIME,DERGS);
    for(iDIME=0;iDIME<NDIME;iDIME++)
    {
        for(iODEG=0;iODEG<NODEG;iODEG++)
        {
            n1=iODEG*NDIME+iDIME;
            CORGS[iDIME]=CORGS[iDIME]+
                SHAPE[iODEG]*ELCOD[n1];
            DERGS[iDIME]=DERGS[iDIME]+
```

```
                DSHAP[iODEG]*ELCOD[n1];
        }
    }
    DVOLU=sqrt(pow(DERGS[0],2.0)+
        pow(DERGS[1],2.0));
    // Gauss integration factor
    DVOLU=DVOLU*WEIGP[jGAUS];
    if((THICK+1)!=1)
    {
        DVOLU=DVOLU*THICK;
    }
    // flux intensity at Gauss point
    //          p=sum(Ni*pi)
    PGASH=(double *)calloc(NDOFN,
        sizeof(double));
    DUCLEAN(NDOFN,1,PGASH);
    for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
    {
        for(iODEG=0;iODEG<NODEG;iODEG++)
        {
            n1=iODEG*NDOFN+iDOFN;
            PGASH[iDOFN]=PGASH[iDOFN]+
                SHAPE[iODEG]*EPRES[n1];
        }
    }
    // Compute equivalent nodal force PE
    for(iNODE=0;iNODE<NNODE;iNODE++)
    {
        kPOIN=LNODS[kELEM*NNODE+iNODE];
        if(kPOIN==NOPRS[iDLEG*NODEG+0])
        {
            // iNODE is start node of the
            // loaded edge
            for(iODEG=0;iODEG<NODEG;iODEG++)
            {
                kNODE=iNODE+iODEG;
                if(kNODE>=NNODE)
                {
                    kNODE=0;
                }
                for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
                {
                    n1=kNODE*NDOFN+iDOFN;
                    PE[n1]=PE[n1]+SHAPE[iODEG]*
                        PGASH[iDOFN]*DVOLU;
```

```
                        }
                    }
                }
            }
        }
        // Assemble PE into global load vector
        for(iNODE=0;iNODE<NNODE;iNODE++)
        {
            kPOIN=LNODS[kELEM*NNODE+iNODE];
            for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
            {
                kEQNS=NDOFN*kPOIN+iDOFN; // global DOF
                iEVAB=NDOFN*iNODE+iDOFN; // local DOF
                GP[kEQNS]=GP[kEQNS]+PE[iEVAB];
            }
        }
    }
    free(POSGP); free(WEIGP); free(ELCOD);
    free(EPRES); free(PE);  free(SHAPE);
    free(DSHAP); free(DERGS); free(PGASH);
    return;
}


/*
*******************************************************
* Subroutine FIEDNOD                                  *
* - Generate nodal generalised displacement field     *
*******************************************************
*/
void FIEDNOD(int NPOIN,double ASDIS[],double UPOIN[])
{
    extern int NDIME,NDOFN;
    int ii,jj,NR;
    for(ii=0;ii<NPOIN;ii++)
    {
        for(jj=0;jj<NDOFN;jj++)
        {
            NR=ii*NDOFN+jj;
            UPOIN[ii*NDOFN+jj]=ASDIS[NR];
        }
    }
    return;
}
```

```
/*
*******************************************************
* Subroutine FIEDCEN                                  *
* -Compute related fields at centroid of each element *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDCEN(int NELEM,int MATNO[],int LNODS[],
             double COORD[],double PROPS[],int ELNOD[],
             double ASDIS[],double CECOD[],
             double UCENP[],double SCENP[])
{
    void ELEPARS(); void DUCLEAN(); void HMATRIX();
    void GMATRIX(); void EDISNOD(); void CMATRIX();
    void RIGIDRV(); void TREFFTZ(); void MATMULT();
    extern int NTREF,NNODE,NDIME,NDOFN,NSTRE;
    int iELEM,kMATS,NEVAB;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*d_Ele,
        *c_Ele,c0,*N_SET,*T_SET,*GDISP,*GSTRE,xp,yp;

    NEVAB=NNODE*NDOFN;
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        // Compute quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,
            sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,
            sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
        HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
        // Compute G matrix
        EGMTX=(double *)calloc(NTREF*NEVAB,
            sizeof(double));
        DUCLEAN(NTREF,NEVAB,EGMTX);
        GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
```

```
        // Nodal displacements
        d_Ele=(double *)calloc(NEVAB,
            sizeof(double));
        DUCLEAN(NEVAB,1,d_Ele);
        EDISNOD(iELEM,LNODS,ASDIS,d_Ele);
        // Calculate the ce coefficients
        c_Ele=(double *)calloc(NTREF*1,
            sizeof(double));
        DUCLEAN(NTREF,1,c_Ele);
        CMATRIX(EHMTX,EGMTX,d_Ele,c_Ele);
        // Recover rigid displacement
        RIGIDRV(ECOOD,c_Ele,d_Ele,&c0);
        // Compute Trefftz internal fields at central point
        N_SET=(double *)calloc(NDOFN*NTREF,
            sizeof(double));
        DUCLEAN(NDOFN,NTREF,N_SET);
        T_SET=(double *)calloc(NSTRE*NTREF,
            sizeof(double));
        DUCLEAN(NSTRE,NTREF,T_SET);
        xp=0;
        yp=0;
        TREFFTZ(xp,yp,N_SET,T_SET);
        GDISP=(double *)calloc(NDOFN*1,sizeof(double));
        DUCLEAN(NDOFN,1,GDISP);
        GSTRE=(double *)calloc(NSTRE*1,sizeof(double));
        DUCLEAN(NSTRE,1,GSTRE);
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,GDISP);
        MATMULT(T_SET,c_Ele,NSTRE,NTREF,1,GSTRE);
        UCENP[iELEM*NDOFN+0]=GDISP[0]+c0;
        SCENP[iELEM*NSTRE+0]=GSTRE[0];
        SCENP[iELEM*NSTRE+1]=GSTRE[1];
        // Coordinates of commputing point
        CECOD[iELEM*NDIME+0]=CenCoord[0]+xp;
        CECOD[iELEM*NDIME+1]=CenCoord[1]+yp;
    }
    free(ECOOD); free(CenCoord); free(EHMTX);
    free(EGMTX); free(d_Ele); free(c_Ele);
    free(N_SET); free(T_SET); free(GDISP);
    free(GSTRE);
}


/*
*******************************************************
* Subroutine QUANGAS                                  *
```

```
* - Evaluate quantities at Gaussian points            *
*****************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void QUANGAS(int iEDGE,double EXISP,double ECOOD[],
             int ELNOD[],double CORGS[],double *DVOLU,
             double AMTRX[],double SHMTX[])
{
    void SHAPFUN();
    void DUCLEAN();
    extern int NDIME,NDOFN,NNODE,NEDGE,NODEG;
    double *SHAPE,*DSHAP,*DERGS;
    int ii,jj,in,kn;

    // shape function and its derivatives
    SHAPE=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,SHAPE);
    DSHAP=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,DSHAP);
    SHAPFUN(EXISP,SHAPE,DSHAP);
    // Coordinates and derivatives of Gauss points
    //   x=sum(Ni*xi) and y=sum(Ni*yi)
    //   dx/dt=sum(dNi/dt*xi), dy/dt=sum(dNi/dt*yi)
    //   DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    DERGS=(double *)calloc(NDIME,sizeof(double));
    DUCLEAN(1,NDIME,DERGS);
    for(ii=0;ii<NDIME;ii++)
    {
        for(jj=0;jj<NODEG;jj++)
        {
            kn=ELNOD[iEDGE*NODEG+jj];
            CORGS[ii]=CORGS[ii]+
                SHAPE[jj]*ECOOD[kn*NDIME+ii];
            DERGS[ii]=DERGS[ii]+
                DSHAP[jj]*ECOOD[kn*NDIME+ii];
        }
    }
    *DVOLU=sqrt(pow(DERGS[0],2.0)+pow(DERGS[1],2.0));
    // Directional cosine at Gaussian point
    //   nx = dy/dS,  ny = - dx/dS
    AMTRX[0]= DERGS[1]/(*DVOLU);
    AMTRX[1]=-DERGS[0]/(*DVOLU);
    // Form shape function matrix of frame function
```

```
    in=ELNOD[iEDGE*NODEG+0];
    for(ii=0;ii<NODEG;ii++)
    {
        kn=in+ii;
        if(kn>=NNODE)
        {
            kn=0;
        }
        SHMTX[kn]=SHAPE[ii];
    }
    return;
}


/*
*********************************************************
* Subroutine TREFFTZ                                    *
* - Evaluate Trefftz functions truncted with            *
*   specified terms number                              *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void TREFFTZ(double sp,double tp,double N_SET[],
             double T_SET[])
{
    extern int NTREF,NNODE,NDOFN;
    int temp,im,n;
    double remaind,r,s;
    // Check the number of terms of Trefftz functions
    temp=NNODE*NDOFN-1;
    if(NTREF<temp)
    {
        printf("Small number of Trefftz functions!");
        exit(0);
    }
    //
    r=sqrt(pow(sp,2.0)+pow(tp,2.0));
    s=atan2(tp,sp);
    for(im=0;im<NTREF;im++)
    {
        // Determine the order of Ni
        n=(int)(ceil((im+1)/2.0));
        // Justify im is odd or even number
```

```
        remaind=fmod(im,2.0);
        if((1+remaind)!=1) // im is even
        {
            N_SET[im]=pow(r,n)*sin(n*s);
            T_SET[0*NTREF+im]=
                n*pow(r,(n-1))*sin((n-1)*s);
            T_SET[1*NTREF+im]=
                n*pow(r,(n-1))*cos((n-1)*s);
        }
        else // im is odd
        {
            N_SET[im]=pow(r,n)*cos(n*s);
            T_SET[0*NTREF+im]=
                n*pow(r,(n-1))*cos((n-1)*s);
            T_SET[1*NTREF+im]=
                -n*pow(r,(n-1))*sin((n-1)*s);
        }
    }
    return;
}


/*
*******************************************************
* Subroutine ASMSTIF                                  *
* - Compute unknown coefficients c in each element    *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void CMATRIX(double EHMTX[],double EGMTX[],
             double d_Ele[],double c_Ele[])
{
    void MATINVE();
    void MATMULT();
    void DUCLEAN();
    extern int NTREF,NNODE,NDOFN;
    double *TEMP;
    int NEVAB;

    NEVAB=NNODE*NDOFN;
    TEMP=(double *)calloc(NTREF*NEVAB,sizeof(double));
    DUCLEAN(NTREF,NEVAB,TEMP);
    MATINVE(EHMTX,NTREF);
```

```
    MATMULT(EHMTX,EGMTX,NTREF,NTREF,NEVAB,TEMP);
    MATMULT(TEMP,d_Ele,NTREF,NEVAB,1,c_Ele);
    free(TEMP);
    return;
}


/*
*******************************************************
* Subroutine RIGIDRV                                  *
* - Recovery of rigid body motion                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void RIGIDRV(double ECOOD[],double c_Ele[],
             double d_Ele[],double *c0)
{
    void DUCLEAN();
    void MATMULT();
    void TREFFTZ();
    extern int NTREF,NDIME,NDOFN,NNODE;
    double sum,xp,yp,*N_SET,*T_SET,*TEMP;
    int iNODE;
    sum=0.0;
    for(iNODE=0;iNODE<NNODE;iNODE++)
    {
        xp=ECOOD[iNODE*NDIME+0];
        yp=ECOOD[iNODE*NDIME+1];
        N_SET=(double *)calloc(1*NTREF,sizeof(double));
        DUCLEAN(1,NTREF,N_SET);
        T_SET=(double *)calloc(2*NTREF,sizeof(double));
        DUCLEAN(2,NTREF,T_SET);
        TREFFTZ(xp,yp,N_SET,T_SET);
        TEMP=(double *)calloc(1*1,sizeof(double));
        DUCLEAN(1,1,TEMP);
        MATMULT(N_SET,c_Ele,1,NTREF,1,TEMP);
        sum=sum+(d_Ele[iNODE]-TEMP[0]);
    }
    *c0=sum/(NNODE*1.0);
    free(N_SET); free(T_SET);
    return;
}
```

```
/*
*******************************************************
* Subroutine EDISNOD                                  *
* - Collect nodal displacements in a given element    *
*******************************************************
*/
void EDISNOD(int iELEM,int LNODS[],double ASDIS[],
             double d_Ele[])
{
    extern int NDIME,NDOFN,NNODE;
    int iNODE,kPOIN,iDOFN,kGR,iLR;
    for(iNODE=0;iNODE<NNODE;iNODE++)
    {
        kPOIN=LNODS[iELEM*NNODE+iNODE];
        for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
        {
            kGR=kPOIN*NDOFN+iDOFN;
            iLR=iNODE*NDOFN+iDOFN;
            d_Ele[iLR]=ASDIS[kGR];
        }
    }
    return;
}


/*
*******************************************************
* Subroutine MATTRAN, MATMULT, MATINVE                *
* - Matrix transpose, multiply and inverse operations *
*******************************************************
*/
/* matrix transpose */
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void MATTRAN(double a[],int m,int n,double b[])
{// a[m][n]   b[n][m]
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            b[i*m+j]=a[j*n+i];
        }
```

```
    }
    return;
}


/* matrix multiply */
void MATMULT(double a[],double b[],int m,int n,
             int k,double c[])
{
    // a[m][n] * b[n][k] = c[m][k]
    int i,j,l;
    for(i=0;i<=m-1;i++)
    {
        for(j=0;j<=k-1;j++)
        {
            c[i*k+j]=0.0;
            for(l=0;l<=n-1;l++)
            {
                c[i*k+j]=c[i*k+j]+a[i*n+l]*b[l*k+j];
            }
        }
    }
    return;
}


/* inverse matrix */
void MATINVE(double a[],int n)
{
    int *is,*js,i,j,k,l,u,v;
    double d,p,eps;
    eps=1.0e-20;
    is=malloc(n*sizeof(int));
    js=malloc(n*sizeof(int));
    for(k=0;k<=n-1;k++)
    {
        d=0.0;
        for(i=k;i<=n-1;i++)
        {
            for(j=k;j<=n-1;j++)
            {
                l=i*n+j;
                p=fabs(a[l]);
                if(p>d)
                {
```

```
                d=p;
                is[k]=i;
                js[k]=j;
            }
        }
    }
    if((d+1)==1)
    {
        free(is);
        free(js);
        printf("**The inversion of matrix fail!\n");
        exit(0);
    }
    if(is[k]!=k)
    {
        for(j=0;j<=n-1;j++)
        {
            u=k*n+j;
            v=is[k]*n+j;
            p=a[u];
            a[u]=a[v];
            a[v]=p;
        }
    }
    if(js[k]!=k)
    {
        for(i=0;i<=n-1;i++)
        {
            u=i*n+k;
            v=i*n+js[k];
            p=a[u];
            a[u]=a[v];
            a[v]=p;
        }
    }
    l=k*n+k;
    a[l]=1.0/a[l];
    for(j=0;j<=n-1;j++)
    {
        if(j!=k)
        {
            u=k*n+j;
            a[u]=a[u]*a[l];
        }
    }
```

```
        for(i=0;i<=n-1;i++)
        {
            if(i!=k)
            {
                for(j=0;j<=n-1;j++)
                {
                    if (j!=k)
                    {
                        u=i*n+j;
                        a[u]=a[u]-a[i*n+k]*a[k*n+j];
                    }
                }
            }
        }
        for(i=0;i<=n-1;i++)
        {
            if(i!=k)
            {
                u=i*n+k;
                a[u]=-a[u]*a[l];
            }
        }
    }
    for(k=n-1;k>=0;k--)
    {
        if (js[k]!=k)
        {
            for(j=0;j<=n-1;j++)
            {
                u=k*n+j;
                v=js[k]*n+j;
                p=a[u];
                a[u]=a[v];
                a[v]=p;
            }
        }
        if(is[k]!=k)
        {
            for(i=0;i<=n-1;i++)
            {
                u=i*n+k;
                v=i*n+is[k];
                p=a[u];
                a[u]=a[v];
                a[v]=p;
```

```
            }
        }
    }
    free(is);
    free(js);
    return;
}


/*
*********************************************************
* Subroutine ITCLEAN, DUCLEAN                           *
* - Integer and double type array initialization        *
*********************************************************
*/
void ITCLEAN(int m,int n,int matrix[])
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            matrix[i*n+j]=0;
        }
    }
    return;
}

void DUCLEAN(int m,int n,double matrix[])
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            matrix[i*n+j]=0.0;
        }
    }
    return;
}
```

5.10 Numerical examples

In this section, two examples are considered and solved by HT-FEM using the MATLAB or C program provided above to investigate the accuracy, stability and convergence of the HT-FEM. The first example is used to investigate the effect of mesh distortion, because in practical engineering many irregular meshes are employed rather than regular ones. The second example is used to compare the effect of improved mesh, and the typical 4-node element and 8-node element are used for this purpose.

Example 5.1 Square domain
Consider a Laplace problem in a square domain of size 1×1. The corresponding boundary conditions are shown in Figure 5.6. The exact solution of this simple problem is $u = X_2$ and $\frac{\partial u}{\partial n} = n_2$.

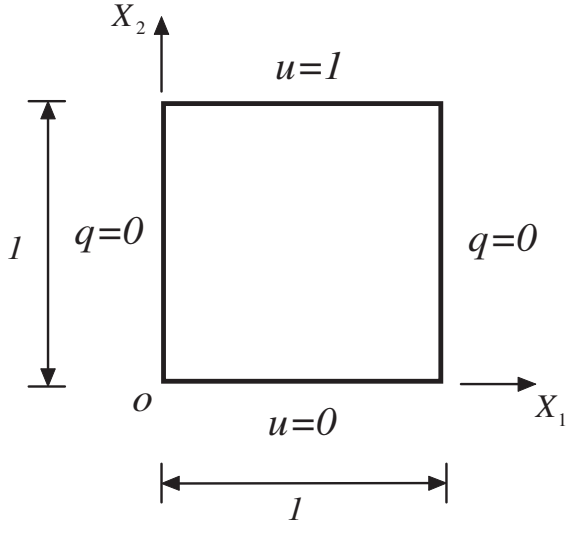


FIGURE 5.6
A Laplace problem in 1×1 square domain

In our computation, the entire domain is discretised by four 4-node elements with linear frame functions. In this example the number of truncated terms of T-complete functions is selected as 6.

Firstly, the regular mesh shown in Figure 5.7 is employed. The input data prepared and results at all nodes and centroids of each element are:

```
*************************************************
Hybrid Trefftz FEM for 2D Laplace problems-Example 1
*************************************************
NTREF NTYPE
6     0
NNODE NEDGE NODEG
4     4     2
NDIME NDOFN NSTRE
2     1     2
NMATS NPROP NGAUS
1     3     5
NPOIN NELEM NVFIX NPLOD NDLEG
9     4     6     0     4
-- Read elementary connections and material numbers
Elem# Mat# Node#1--->#NNODE
1     1    1   2   5   4
2     1    2   3   6   5
3     1    6   9   8   5
4     1    5   8   7   4
-- Read nodal coordinates
Node# Coord#1--->#NDIME
1     0.0     0.0
2     0.5     0.0
3     1.0     0.0
4     0.0     0.5
5     0.5     0.5
6     1.0     0.5
7     0.0     1.0
8     0.5     1.0
9     1.0     1.0
-- Read constrained boundary conditions
Num# Node# DOF#1--->#NDOFN Val#1--->#NDOFN
1    1     1      0.0
2    2     1      0.0
3    3     1      0.0
4    7     1      1.0
5    8     1      1.0
6    9     1      1.0
-- No Concentrated flux
-- Distributed edge flux qn
Num# Ele# Node#1-->#2 Value#qn1-->#qn2
1    1    4     1     0.0     0.0
2    4    7     4     0.0     0.0
3    2    3     6     0.0     0.0
4    3    6     9     0.0     0.0
```

```
-- Read material properties
Mat# Pro#1--->#NPROP
1    1.0   0.3   1.0


==========================================
**Basic parameters

NTREF=    6  NTYPE=    0

NDIME=    2  NDOFN=    1  NSTRE=    2

NNODE=    4  NEDGE=    4  NODEG=    2

NMATS=    1  NPROP=    3  NGAUS=    5

NPOIN=    9  NELEM=    4  NVFIX=    6

NPLOD=    0  NDLEG=    4


** Nodal generalised displacements
---------------------------------
Num# COD#1-->#NDIME DIS#1-->#NODFN
---------------------------------
    1     0.0000     0.0000     0.0000
    2     0.5000     0.0000     0.0000
    3     1.0000     0.0000     0.0000
    4     0.0000     0.5000     0.5000
    5     0.5000     0.5000     0.5000
    6     1.0000     0.5000     0.5000
    7     0.0000     1.0000     1.0000
    8     0.5000     1.0000     1.0000
    9     1.0000     1.0000     1.0000


**Central generalised displacements
---------------------------------
Elem# COD#1-->#NDIME DIS#1-->#NODFN
---------------------------------
    1     0.2500     0.2500     0.2500
    2     0.7500     0.2500     0.2500
    3     0.7500     0.7500     0.7500
    4     0.2500     0.7500     0.7500
```

```
** Central generalised stresses
-------------------------
Elem#    STRES#1-->#NSTRE
-------------------------
    1     0.0000     1.0000
    2    -0.0000     1.0000
    3     0.0000     1.0000
    4    -0.0000     1.0000
```

It is evident that the numerical results are in good agreement with the analytical solutions.

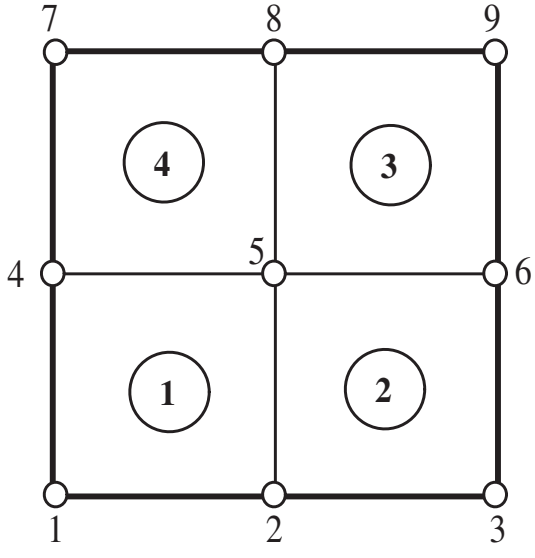


FIGURE 5.7
Configuration of regular meshes

Secondly, to demonstrate the influence of element distortion on the HT-FEM results, the irregular mesh shown in Figure 5.8 is employed. The corresponding data preparation and numerical results at all nodes and four centroids of elements are listed below.

```
*************************************************
Hybrid Trefftz FEM for 2D Laplace problems-Example 1
*************************************************
```

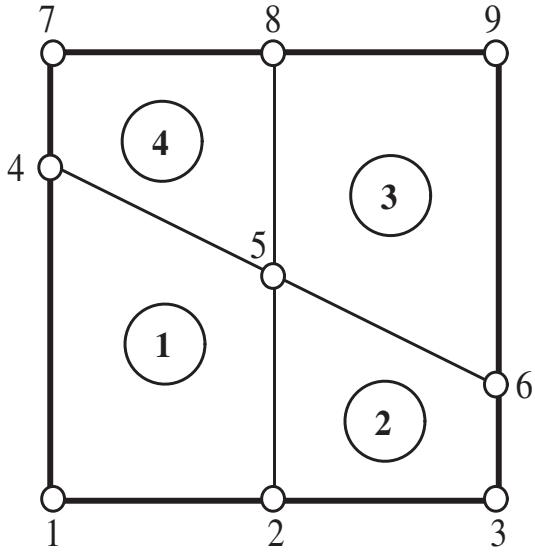


FIGURE 5.8
Configuration of irregular meshes

```
NTREF NTYPE
6     0
NNODE NEDGE NODEG
4     4     2
NDIME NDOFN NSTRE
2     1     2
NMATS NPROP NGAUS
1     3     5
NPOIN NELEM NVFIX NPLOD NDLEG
9     4     6     0     4
-- Read elementary connections and material numbers
Elem# Mat# Node#1--->#NNODE
1     1    1   2   5   4
2     1    2   3   6   5
3     1    6   9   8   5
4     1    5   8   7   4
-- Read nodal coordinates
Node# Coord#1--->#NDIME
1     0.00     0.00
2     0.50     0.00
3     1.00     0.00
```

```
4     0.00     0.75
5     0.50     0.50
6     1.00     0.25
7     0.00     1.00
8     0.50     1.00
9     1.00     1.00
-- Read constrained boundary conditions
Num# Node# DOF#1--->#NDOFN Val#1--->#NDOFN
1    1     1      0.0
2    2     1      0.0
3    3     1      0.0
4    7     1      1.0
5    8     1      1.0
6    9     1      1.0
-- No Concentrated flux
-- Distributed edge flux qn
Num# Ele# Node#1-->#2 Value#qn1-->#qn2
1    1    4     1     0.0,    0.0
2    4    7     4     0.0,    0.0
3    2    3     6     0.0,    0.0
4    3    6     9     0.0,    0.0
-- Read material properties
Mat# Pro#1--->#NPROP
1    1.0   0.3   1.0


===============================================
**Basic parameters

NTREF=    6  NTYPE=    0

NDIME=    2  NDOFN=    1  NSTRE=    2

NNODE=    4  NEDGE=    4  NODEG=    2

NMATS=    1  NPROP=    3  NGAUS=    5

NPOIN=    9  NELEM=    4  NVFIX=    6

NPLOD=    0  NDLEG=    4


** Nodal generalised displacements
---------------------------------
Num# COD#1-->#NDIME DIS#1-->#NODFN
```

```
---------------------------------
    1     0.0000     0.0000     0.0000
    2     0.5000     0.0000     0.0000
    3     1.0000     0.0000     0.0000
    4     0.0000     0.7500     0.7500
    5     0.5000     0.5000     0.5000
    6     1.0000     0.2500     0.2500
    7     0.0000     1.0000     1.0000
    8     0.5000     1.0000     1.0000
    9     1.0000     1.0000     1.0000


**Central generalised displacements
---------------------------------
Elem# COD#1-->#NDIME DIS#1-->#NODFN
---------------------------------
    1     0.2500     0.3125     0.3125
    2     0.7500     0.1875     0.1875
    3     0.7500     0.6875     0.6875
    4     0.2500     0.8125     0.8125

** Central generalised stresses
------------------------
Elem#    STRES#1-->#NSTRE
------------------------
    1     0.0000     1.0000
    2    -0.0000     1.0000
    3     0.0000     1.0000
    4    -0.0000     1.0000
```

It can be seen that the results of potential and flux at nodes and centroids of the elements are again in good agreement with the analytical results. This indicates that the results for HT-FEM potential elements are insensitive to mesh distortion.

Example 5.2 Cylinder domain
The second example is for the potential problem with a hollow cylinder domain. On all boundaries, the following Dirichlet conditions are applied:

$$u = u_o \qquad when \;\; r = r_o$$
$$u = u_i \qquad when \;\; r = r_i$$

The corresponding analytical solutions of potential and its derivatives are given by

$$u = a + b \ln r$$

$$\frac{\partial u}{\partial X_1} = b\frac{X_1}{r^2}, \qquad \frac{\partial u}{\partial X_2} = b\frac{X_2}{r^2}$$

with

$$a = \frac{u_i \ln(r_o) - u_o \ln(r_i)}{\ln(r_o/r_i)}, \quad b = \frac{u_o - u_i}{\ln(r_o/r_i)}$$

Due to the axisymmetric properties of the geometry and boundary conditions, only one quarter of the cylinder domain is considered, and the corresponding boundary conditions are shown in Figure 5.9.

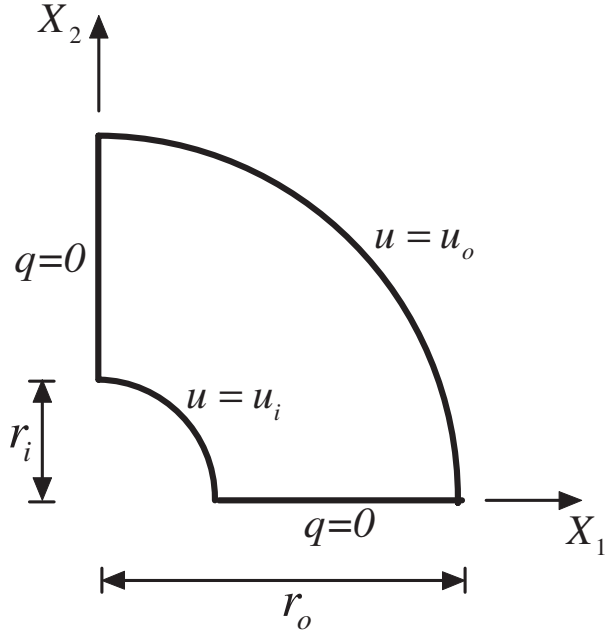


FIGURE 5.9
Configuration of one quarter of the cylinder domain and related boundary conditions

In our computation, $r_i = 5$, $r_o = 20$, $u_i = 10$, $u_o = 0$ are employed. In this example, two types of element (one is a 4-node element and the other an 8-node element) are used to discretise the solution domain, and the total number of elements is 9 (see Figure 5.10 and Figure 5.11). The purpose is to study the effect of element type on accuracy. Taking into consideration the requirement of minimum terms of Trefftz functions discussed in Section 5.3, we here set $m = 8$. The corresponding results are listed in Table 5.1 and Table 5.2, from which it is evident that the numerical accuracy at the centroids of the elements is significantly improved as the type of element changes from a 4-node linear element to an 8-node nonlinear element, although the

same number of Trefftz complete functions and the same number of elements are employed.

The complete list of input data and results for the typical 8-node element are also provided for reference:

```
**************************************************
Hybrid Trefftz FEM for 2D Laplace problems-Example 2
**************************************************
NTREF NTYPE
8     0
NNODE NEDGE NODEG
8     4     3
NDIME NDOFN NSTRE
2     1     2
NMATS NPROP NGAUS
1     3     5
NPOIN NELEM NVFIX NPLOD NDLEG
40    9     14    0     6
-- Read elementary connections and material numbers
Elem# Mat# Node#1--->#NNODE
1     1     1   2   3   9  14  13  12   8
2     1     3   4   5  10  16  15  14   9
3     1     5   6   7  11  18  17  16  10
4     1    12  13  14  20  25  24  23  19
5     1    14  15  16  21  27  26  25  20
6     1    16  17  18  22  29  28  27  21
7     1    23  24  25  31  36  35  34  30
8     1    25  26  27  32  38  37  36  31
9     1    27  28  29  33  40  39  38  32
-- Read nodal coordinates
Node# Coord#1--->#NDIME
 1     5.000     0.000
 2     6.667     0.000
 3     8.333     0.000
 4    10.667     0.000
 5    13.000     0.000
 6    16.500     0.000
 7    20.000     0.000
 8     4.830     1.294
 9     8.049     2.157
10    12.557     3.365
11    19.319     5.176
12     4.330     2.500
13     5.774     3.333
14     7.217     4.167
```

```
15     9.238     5.333
16    11.258     6.500
17    14.289     8.250
18    17.321    10.000
19     3.536     3.536
20     5.893     5.893
21     9.192     9.192
22    14.142    14.142
23     2.500     4.330
24     3.333     5.774
25     4.167     7.217
26     5.333     9.238
27     6.500    11.258
28     8.250    14.289
29    10.000    17.321
30     1.294     4.830
31     2.157     8.049
32     3.365    12.557
33     5.176    19.319
34     0.000     5.000
35     0.000     6.667
36     0.000     8.333
37     0.000    10.667
38     0.000    13.000
39     0.000    16.500
40     0.000    20.000
-- Read constrained boundary conditions
Num# Node# DOF#1--->#NDOFN Val#1--->#NDOFN
 1    1    1     10.0
 2    8    1     10.0
 3   12    1     10.0
 4   19    1     10.0
 5   23    1     10.0
 6   30    1     10.0
 7   34    1     10.0
 8    7    1      0.0
 9   11    1      0.0
10   18    1      0.0
11   22    1      0.0
12   29    1      0.0
13   33    1      0.0
14   40    1      0.0
-- No Concentrated load
-- Distributed edge load
Num# Ele# Node#1-->#2 Value#qn1-->#qn2
```

```
1    1     1   2   3  0.0  0.0  0.0
2    2     3   4   5  0.0  0.0  0.0
3    3     5   6   7  0.0  0.0  0.0
4    9    40  39  38  0.0  0.0  0.0
5    8    38  37  36  0.0  0.0  0.0
6    7    36  35  34  0.0  0.0  0.0
-- Read material properties
Mat# Pro#1--->#NPROP
1    1.0   0.3   1.0


===============================================
**Basic parameters

NTREF=    8  NTYPE=    0

NDIME=    2  NDOFN=    1  NSTRE=    2

NNODE=    8  NEDGE=    4  NODEG=    3

NMATS=    1  NPROP=    3  NGAUS=    5

NPOIN=   40  NELEM=    9  NVFIX=   14

NPLOD=    0  NDLEG=    6


**Generalised displacements at nodes
-------------------------------------
Node# COD#1-->#NDIME DISPL#1-->#NODFN
-------------------------------------
    1     5.0000     0.0000    10.0000
    2     6.6670     0.0000     7.9255
    3     8.3330     0.0000     6.3101
    4    10.6670     0.0000     4.5356
    5    13.0000     0.0000     3.1010
    6    16.5000     0.0000     1.3889
    7    20.0000     0.0000     0.0000
    8     4.8300     1.2940    10.0000
    9     8.0490     2.1570     6.3187
   10    12.5570     3.3650     3.1111
   11    19.3190     5.1760     0.0000
   12     4.3300     2.5000    10.0000
   13     5.7740     3.3330     7.9256
   14     7.2170     4.1670     6.3096
```

```
   15     9.2380     5.3330     4.5357
   16    11.2580     6.5000     3.1011
   17    14.2890     8.2500     1.3890
   18    17.3210    10.0000     0.0000
   19     3.5360     3.5360    10.0000
   20     5.8930     5.8930     6.3179
   21     9.1920     9.1920     3.1114
   22    14.1420    14.1420     0.0000
   23     2.5000     4.3300    10.0000
   24     3.3330     5.7740     7.9256
   25     4.1670     7.2170     6.3096
   26     5.3330     9.2380     4.5357
   27     6.5000    11.2580     3.1011
   28     8.2500    14.2890     1.3890
   29    10.0000    17.3210     0.0000
   30     1.2940     4.8300    10.0000
   31     2.1570     8.0490     6.3187
   32     3.3650    12.5570     3.1111
   33     5.1760    19.3190     0.0000
   34     0.0000     5.0000    10.0000
   35     0.0000     6.6670     7.9255
   36     0.0000     8.3330     6.3101
   37     0.0000    10.6670     4.5356
   38     0.0000    13.0000     3.1010
   39     0.0000    16.5000     1.3889
   40     0.0000    20.0000     0.0000


**Generalised Displacement at Element Centroid
-------------------------------------
Elem# COD#1-->#NDIME DISPL#1-->#NODFN
-------------------------------------
    1     6.2750     1.6814     8.1095
    2    10.0399     2.6903     4.7190
    3    15.5305     4.1614     1.5720
    4     4.5938     4.5938     8.1093
    5     7.3498     7.3498     4.7189
    6    11.3690    11.3690     1.5720
    7     1.6814     6.2750     8.1095
    8     2.6903    10.0399     4.7190
    9     4.1614    15.5305     1.5720

**Generalised Stress at Element Centroid
------------------------
Elem#    STRES#1-->#NSTRE
```

```
-------------------------
    1     -1.0723     -0.2873
    2     -0.6695     -0.1794
    3     -0.4330     -0.1160
    4     -0.7850     -0.7850
    5     -0.4901     -0.4901
    6     -0.3170     -0.3170
    7     -0.2873     -1.0723
    8     -0.1794     -0.6695
    9     -0.1160     -0.4330
```

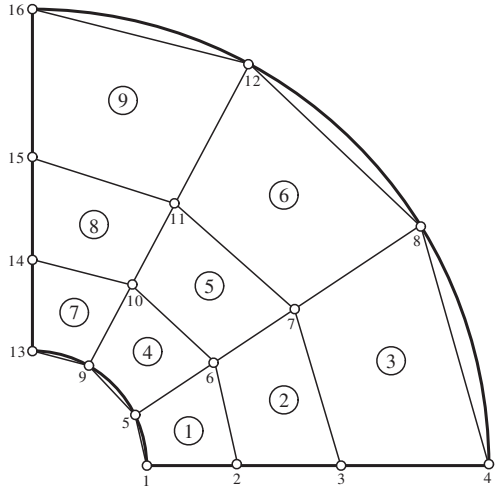


FIGURE 5.10
Configuration of domain discretisation with 4-node elements

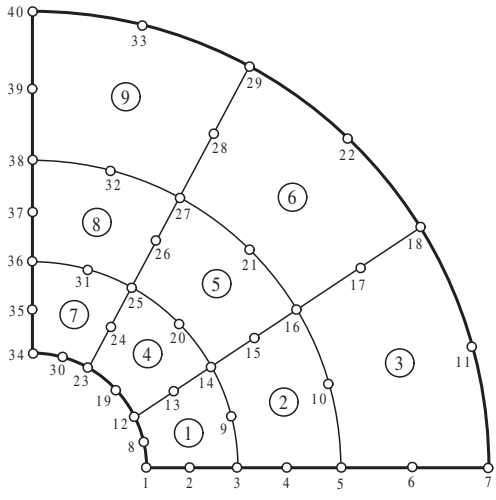


FIGURE 5.11
Configuration of domain discretisation with 8-node elements

TABLE 5.1
Results at centroids of all 4-node elements

Number	$Rerr(f) = \left\lvert \frac{f_{numerical} - f_{exact}}{f_{exact}} \right\rvert \times 100\%$		
	$Rerr(u)$	$Rerr\left(\partial u/\partial X_1\right)$	$Rerr\left(\partial u/\partial X_2\right)$
1	0.1326	1.8018	1.8012
2	1.3107	1.8003	1.8086
3	4.8821	1.8016	1.7900
4	0.1319	1.7992	1.7992
5	1.3105	1.8011	1.8011
6	4.8789	1.8010	1.8010
7	0.1326	1.8012	1.8018
8	1.3107	1.8086	1.8003
9	4.8821	1.7900	1.8016

TABLE 5.2
Results at centroids of all 8-node elements

Number	$Rerr(f) = \left\lvert \frac{f_{numerical} - f_{exact}}{f_{exact}} \right\rvert \times 100\%$		
	$Rerr(u)$	$Rerr\left(\partial u/\partial X_1\right)$	$Rerr\left(\partial u/\partial X_2\right)$
1	0.0247	0.0255	0.0263
2	0.0472	0.1297	0.1303
3	0.1546	0.0924	0.0937
4	0.0239	0.0198	0.0198
5	0.0470	0.1287	0.1287
6	0.1575	0.0882	0.0882
7	0.0247	0.0263	0.0255
8	0.0472	0.1303	0.1297
9	0.1546	0.0937	0.0924

References

[1] Courant R, Hilbert D (1953), *Methods of Mathematical Physics* (Vol I). New York: Inter-science.

[2] Courant R, Hilbert D (1962), *Methods of Mathematical Physics* (Vol II). New York: Inter-science.

[3] Dautray R, Lions JL (1990), *Mathematical Analysis and Numerical Methods for Science and Technology*. Berlin: Springer-Verlag.

[4] Fredholm EI (1903), Sur une classe d'equations fonctionelles. *Acta Math*, **27**: 365-390.

[5] Zielinski AP, Zienkiewicz OC (1985), Generalised finite element analysis with T-complete boundary solution functions. *Int J Numer Meth Eng*, **21**: 509-528.

[6] Cheung YK, Jin WG, Zienkiewicz OC (1989), Direct solution procedure for solution of harmonic problems using complete, non-singular, Trefftz functions. *Commun Appl Numer Meth*, **5**: 159-169.

[7] Cheung YK, Jin WG, Zienkiewicz OC (1991), Solution of Helmholtz equation by Trefftz method. *Int J Numer Meth Eng*, **32**: 63-78.

[8] Jirousek J, Wroblewski A (1994), Least-squares T-elments: Equivalent FE and BE forms of a substructure-oriented boundary solution approach. *Commun Appl Numer Meth*, **10**: 21-32.

[9] Jirousek J, Stojek M (1995), Numerical assessment of a new T-element approach. *Comput Struct*, **57**: 367-378.

[10] Stojek M (1998), Least-squares Trefftz-type elements for the Helmholtz equation. *Int J Numer Meth Eng*, **41**: 831-849.

[11] Wang H, Qin QH, Arounsavat D (2007), Application of hybrid Trefftz finite element method to non-linear problems of minimal surface. *Int J Numer Meth Eng*, **69**: 1262-1277.

[12] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*. Southampton: WIT Press.

[13] Jirousek J, Guex L (1986), The hybrid-Trefftz finite element model and its application to plate bending. *Int J Numer Meth Eng*, **23**: 651-693.

[14] Colton DL (1976), *Partial Differential Equations in the Complex Domain*. London: Pitman.

[15] Henrici P (1960), Complete systems of solutions for a class of singular elliptic partial differential equations. In: *Boundary Problems in Differential Equations*, ed. Langer RE. University of Wisconsin Press.

6

Plane stress/strain problems

6.1 Introduction

In the previous chapter, applications of T-elements to potential problems were presented. Extension of the programming procedure to plane elasto-statics is described in this chapter. In most engineering problems, solutions of displacement and stress distributions in linear elastic continua are required, in which the assumption of small strain is usually employed to maintain the linearity of the problem. Special cases of such problems may range from two-dimensional plane stress or strain distributions, axisymmetric solids, plate and shell bending, to fully three-dimensional solids. Because of the difficulty of deriving analytical solutions, engineers resort to numerical approaches such as HT-FEM discussed throughout this book.

The early application of HT-FEM to plane elasticity was reported by Jirousek and Teodorescu [1]. Their paper dealt with two alternative variational formulations of HT plane elasticity elements, depending upon whether the auxiliary frame function displacement field is assumed along the whole element boundary or is confined only to the inter-element portion. Subsequently, various versions of HT plane elasticity elements were investigated by several researchers [2 - 5]. Additionally, a family of *p*-method plane elasticity elements was derived based on the HT-FE formulation by Jirousek and Venkatesh [6], and suitable special-purpose Trefftz functions were developed to treat singularity and/or stress concentration problems avoiding a local mesh refinement. Most of the developments in this field can also be found from review papers by Jirousek and Wroblewski [7] and Qin [8].

In this chapter, the programming application of HT-FEM to plane linear elasticity is presented in detail and the corresponding computer codes are provided in both MATLAB and C language.

6.2 Linear theory of elasticity

In linear elastic theory, it is assumed that the material of interest behaves linearly and that changes in orientation of the body in the deformed state are negligible. As a result, linear strain displacement relations can be used and equilibrium equations

refer to the undeformed geometry [9].

For the sake of convenience, throughout this chapter, the rectangular Cartesian coordinates system (X_1, X_2) is used (see Figure 6.1).

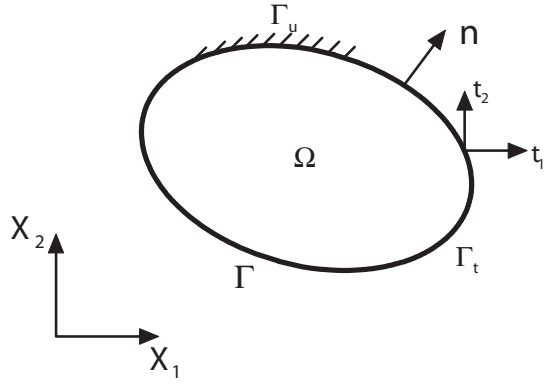


FIGURE 6.1
Configuration of plane linear elasticity

6.2.1 Equilibrium equations

For a plane stress/strain problem, deformation behaviour is governed by the following equilibrium equation:

$$\mathbf{L}\sigma + \mathbf{b} = \mathbf{0} \tag{6.1}$$

where $\sigma = [\sigma_{11} \quad \sigma_{22} \quad \sigma_{12}]^{\mathrm{T}}$ is the stress vector, $\mathbf{b} = [b_1 \quad b_2]^{\mathrm{T}}$ is the body force vector, and the differential operator matrix $\mathbf{L}$ is given by

$$\mathbf{L} = \begin{bmatrix} \frac{\partial}{\partial X_1} & 0 & \frac{\partial}{\partial X_2} \\ 0 & \frac{\partial}{\partial X_2} & \frac{\partial}{\partial X_1} \end{bmatrix} \tag{6.2}$$

6.2.2 Strain-displacement relations

The strain is related to displacement by the geometry equation.

$$\varepsilon = \mathbf{L}^{\mathrm{T}}\mathbf{u} \tag{6.3}$$

where $\varepsilon = [\varepsilon_{11} \quad \varepsilon_{22} \quad \gamma_{12}]^{\mathrm{T}}$ is the strain vector and $\mathbf{u} = [u_1 \quad u_2]^{\mathrm{T}}$ the displacement vector.

6.2.3 Constitutive relations (stress-strain relations)

In two-dimensional elastic problems, the stress-strain relation is written in compact matrix form as

$$\sigma = \mathbf{D}\varepsilon \tag{6.4}$$

where **D** is the material coefficient matrix with constant components for the case of isotropic homogeneous material.

Generally there are two different cases for plane linear elasticity: one is the plane strain case, and the other is the plane stress case. The corresponding coefficient matrix **D** is different for the different cases. The matrix D for each case is presented below.

- Plane strain

The assumption of plane strain is applicable for bodies which are long enough and whose geometry and loading do not vary significantly in the longitudinal direction. For example, analysis of dams or long cylinders can be performed using this assumption (see Figure 6.2). For such cases, the material coefficient matrix **D** has the form

$$\mathbf{D} = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{6.5}$$

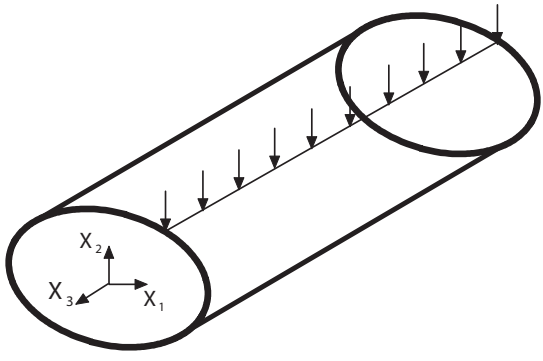


FIGURE 6.2
Illustration of plane strain cases

- Plane stress

The assumption of plane stress is applicable for bodies with a very small dimension in one of the coordinate directions (see Figure 6.3). Thus the analysis of thin plates

loaded in the plane can be made using the assumption of plane stress. For such cases, the coefficient matrix **D** has the form

$$\mathbf{D} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{(1-\nu)}{2} \end{bmatrix} \tag{6.6}$$

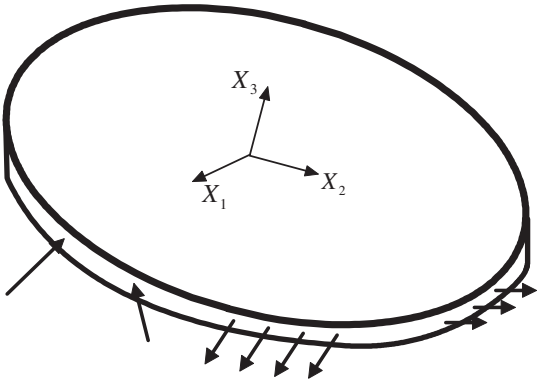


FIGURE 6.3
Illustration of plane stress cases

It can be seen from Eqs. (6.5) and (6.6) that **D** is dependent on two material parameters only: Young's modulus E and Poisson's ratio ν in an isotropic homogeneous material.

For simplicity, Eqs. (6.5) and (6.6) can be uniformly written as

$$\mathbf{D} = \begin{bmatrix} \tilde{\lambda}+2G & \tilde{\lambda} & 0 \\ \tilde{\lambda} & \tilde{\lambda}+2G & 0 \\ 0 & 0 & G \end{bmatrix} \tag{6.7}$$

where

$$\tilde{\lambda} = \frac{2\tilde{\nu}}{1-2\tilde{\nu}}G, \qquad G = \frac{E}{2(1+\nu)}$$

and

$$\tilde{\nu} = \begin{cases} \nu, & \text{for plane strain} \\ \dfrac{\nu}{1+\nu}, & \text{for plane stress} \end{cases}$$

6.2.4 Boundary conditions

Boundary conditions may apply either to displacements or tractions. Displacement boundary conditions require certain displacements to prevail at given points on the

boundary of the body, whereas traction boundary conditions require that the tractions induced must be in equilibrium with the external forces applied at specified points on the boundary of the body. These two categories of boundary condition can be mathematically written as

$$\begin{aligned} \mathbf{u} &= \bar{\mathbf{u}} && \text{on } \Gamma_u \\ \mathbf{t} &= \mathbf{A}\sigma = \bar{\mathbf{t}} && \text{on } \Gamma_t \end{aligned} \tag{6.8}$$

in which $\mathbf{t} = [t_1 \quad t_2]^{\mathrm{T}}$ denotes the traction vector and $\mathbf{A}$ is a transformation matrix related to the direction cosine of the outward normal $\mathbf{n} = [n_1 \quad n_2]^{\mathrm{T}}$:

$$\mathbf{A} = \begin{bmatrix} n_1 & 0 & n_2 \\ 0 & n_2 & n_1 \end{bmatrix} \tag{6.9}$$

6.2.5 Governing equations in terms of displacements

Substituting Eqs. (6.3) and (6.4) into Eq. (6.1) yields the well-known second-order Navier partial differential equations in terms of displacements

$$\mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{u} + \mathbf{b} = \mathbf{0} \tag{6.10}$$

6.3 Trefftz finite element formulation

To obtain a HT-FE solution of the system consisting of Eqs. (6.8) and (6.10), as in the conventional FEM, the solution domain Ω (see Figure 6.1) is divided into a series of elements. Over each element, two independent fields are assumed in the manner presented in Ref. [10]. For simplicity, body force terms are omitted in the following writing. The treatment of body force is discussed in Chapter 7.

6.3.1 Non-conforming intra-element field

The intra-displacement field

$$\mathbf{u}_e = \begin{bmatrix} N_1^1 & N_2^1 & \cdots & N_m^1 \\ N_1^2 & N_2^2 & \cdots & N_m^2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \mathbf{N}_e \mathbf{c}_e \tag{6.11}$$

is assumed in the domain Ω_e (see Figure 6.4), where

$$\mathbf{N}_e = \begin{bmatrix} N_1^1 & N_2^1 & \cdots & N_m^1 \\ N_1^2 & N_2^2 & \cdots & N_m^2 \end{bmatrix} = [\mathbf{N}_1 \quad \mathbf{N}_2 \quad \cdots \quad \mathbf{N}_m] \tag{6.12}$$

is a set of complete solutions of the homogeneous Navier equations in terms of displacements which satisfies

$$\mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{N}_j = \mathbf{0} \tag{6.13}$$

with $\mathbf{N}_j = \begin{bmatrix} N_j^1 & N_j^2 \end{bmatrix}^{\mathrm{T}}$. $\mathbf{c}_e = [c_1 \quad c_2 \quad \cdots \quad c_m]^{\mathrm{T}}$ is a vector of undetermined coefficients with m components.

The expression of stress is obtained by substituting intra-element displacement field Eq. (6.11) into Eqs. (6.3) and (6.4), which yields

$$\sigma_e = \mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{u}_e = \mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{N}_e\mathbf{c}_e = \mathbf{T}_e\mathbf{c}_e \tag{6.14}$$

where

$$\mathbf{T}_e = \mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{N}_e \tag{6.15}$$

Similarly, the tractions on the boundary can be written as

$$\mathbf{t}_e = \mathbf{A}\sigma_e = \mathbf{A}\mathbf{T}_e\mathbf{c}_e = \mathbf{Q}_e\mathbf{c}_e \tag{6.16}$$

with

$$\mathbf{Q}_e = \mathbf{A}\mathbf{T}_e \tag{6.17}$$

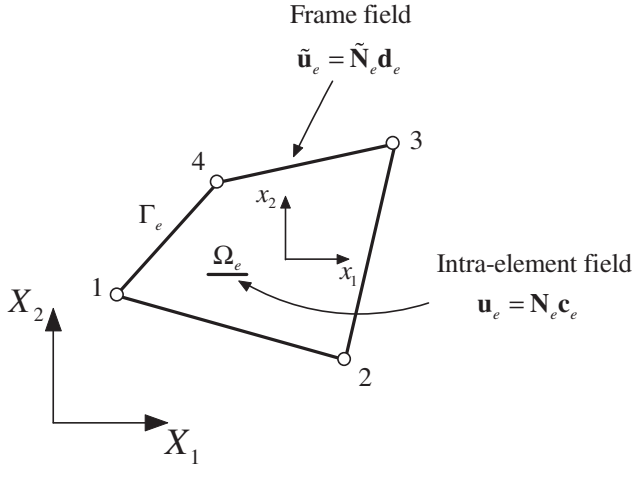


FIGURE 6.4
Intra-element field and frame field in a particular element e for plane elastic problems

6.3.2 Auxiliary conforming frame field

To enforce conformity on displacements along inter-element boundaries, i.e., $\mathbf{u}_e = \mathbf{u}_f$ on $\Gamma_e \cap \Gamma_f$ of any two neighbouring elements, we introduce an auxiliary inter-element frame field $\tilde{\mathbf{u}}_e$ approximated in terms of the same degrees of freedom (DOF),

$\mathbf{d}_e$ as used in conventional elements. The frame field $\tilde{\mathbf{u}}_e$, defined on the element boundary, can thus be written as

$$\tilde{\mathbf{u}}_e(\mathbf{x}) = \begin{bmatrix} \tilde{N}_{e1} & 0 & \tilde{N}_{e2} & 0 & \cdots & \tilde{N}_{en} & 0 \\ 0 & \tilde{N}_{e1} & 0 & \tilde{N}_{e2} & \cdots & 0 & \tilde{N}_{en} \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{21} \\ u_{12} \\ u_{22} \\ \vdots \\ u_{1n} \\ u_{2n} \end{bmatrix} = \tilde{\mathbf{N}}_e(\mathbf{x})\,\mathbf{d}_e \tag{6.18}$$

It is independently assumed along the element boundary in terms of nodal DOF $\mathbf{d}_e$, where $\tilde{\mathbf{N}}_e$ represents the conventional FE interpolating functions and u_{1i} and u_{2i} denote two displacement components at node i, respectively.

For example, a simple interpolation of the frame field on side 2-3 of a particular 4-node element (Figure 6.4) can be given in the form

$$\tilde{\mathbf{u}}_{23} = \begin{bmatrix} \hat{N}_1 & 0 & \hat{N}_2 & 0 \\ 0 & \hat{N}_1 & 0 & \hat{N}_2 \end{bmatrix} \begin{bmatrix} u_{12} \\ u_{22} \\ u_{13} \\ u_{23} \end{bmatrix} \tag{6.19}$$

where

$$\hat{N}_1 = \frac{1-\xi}{2}, \qquad \hat{N}_2 = \frac{1+\xi}{2} \tag{6.20}$$

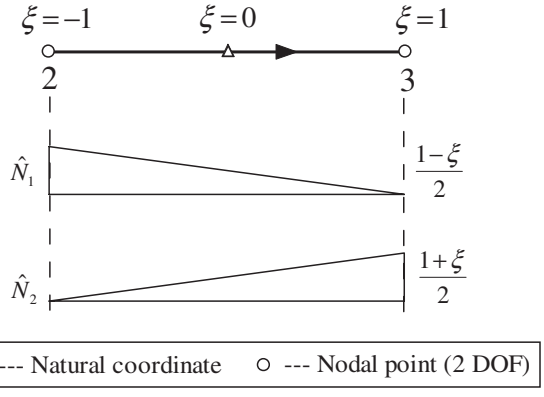


FIGURE 6.5
Typical linear interpolation for frame field on one edge

Since $\tilde{\mathbf{N}}_e$ is defined on the whole element boundary, expanding Eq. (6.19) to the

entire element boundary, we have

$$\tilde{\mathbf{u}}_{23} = \begin{bmatrix} 0 & 0 & \hat{N}_1 & 0 & \hat{N}_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \hat{N}_1 & 0 & \hat{N}_2 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{21} \\ u_{12} \\ u_{22} \\ u_{13} \\ u_{23} \\ u_{14} \\ u_{24} \end{bmatrix} = \tilde{\mathbf{N}}_e(\mathbf{x})\,\mathbf{d}_e \tag{6.21}$$

6.3.3 Modified variational functional

Assume that Ψ_e is the variational functional related to a particular element e, which has the following form for the case of a three-dimensional linear elastic problem [10]

$$\Psi_e = \frac{1}{2}\int_{\Omega_e} \varepsilon^{\mathrm{T}}\sigma \mathrm{d}\Omega - \int_{\Gamma_{eu}} \mathbf{t}^{\mathrm{T}}\bar{\mathbf{u}}\mathrm{d}\Gamma - \int_{\Gamma_{et}} \left(\mathbf{t}-\bar{\mathbf{t}}\right)^{\mathrm{T}}\tilde{\mathbf{u}}\mathrm{d}\Gamma - \int_{\Gamma_{eI}} \mathbf{t}^{\mathrm{T}}\tilde{\mathbf{u}}\mathrm{d}\Gamma \tag{6.22}$$

where Γ_{eI} is the inter-element boundary of the element e. $\Gamma_e = \Gamma_{eu} + \Gamma_{et} + \Gamma_{eI}$, $\Gamma_{eu} = \Gamma_e \cap \Gamma_u$, $\Gamma_{et} = \Gamma_e \cap \Gamma_t$.

Noting that $\Gamma_e = \Gamma_{eu} + \Gamma_{et} + \Gamma_{eI}$ and $\bar{\mathbf{u}} = \tilde{\mathbf{u}}$ on the boundary, Eq. (6.22) can be simplified as

$$\Psi_e = \frac{1}{2}\int_{\Omega_e} \varepsilon^{\mathrm{T}}\sigma \mathrm{d}\Omega - \int_{\Gamma_e} \mathbf{t}^{\mathrm{T}}\tilde{\mathbf{u}}\mathrm{d}\Gamma + \int_{\Gamma_{et}} \bar{\mathbf{t}}^{\mathrm{T}}\tilde{\mathbf{u}}\mathrm{d}\Gamma \tag{6.23}$$

Integrating the domain integral in Eq. (6.23) by parts gives

$$\begin{aligned} \int_{\Omega_e} \varepsilon^{\mathrm{T}}\sigma \mathrm{d}\Omega &= \int_{\Omega_e} u_{i,j}\sigma_{ij}\mathrm{d}\Omega = \int_{\Omega_e} \left(u_i\sigma_{ij}\right)_{,j}\mathrm{d}\Gamma - \int_{\Omega_e} u_i\sigma_{ij,j}\mathrm{d}\Omega \\ &= \int_{\Gamma_e} u_i\sigma_{ij}n_j\mathrm{d}\Gamma - \int_{\Omega_e} u_i\sigma_{ij,j}\mathrm{d}\Omega \\ &= \int_{\Gamma_e} u_i t_i \mathrm{d}\Gamma - \int_{\Omega_e} u_i b_i \mathrm{d}\Omega \\ &= \int_{\Gamma_e} \mathbf{t}^{\mathrm{T}}\mathbf{u}\mathrm{d}\Gamma - \int_{\Omega_e} \mathbf{b}^{\mathrm{T}}\mathbf{u}\mathrm{d}\Omega \end{aligned} \tag{6.24}$$

Making use of Eq. (6.24) and assuming there is no body force, the functional (6.23) can be further simplified as

$$\Psi_e = \frac{1}{2}\int_{\Gamma_e} \mathbf{t}^{\mathrm{T}}\mathbf{u}\mathrm{d}\Gamma - \int_{\Gamma_e} \mathbf{t}^{\mathrm{T}}\tilde{\mathbf{u}}\mathrm{d}\Gamma + \int_{\Gamma_{et}} \bar{\mathbf{t}}^{\mathrm{T}}\tilde{\mathbf{u}}\mathrm{d}\Gamma \tag{6.25}$$

For the plane stress/strain problems considered in this chapter, Eq. (6.25) becomes

$$\Psi_e = \frac{1}{2}\int_{\Gamma_e} \mathbf{t}^{\mathrm{T}}\mathbf{u}t_e\mathrm{d}\Gamma - \int_{\Gamma_e} \mathbf{t}^{\mathrm{T}}\tilde{\mathbf{u}}t_e\mathrm{d}\Gamma + \int_{\Gamma_{et}} \bar{\mathbf{t}}^{\mathrm{T}}\tilde{\mathbf{u}}t_e\mathrm{d}\Gamma \tag{6.26}$$

where t_e denotes the thickness of the element e.

Substituting the intra-element fields (6.11) and (6.16) and the frame field (6.18) into the functional (6.25) produces

$$\Psi_e = \frac{1}{2}\mathbf{c}_e^{\mathrm{T}}\mathbf{H}_e\mathbf{c}_e - \mathbf{c}_e^{\mathrm{T}}\mathbf{G}_e\mathbf{d}_e + \mathbf{d}_e^{\mathrm{T}}\mathbf{g}_e \tag{6.27}$$

in which

$$\mathbf{H}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathrm{T}}\mathbf{N}_e \mathrm{d}\Gamma \tag{6.28}$$

$$\mathbf{G}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathrm{T}}\tilde{\mathbf{N}}_e \mathrm{d}\Gamma \tag{6.29}$$

$$\mathbf{g}_e = \int_{\Gamma_{eq}} \tilde{\mathbf{N}}_e^{\mathrm{T}}\bar{\mathbf{t}} \mathrm{d}\Gamma \tag{6.30}$$

To enforce inter-element continuity of stress and displacement fields on the common element boundary, the unknown vector $\mathbf{c}_e$ should be expressed in terms of nodal DOF $\mathbf{d}_e$. Minimisation of the functional Ψ_e yields

$$\frac{\partial \Psi_e}{\partial \mathbf{c}_e^{\mathrm{T}}} = \mathbf{H}_e\mathbf{c}_e - \mathbf{G}_e\mathbf{d}_e = \mathbf{0} \tag{6.31}$$

$$\frac{\partial \Psi_e}{\partial \mathbf{d}_e^{\mathrm{T}}} = -\mathbf{G}_e^{\mathrm{T}}\mathbf{c}_e + \mathbf{g}_e = \mathbf{0} \tag{6.32}$$

Eq. (6.31) is used to determine the optional relationship between $\mathbf{c}_e$ and $\mathbf{d}_e$. It provides

$$\mathbf{c}_e = \mathbf{H}_e^{-1}\mathbf{G}_e\mathbf{d}_e \tag{6.33}$$

Substitution of Eq. (6.33) into Eq. (6.32) yields the element stiffness equation

$$\mathbf{K}_e\mathbf{d}_e = \mathbf{p}_e \tag{6.34}$$

where

$$\mathbf{K}_e = \mathbf{G}_e^{\mathrm{T}}\mathbf{H}_e^{-1}\mathbf{G}_e \tag{6.35}$$

and

$$\mathbf{p}_e = \mathbf{g}_e \tag{6.36}$$

stand for the element stiffness matrix and the equivalent nodal load vector, respectively.

The symmetric positive definition matrix $\mathbf{H}_e$ is the internal deformability matrix of the element and, in the absence of body forces, the matrix product [6]

$$\mathbf{c}_e^{\mathrm{T}}\mathbf{H}_e\mathbf{c}_e = \mathbf{d}_e^{\mathrm{T}}\mathbf{K}_e\mathbf{d}_e = 2U_e \tag{6.37}$$

represents double the strain energy U_e stored in the element. It is of interest to observe that the $\mathbf{H}_e$ matrix is hierarchic and as a consequence, the matrix of lower order is embedded in it corresponding to an increased number of homogeneous solutions. In addition, from Eq. (6.37) we notice that the symmetric property of matrix $\mathbf{H}_e$ is confirmed again.

6.3.4 Recovery of rigid-body motion

Once the nodal displacement $\mathbf{d}_e$ has been determined, the internal parameters $\mathbf{c}_e$ can be found from Eq. (6.33). The internal displacements calculated by Eq. (6.11) may, however, be in error by three rigid-body motion modes, since such terms have been discarded to prevent the element deformability matrix $\mathbf{H}_e$ from being singular. But now these missing terms can easily be recovered by setting for the augmented internal displacements

$$\mathbf{u}_e = \mathbf{N}_e \mathbf{c}_e + \begin{bmatrix} 1 & 0 & x_2 \\ 0 & 1 & -x_1 \end{bmatrix} \mathbf{c}_0 \tag{6.38}$$

where the undetermined rigid-body motion vector $\mathbf{c}_0$ can be calculated using the least square matching of $\mathbf{u}_e$ and $\tilde{\mathbf{u}}_e$ at n element nodes [10]

$$\sum_{i=1}^{n} \left[(u_{1i} - \tilde{u}_{1i})^2 + (u_{2i} - \tilde{u}_{2i})^2 \right] = \min \tag{6.39}$$

which yields

$$\mathbf{R}_e \mathbf{c}_0 = \mathbf{r}_e \tag{6.40}$$

with

$$\mathbf{R}_e = \sum_{i=1}^{n} \begin{bmatrix} 1 & 0 & x_{2i} \\ 0 & 1 & -x_{1i} \\ x_{2i} & -x_{1i} & x_{1i}^2 + x_{2i}^2 \end{bmatrix} \tag{6.41}$$

$$\mathbf{r}_e = \sum_{i=1}^{n} \begin{bmatrix} \Delta u_{e1i} \\ \Delta u_{e2i} \\ \Delta u_{e1i} x_{2i} - \Delta u_{e2i} x_{1i} \end{bmatrix} \tag{6.42}$$

and

$$\Delta u_{eji} = \tilde{u}_{eji} - \hat{u}_{eji} \qquad (j = 1, 2) \tag{6.43}$$

From Eq. (6.40), the rigid-body motion vector can be obtained

$$\mathbf{c}_0 = \mathbf{R}_e^{-1} \mathbf{r}_e \tag{6.44}$$

6.4 T-complete functions

As we have seen from the previous section, the assumed internal field (6.11) requires knowledge of the homogeneous solution of the governing equation (6.10) in the absence of body force $\mathbf{b}$. This is the so-called T-complete functions of elastostatics and can be generated in a systematic way from Muskhelishvili's complex variable formulation [11]. Using two analytic functions $\phi(z)$ and $\chi(z)$ we can express the displacements and stresses in the form [11]:

$$2G(u_1 + iu_2) = \kappa \phi(z) - z\overline{\phi'(z)} - \overline{\chi(z)} \tag{6.45}$$

$$\begin{cases} \sigma_{11}+\sigma_{22}=2\left[\phi'(z)+\overline{\phi'(z)}\right]=4Re\left[\phi'(z)\right] \\ \sigma_{22}-\sigma_{11}+2i\sigma_{12}=2\left[\bar{z}\phi''(z)+\chi'(z)\right] \end{cases} \tag{6.46}$$

which can be written in an equivalent form as

$$\begin{cases} \sigma_{11}-i\sigma_{12}=\phi'(z)+\overline{\phi'(z)}-\bar{z}\phi''(z)-\chi'(z) \\ \sigma_{22}+i\sigma_{12}=\phi'(z)+\overline{\phi'(z)}+\bar{z}\phi''(z)+\chi'(z) \end{cases} \tag{6.47}$$

where $z=x_1+ix_2$ with $i=\sqrt{-1}$. G stands for the shear modulus, $\kappa=(3-\nu)/(1+\nu)$ for plane stress and $\kappa=3-4\nu$ for plane strain, ν is Poisson's ratio. $\underline{(\,)}'$ denotes complex differentiation with respect to complex space variable z, and $\overline{(\,)}$ represents the complex conjugate obtained by replacing i with $-i$ in the complex variable. The stresses can be further written as

$$\sigma_{11}=\mathrm{Re}\left[2\phi'(z)-\bar{z}\phi''(z)-\chi'(z)\right] \tag{6.48}$$

$$\sigma_{22}=\mathrm{Re}\left[2\phi'(z)+\bar{z}\phi''(z)+\chi'(z)\right] \tag{6.49}$$

$$\sigma_{12}=\mathrm{Im}\left[\bar{z}\phi''(z)+\chi'(z)\right] \tag{6.50}$$

where Re[ ] and Im[ ] stand for the real and imaginary parts of a complex number, respectively.

Of particular interest is a complete set of polynomial solutions which may be generated by setting in Eq. (6.45) in turn [1, 6]

$$\left.\begin{array}{ll} \phi(z)=iz^k, & \chi(z)=0 \\ \phi(z)=z^k, & \chi(z)=0 \\ \phi(z)=0, & \chi(z)=iz^k \\ \phi(z)=0, & \chi(z)=z^k \end{array}\right\} \quad (k=1,2,3,\cdots) \tag{6.51}$$

This leads, for $\mathbf{N}_j$ of Eq. (6.13), to the following sequence*

$$2G\mathbf{N}^*_{1k}=\begin{Bmatrix}\mathrm{ReZ_{1k}}\\ \mathrm{ImZ_{1k}}\end{Bmatrix} \text{ with } Z_{1k}=i\kappa z^k+ikz\bar{z}^{k-1} \tag{6.52}$$

$$2G\mathbf{N}^*_{2k}=\begin{Bmatrix}\mathrm{ReZ_{2k}}\\ \mathrm{ImZ_{2k}}\end{Bmatrix} \text{ with } Z_{2k}=\kappa z^k-kz\bar{z}^{k-1} \tag{6.53}$$

$$2G\mathbf{N}^*_{3k}=\begin{Bmatrix}\mathrm{ReZ_{3k}}\\ \mathrm{ImZ_{3k}}\end{Bmatrix} \text{ with } Z_{3k}=i\bar{z}^k \tag{6.54}$$

$$2G\mathbf{N}^*_{4k}=\begin{Bmatrix}\mathrm{ReZ_{4k}}\\ \mathrm{ImZ_{4k}}\end{Bmatrix} \text{ with } Z_{4k}=-\bar{z}^k \tag{6.55}$$

*In the deduction below, the complex relations $\overline{iz^n}=-i\bar{z}^n$ and $\overline{z^n}=\bar{z}^n$ are frequently used.

Substituting Eq. (6.51) into Eqs. (6.48) - (6.50) yields the corresponding stress complete fields:

$$\sigma_e = \mathbf{T}_e \mathbf{c}_e = \sum_{j=1}^{m} \mathbf{T}_j \mathbf{c}_j \tag{6.56}$$

where

$$\mathbf{T}_{1k}^{*} = \left\{ \begin{array}{l} \mathrm{Re}\,(\mathrm{R_{1k}} - \mathrm{S_{1k}}) \\ \mathrm{Re}\,(\mathrm{R_{1k}} + \mathrm{S_{1k}}) \\ \mathrm{ImS_{1k}} \end{array} \right\} \text{ with } R_{1k} = 2ikz^{k-1},\ S_{1k} = ik\,(k-1)\,z^{k-2}\overline{z} \tag{6.57}$$

$$\mathbf{T}_{2k}^{*} = \left\{ \begin{array}{l} \mathrm{Re}\,(\mathrm{R_{2k}} - \mathrm{S_{2k}}) \\ \mathrm{Re}\,(\mathrm{R_{2k}} + \mathrm{S_{2k}}) \\ \mathrm{ImS_{2k}} \end{array} \right\} \text{ with } R_{2k} = 2kz^{k-1},\ S_{2k} = k\,(k-1)\,z^{k-2}\overline{z} \tag{6.58}$$

$$\mathbf{T}_{3k}^{*} = \left\{ \begin{array}{r} -\mathrm{ReS_{3k}} \\ \mathrm{ReS_{3k}} \\ \mathrm{ImS_{3k}} \end{array} \right\} \text{ with } S_{3k} = ikz^{k-1} \tag{6.59}$$

$$\mathbf{T}_{4k}^{*} = \left\{ \begin{array}{r} -\mathrm{ReS_{4k}} \\ \mathrm{ReS_{4k}} \\ \mathrm{ImS_{4k}} \end{array} \right\} \text{ with } S_{4k} = kz^{k-1} \tag{6.60}$$

It should be mentioned that in the case of $k = 1$, the terms S_{1k} and S_{2k} are equal to zero according to Eqs.(6.48) - (6.51).

As noted in the previous section, special care should be taken to discard from $\mathbf{u}_e$ all rigid-body motion terms and to form the vector $\mathbf{N}_e = [\mathbf{N}_1 \quad \mathbf{N}_2 \quad \cdots \quad \mathbf{N}_m]$ as a set of linearly independent functions $\mathbf{N}_j$ associated with non-vanishing strains. For $k = 0$, the expressions (6.52) - (6.55) yield constant translations along the coordinate axes. Furthermore, for $k = 1$ the relation (6.52) yields constant rotation $2\omega = (u_{1,2} - u_{2,1}) = -(1+\kappa)/G$ with $2Gu_1 = -(1+\kappa)x_2$ and $2Gu_2 = (1+\kappa)x_1$. As a result, starting with $k = 1$ for the relations (6.53) - (6.55) and $k > 1$ for the relations (6.52) - (6.55), the sequence generates independent displacement patterns of increasing polynomial degree.

The usage of the generating sequences (6.52) - (6.55) for the homogeneous displacement solutions $\mathbf{N}_j^{*}$ and Eqs. (6.57) - (6.60) calls for the following comments [6]:

- The hybrid techniques used in HT-FEM imply that the generated homogeneous solutions $\mathbf{N}_j^{*}$ yield non-vanishing stresses $\mathbf{T}_j^{*}$ (condition necessary to ensure that the internal deformability matrix $\mathbf{H}_e$ be non-singular) and, as a consequence, the three rigid-body motion modes should be excluded.

- A minimum of

$$m_{\min} = N_{dof} - 3 \tag{6.61}$$

of independent solutions $\mathbf{N}_j$, where N_{dof} is the number of nodal DOF of the element, is necessary (but sometimes not sufficient) for the resulting stiffness

matrix to have full rank. Whenever necessary, full rank can always be achieved by using a suitable higher number of functions, $m > m_{\min}$, than given by the suitable condition (6.61).

- To preserve the desirable geometrical invariance of the element under rotation of the coordinate axes, a simple "truncation rule" must be observed whereby the generating sequence may be stopped after either relation (6.58) or (6.60). Since $k = 1$ generates only three independent functions, this rule implies that m should always be an odd number.

In practical computation, for example, seven Trefftz complete functions can be chosen with three independent patterns for $k = 1$ and four independent patterns for $k = 2$ for the case of a 4-nodal element ($m_{\min} = N_{dof} - 3 = 2 \times 4 - 3 = 5$), that is,

$$\mathbf{N}_1 = \mathbf{N}^*_{2k}|_{k=1}, \qquad \mathbf{T}_1 = \mathbf{T}^*_{2k}|_{k=1} \tag{6.62}$$

$$\mathbf{N}_2 = \mathbf{N}^*_{3k}|_{k=1}, \qquad \mathbf{T}_2 = \mathbf{T}^*_{3k}|_{k=1} \tag{6.63}$$

$$\mathbf{N}_3 = \mathbf{N}^*_{4k}|_{k=1}, \qquad \mathbf{T}_3 = \mathbf{T}^*_{4k}|_{k=1} \tag{6.64}$$

$$\mathbf{N}_4 = \mathbf{N}^*_{1k}|_{k=2}, \qquad \mathbf{T}_4 = \mathbf{T}^*_{1k}|_{k=2} \tag{6.65}$$

$$\mathbf{N}_5 = \mathbf{N}^*_{2k}|_{k=2}, \qquad \mathbf{T}_5 = \mathbf{T}^*_{2k}|_{k=2} \tag{6.66}$$

$$\mathbf{N}_6 = \mathbf{N}^*_{3k}|_{k=2}, \qquad \mathbf{T}_6 = \mathbf{T}^*_{3k}|_{k=2} \tag{6.67}$$

$$\mathbf{N}_7 = \mathbf{N}^*_{4k}|_{k=2}, \qquad \mathbf{T}_7 = \mathbf{T}^*_{4k}|_{k=2} \tag{6.68}$$

6.5 Computation of H and G matrix

The computation of matrices $\mathbf{H}_e$ and $\mathbf{G}_e$ defined in Eqs. (6.28) and (6.29) is performed in a manner similar to that described in the previous chapter. In the following, we again assume that there are n_e edges and n nodes of each element and on each edge there are n_o nodes.

6.5.1 Geometric characteristics of boundary edges

Using the expression (4.2), the derivatives of space variables with respect to ξ can be written as

$$\begin{Bmatrix} \dfrac{\mathrm{d}x_1}{\mathrm{d}\xi} \\ \dfrac{\mathrm{d}x_2}{\mathrm{d}\xi} \end{Bmatrix} = \sum_{i=1}^{n_o} \frac{\mathrm{d}N_i(\xi)}{\mathrm{d}\xi} \begin{Bmatrix} x_{1i} \\ x_{2i} \end{Bmatrix} \tag{6.69}$$

The normal cosine at any point on the edge is determined by

$$n_1 = \frac{\mathrm{d}x_2}{\mathrm{d}\Gamma} = \frac{\mathrm{d}x_2}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}\Gamma} = \left[\sum_{i=1}^{n_o}\frac{\mathrm{d}N_i(\xi)}{\mathrm{d}\xi}x_{2i}\right]\frac{1}{J} \tag{6.70}$$

$$n_2 = -\frac{\mathrm{d}x_1}{\mathrm{d}\Gamma} = -\frac{\mathrm{d}x_1}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}\Gamma} = -\left[\sum_{i=1}^{n_o}\frac{\mathrm{d}N_i(\xi)}{\mathrm{d}\xi}x_{1i}\right]\frac{1}{J} \tag{6.71}$$

where

$$\mathrm{d}\Gamma = \sqrt{(\mathrm{d}x_1)^2 + (\mathrm{d}x_2)^2} = \sqrt{\left(\frac{\mathrm{d}x_1}{\mathrm{d}\xi}\right)^2 + \left(\frac{\mathrm{d}x_2}{\mathrm{d}\xi}\right)^2}\mathrm{d}\xi = J\mathrm{d}\xi \tag{6.72}$$

and

$$J(\xi) = \sqrt{\left(\frac{\mathrm{d}x_1}{\mathrm{d}\xi}\right)^2 + \left(\frac{\mathrm{d}x_2}{\mathrm{d}\xi}\right)^2} \tag{6.73}$$

6.5.2 Computation of matrix H

Substituting Eq. (6.17) into Eq. (6.28), we have

$$\mathbf{H}_e = \int_{\Gamma_e}\mathbf{Q}_e^{\mathrm{T}}\mathbf{N}_e t_e \mathrm{d}\Gamma = \int_{\Gamma_e}\mathbf{T}_e^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{N}_e t_e \mathrm{d}\Gamma \tag{6.74}$$

where

$$\mathbf{A} = \begin{bmatrix} n_1 & 0 & n_2 \\ 0 & n_2 & n_1 \end{bmatrix}$$

represents the matrix of directional cosine of outward normal of the edge and is generally related to the coordinates of points on the edge. $\mathbf{T}_e$ and $\mathbf{N}_e$ defined in Eqs. (6.12) and (6.15) are dependent on the spatial coordinates.

As was done in Section 5.5.2, we introduce again the matrix function:

$$\mathbf{F}(\mathbf{x}) = [F_{ij}(\mathbf{x})]_{m\times m} = \mathbf{T}_e^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{N}_e t_e \tag{6.75}$$

Substitution of Eq. (6.75) into Eq. (6.74) yields

$$H_{ij} = \int_{\Gamma_e}F_{ij}(\mathbf{x})\,\mathrm{d}\Gamma = \sum_{l=1}^{n_e}\int_{\Gamma_{el}}F_{ij}(\mathbf{x})\,\mathrm{d}\Gamma \tag{6.76}$$

Using Eq. (6.72) and Gaussian numerical integration as described in Chapter 4, we finally obtain

$$H_{ij} = \sum_{l=1}^{n_e}\left[\int_{-1}^{+1}F_{ij}(\mathbf{x}(\xi))J(\xi)\,\mathrm{d}\xi\right] \approx \sum_{l=1}^{n_e}\left[\sum_{k=1}^{n_s}w_k F_{ij}(\mathbf{x}(\xi_k))J(\xi_k)\right] \tag{6.77}$$

where n_s is the number of Gaussian sampling points employed in the Gaussian numerical integration.

6.5.3 Computation of matrix G

Making use of Eq. (6.17), Eq. (6.29) can be rewritten as

$$\mathbf{G}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathbf{T}} \tilde{\mathbf{N}}_e t_e \mathrm{d}\Gamma = \int_{\Gamma_e} \mathbf{T}_e^{\mathrm{T}} \mathbf{A}^{\mathrm{T}} \tilde{\mathbf{N}}_e t_e \mathrm{d}\Gamma \tag{6.78}$$

where $\tilde{\mathbf{N}}_e$ is defined in Eq. (6.18).

As was done for matrix $\mathbf{H}$, we introduce a new matrix:

$$\tilde{\mathbf{F}}(\mathbf{x}) = \left[\tilde{F}_{ij}(\mathbf{x})\right]_{m \times n_b} = \mathbf{T}_e^{\mathrm{T}} \mathbf{A}^{\mathrm{T}} \tilde{\mathbf{N}}_e t_e \tag{6.79}$$

where $n_b = n \times 2$ denotes the total number of DOF at nodes in the element. The matrix $\mathbf{G}_e$ can thus be expressed as

$$G_{ij} = \int_{\Gamma_e} \tilde{F}_{ij}(\mathbf{x}) \mathrm{d}\Gamma = \sum_{l=1}^{n_e} \int_{\Gamma_{el}} \tilde{F}_{ij}(\mathbf{x}) \mathrm{d}\Gamma \tag{6.80}$$

Again, using Eq. (6.72) and Gaussian numerical integration formulation we have

$$G_{ij} = \sum_{l=1}^{n_e} \left[\int_{-1}^{1} F_{ij}(\mathbf{x}(\xi)) J(\xi) \mathrm{d}\xi \right] \approx \sum_{l=1}^{n_e} \left[\sum_{k=1}^{n_s} w_k F_{ij}(\mathbf{x}(\xi_k)) J(\xi_k) \right] \tag{6.81}$$

6.6 Evaluation of equivalent nodal loads

Consider an element edge which is assumed to be subject to a distributed loading per unit length in the normal and tangential directions as shown in Figure 6.6. We define that a pressure normal to a face is assumed to be positive if it acts in a direction pointing towards the element, and a tangential load is assumed to be positive if it acts in an anticlockwise direction with respect to the loaded element [12].

For a plane stress/strain problem the equivalent nodal load vector can be written as

$$\mathbf{p}_e = \int_{\Gamma_{eq}} \tilde{\mathbf{N}}_e^{\mathrm{T}} \bar{\mathbf{t}} \mathrm{d}\Gamma \tag{6.82}$$

where Γ_{eq} is the element boundary on which $\bar{\mathbf{t}} = \{\bar{t}_1 \quad \bar{t}_2\}^{\mathrm{T}}$ is applied. $\bar{\mathbf{t}}$ is related to $\{p_n \quad p_t\}^{\mathrm{T}}$ (see Figure 6.6) by

$$\begin{Bmatrix} t_1 \\ t_2 \end{Bmatrix} = \begin{bmatrix} -\sin\alpha & \cos\alpha \\ \cos\alpha & \sin\alpha \end{bmatrix} \begin{Bmatrix} p_n \\ p_t \end{Bmatrix} = \begin{bmatrix} -n_1 & -n_2 \\ -n_2 & n_1 \end{bmatrix} \begin{Bmatrix} p_n \\ p_t \end{Bmatrix} \tag{6.83}$$

where n_1 and n_2 are unit outward normal components, which are expressed in Eqs. (6.70) and (6.71), and have a relation with the angle α displayed in Figure 6.6

$$n_1 = \frac{\mathrm{d}x_2}{\mathrm{d}\Gamma} = \sin\alpha, \qquad n_2 = -\frac{\mathrm{d}x_1}{\mathrm{d}\Gamma} = -\cos\alpha \tag{6.84}$$

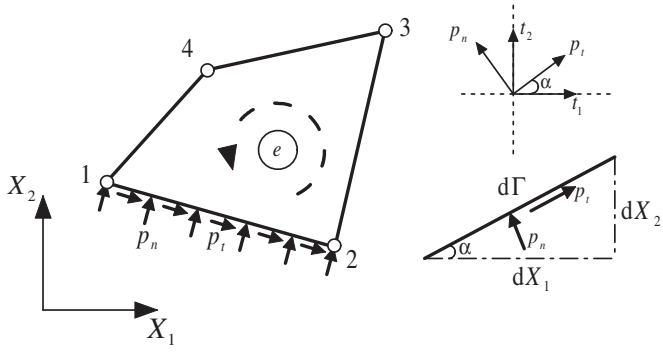


FIGURE 6.6
Illustration of normal and tangential traction on a loaded edge

Although the matrix of shape function $\tilde{\mathbf{N}}_e$ is defined on the whole element boundary, the nonzero equivalent nodal loads exist on the loaded edge only, so we can rewrite Eq. (6.82) in component form as

$$\mathbf{p}_e = \begin{Bmatrix} p_{1i} \\ p_{2i} \end{Bmatrix} = \int_{\Gamma_e} \begin{Bmatrix} \hat{N}_i(\xi)\, t_1 \\ \hat{N}_i(\xi)\, t_2 \end{Bmatrix} \mathrm{d}\Gamma \tag{6.85}$$

in which $\hat{N}_i(\xi)$ is the shape function defined at the ith node on the loaded edge, and p_{1i} and p_{2i} are the corresponding components of equivalent nodal loads, respectively.

Substituting Eq. (6.83) into Eq. (6.85) yields the following boundary integral:

$$\begin{Bmatrix} p_{1i} \\ p_{2i} \end{Bmatrix} = \int_{\Gamma_e} \begin{Bmatrix} \hat{N}_i(\xi)(-n_1 p_n - n_2 p_t) \\ \hat{N}_i(\xi)(-n_2 p_n + n_1 p_t) \end{Bmatrix} \mathrm{d}\Gamma \tag{6.86}$$

Considering the transformation (6.72) of infinitesimal length dΓ represented by local coordinate variable ξ, we have

$$\begin{Bmatrix} p_{1i} \\ p_{2i} \end{Bmatrix} = \int_{-1}^{+1} \begin{Bmatrix} N_i(\xi)(-n_1 p_n - n_2 p_t) J(\xi) \\ N_i(\xi)(-n_2 p_n + n_1 p_t) J(\xi) \end{Bmatrix} \mathrm{d}\xi \tag{6.87}$$

from which approximated values of integration can be obtained by means of one-dimensional Gaussian numerical integration

$$\begin{Bmatrix} p_{1i} \\ p_{2i} \end{Bmatrix} \approx \sum_{k=1}^{n_s} \begin{Bmatrix} w_k N_i(\xi_k)\,(-n_1 p_n - n_2 p_t)|_{\xi_k} J(\xi_k) \\ w_k N_i(\xi_k)\,(-n_2 p_n + n_1 p_t)|_{\xi_k} J(\xi_k) \end{Bmatrix} \tag{6.88}$$

where n_s is the number of Gaussian sampling points. The load parameters p_t and p_n at Gaussian sampling points ξ_k can be evaluated in the following way. Assume

that there are n_o nodes on the loaded edge. Then, the distributions of normal and tangential loads along the loaded edge are expressed as

$$\begin{Bmatrix} p_n \\ p_t \end{Bmatrix} = \sum_{i=1}^{n_o} \hat{N}_i(\xi) \begin{Bmatrix} p_{ni} \\ p_{ti} \end{Bmatrix} \tag{6.89}$$

At the point ξ_k, Eq. (6.89) becomes

$$\begin{Bmatrix} p_n(\xi_k) \\ p_t(\xi_k) \end{Bmatrix} = \sum_{i=1}^{n_o} \hat{N}_i(\xi_k) \begin{Bmatrix} p_{ni} \\ p_{ti} \end{Bmatrix} \tag{6.90}$$

It is worth pointing out here that the above procedure to evaluate equivalent nodal loads can also be used to deal with elements of other shapes, in addition to the 4-node element considered.

6.7 MATLAB functions for plane elastic problems

Due to the application of modularization programming, the basic structure for solving plane elasticity using HT-FEM employed in the book is the same as the one considered in Chapter 5, so, such subroutines as `TYPELEM`, `ELEPARS`, `HMATRIX`, `GMATRIX`, `KMATRIX`, `FIEDNOD`, `CMATRIX` and `EDISNOD` are the same as the ones listed in Chapter 5, besides those common routines discussed in Chapter 4. For simplicity, we ignore those routines which have been described in Chapter 5 and just list the different ones.

```
function MAINFUN
% Main program using HTFEM for plane elasticity
% ******************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;

disp('*********************************************');
disp('              Hybrid Trefftz FEM');
disp('         for plane linear elastic problems');
disp('              without body forces');
disp('*********************************************');

% Input data from file
[NPOIN,NELEM,COORD,MATNO,LNODS,NVFIX,NOFIX,IFPRE,...
    PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,NOPRS,PRESS,...
    PROPS]=INPUTDT;
% Generate local relations of nodes and edges
```

```
[ELNOD]=TYPELEM;
% Element loop for stiffness matrix
NEQNS=NPOIN*NDOFN;
GSTIF=zeros(NEQNS,NEQNS);
GLOAD=zeros(NEQNS,1);
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute element stiffnesss matrix
    [ESTIF]=KMATRIX(EHMTX,EGMTX);
    % Assemble stiffness matrix
    [GSTIF]=ASMSTIF(iELEM,NNODE,LNODS,ESTIF,GSTIF);
end
% Compute equivalent loads
[GLOAD]=PVECTOR(LNODS,COORD,NDLEG,NEASS,NOPRS,...
    PRESS,NPLOD,LODPT,POINT,GLOAD);
% Introduce constrained displacements and point loads
[GSTIF,GLOAD]=INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,...
    GSTIF,GLOAD);
% Solve linear system of equations and store
% displacements of each node in the array ASDIS
[ASDIS]=LSSOLVR(GSTIF,GLOAD,NEQNS);
% Output nodal potential
[UPOIN]=FIEDNOD(NPOIN,ASDIS);
% Compute potential and flux components at central
% point of element
[CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,LNODS,COORD,...
    PROPS,ELNOD,ASDIS);
% Output results
OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,...
    NVFIX,NPLOD,NDLEG);
disp('--- All procedures are finished ---');


-------------------------------------------------------
function [GP]=PVECTOR(LNODS,COORD,NDLEG,NEASS,NOPRS,...
    PRESS,NPLOD,LODPT,POINT,GP)
% Compute effective nodal forces
% Input parameters:
%   MATNO: Material index of each element
```

```
%   PROPS: Properties of materials
%   LNODS: Element connections
%   COORD: Coordinates of nodes
%   NPLOD: Number of concentrated loads
%   LODPT: Global index of nodes at which concentrated
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NDLEG: Number of loaded edges
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   GP:    Global effective nodal loads
% Output parameters:
%   GP:    Global effective nodal loads
% ***************************************
global NDIME NDOFN NNODE NEDGE NODEG NGAUS;

% Add the point loads to global load vector
if NPLOD>0
    for iPLOD=1:NPLOD
        NRT=(LODPT(iPLOD)-1)*NDOFN;
        for iDOFN=1:NDOFN
            NR=NRT+iDOFN;
            GP(NR,1)=GP(NR,1)+POINT(iPLOD,iDOFN);
        end
    end
end

% Total number of local DOF of each element
NEVAB=NNODE*NDOFN;
% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);

% Calculate equivalent nodal loads on the distributed
% loaded edge
for iDLEG=1:NDLEG
    kELEM=NEASS(iDLEG);
    % Determine coordinates of nodes on the element
    % edge
    ELCOD=zeros(NODEG,NDIME);
    for iODEG=1:NODEG
        kPOIN=NOPRS(iDLEG,iODEG);
        for iDIME=1:NDIME
            ELCOD(iODEG,iDIME)=COORD(kPOIN,iDIME);
```

```
    end
    %ELCOD(iODEG,:)=COORD(kPOIN,:)
end
% Determine local nodal load intensity
EPRES=zeros(NODEG,NDOFN);
for iODEG=1:NODEG
    for iDOFN=1:NDOFN
        ii=(iODEG-1)*NDOFN+iDOFN;
        EPRES(iODEG,iDOFN)=PRESS(iDLEG,ii);
    end
end
% Integration along loaded edge
%    P(iODEG)(iDOFN)=integral(N(iODEG)*p(iDOFN)*dS)
% dS is arc-length
PE=zeros(NEVAB,1);
for iGAUS=1:NGAUS
    EXISP=POSGP(iGAUS);
    % Shape functions and its derivatives for 1D
    % line element
    SHAPE=zeros(1,NODEG);
    DSHAP=zeros(1,NODEG);
    [SHAPE,DSHAP]=SHAPFUN(EXISP);

    % Coordinates and derivatives of Gaussian
    % points
    %  x=sum(Ni*xi), y=sum(Ni*yi)
    %  dx/dt=sum(dNi/dt*xi)
    %  dy/dt=sum(dNi/dt*yi)
    %  DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    CORGS=zeros(1,NDIME);
    DERGS=zeros(1,NDIME);
    for iDIME=1:NDIME
        for iODEG=1:NODEG
            CORGS(iDIME)=CORGS(iDIME)+...
                SHAPE(iODEG)*ELCOD(iODEG,iDIME);
            DERGS(iDIME)=DERGS(iDIME)+...
                DSHAP(iODEG)*ELCOD(iODEG,iDIME);
        end
    end
    DVOLU=sqrt(DERGS(1)^2+DERGS(2)^2);
    % Directional cosine at Gaussian point
    %  nx = dy/dS,   ny = - dx/dS
    DS1= DERGS(2)/DVOLU;
    DS2=-DERGS(1)/DVOLU;
```

```
% Gauss integration factor
DVOLU=DVOLU*WEIGP(iGAUS);

% Load intensity at Gaussian point
%          p=sum(Ni*pi)
% for plane elastic problems, that is, NDOFN=2,
%   PGASH(1) is normal pressure, which is
%   assumed to be positive if it acts in a
%   direction into the element
%   PGASH(2) is tangential load, which is
%   assumed to be positive if it acts in an
%   anticlockwise direction with respect to
%   the loaded element
PGASH=zeros(NDOFN,1);
for iDOFN=1:NDOFN % pn, pt
    for iODEG=1:NODEG
        PGASH(iDOFN)=PGASH(iDOFN)+...
            SHAPE(iODEG)*EPRES(iODEG,iDOFN);
    end
end
%px=-pn*nx-pt*ny
%py=-pn*ny+pt*nx
PX=-DS1*PGASH(1)-DS2*PGASH(2);
PY=-DS2*PGASH(1)+DS1*PGASH(2);
PGASH(1)=PX;
PGASH(2)=PY;
% PGASH now is reset value
% Compute equivalent nodal forces PE
for iNODE=1:NNODE
    kPOIN=LNODS(kELEM,iNODE);
    if kPOIN==NOPRS(iDLEG,1)
        % iNODE is start point of loaded edge
        for iODEG=1:NODEG
            kNODE=iNODE+iODEG-1;
            if kNODE>NNODE
                kNODE=1;
            end
            for iDOFN=1:NDOFN
                iEVAB=(kNODE-1)*NDOFN+iDOFN;
                PE(iEVAB,1)=PE(iEVAB,1)+...
                    SHAPE(iODEG)*...
                    PGASH(iDOFN)*DVOLU;
            end
        end
    end
```

```
        end
    end
    % Assemble PE into global forces vector GP
    for iNODE=1:NNODE
        kPOIN=LNODS(kELEM,iNODE);
        for iDOFN=1:NDOFN
            kEQNS=NDOFN*(kPOIN-1)+iDOFN; % global DOF
            iEVAB=NDOFN*(iNODE-1)+iDOFN; % local DOF
            GP(kEQNS,1)=GP(kEQNS,1)+PE(iEVAB,1);
        end
    end
end
clear ELCOD EPRES PE SHAPE DSHAP PGASH POSGP WEIGP;


------------------------------------------------------
function [CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    LNODS,COORD,PROPS,ELNOD,ASDIS)
% Compute displacements and stresses at central point
% of each element
% Input parameters:
%   NELEM: Number of elements in domain
%   MATNO: Material index of each element
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
%   PROPS: Properties of materials
%   ELNOD: Local relation of edge and nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
% Output parameters:
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
% **********************************
global NDIME NDOFN NNODE NEDGE NODEG NSTRE NGAUS;

CECOD=zeros(NELEM,NDIME);
UCENP=zeros(NELEM,NDOFN);
SCENP=zeros(NELEM,NSTRE);
% Element loop for internal fields at central point
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Identify nodal field of the specified element
```

```
    [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Calculate the ce coefficients: m by 1
    [c_Ele]=CMATRIX(EHMTX,EGMTX,d_Ele);
    % Recover rigid-body motion vector: 3 by 1
    [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,PROPS);
    % Compute Trefftz internal fields at central point
    xp=0;
    yp=0;
    [N_SET,T_SET]=TREFFTZ(xp,yp,kMATS,PROPS);
    GDISP=N_SET*c_Ele;
    GSTRE=T_SET*c_Ele;
    UCENP(iELEM,1)=GDISP(1)+c0(1)+yp*c0(3);
    UCENP(iELEM,2)=GDISP(2)+c0(2)-xp*c0(3);
    SCENP(iELEM,1)=GSTRE(1);
    SCENP(iELEM,2)=GSTRE(2);
    SCENP(iELEM,3)=GSTRE(3);
    % Coordinates of computing points
    CECOD(iELEM,:)=[CenCoord(1,1)+xp,CenCoord(1,2)+yp];
end
clear EHMTX EGMTX c_Ele d_Ele;


------------------------------------------------------
function [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
    EXISP,ECOOD,ELNOD)
% Evaluate quantities at Gaussian points
% Input parameters:
%   iEDGE: Number index of element edge
%   EXISP: Local coordinate of Gaussian points
%   ECOOD: Coordinates of element nodes
%   ELNOD: Local relations of edge and nodes
% Output parameters:
%   CORGS: Coordinates of Gauss point
%   DVOLU: Functions of coordinate tranformation for
%          Gauss integral
%   AMTRX: Directional cosine at Gauss point
%   SHMTX: Shape function of frame field defined on
%          entire boundary
% ************************************
global NDIME NDOFN NNODE NEDGE NODEG;
```

```
% Shape functions and its derivatives for 1D line
% element
SHAPE=zeros(1,NODEG);
DSHAP=zeros(1,NODEG);
[SHAPE,DSHAP]=SHAPFUN(EXISP);

% Coordinates, derivatives and arc-length at Gaussian
% points
%  x=sum(Ni*xi) and y=sum(Ni*yi)
%  dx/dt=sum(dNi/dt*xi) and dy/dt=sum(dNi/dt*yi)
%  DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
CORGS=zeros(1,NDIME);
DERGS=zeros(1,NDIME);
for iDIME=1:NDIME
    for iODEG=1:NODEG
        kNODE=ELNOD(iEDGE,iODEG);
        CORGS(iDIME)=CORGS(iDIME)+...
            SHAPE(iODEG)*ECOOD(kNODE,iDIME);
        DERGS(iDIME)=DERGS(iDIME)+...
            DSHAP(iODEG)*ECOOD(kNODE,iDIME);
    end
end
DVOLU=sqrt(DERGS(1)^2+DERGS(2)^2);

% Directional cosine at Gauss point
%  nx = dy/dS      ny = - dx/dS
%  A=[n1,  0, n2
%      0, n2, n1]
AMTRX=zeros(2,3);
n1= DERGS(2)/DVOLU;
n2=-DERGS(1)/DVOLU;
AMTRX(1,1)=n1;
AMTRX(1,3)=n2;
AMTRX(2,2)=n2;
AMTRX(2,3)=n1;
%AMTRX=[n1,0,n2;0,n2,n1];

% Form shape function matrix of frame function
NEVAB=NNODE*NDOFN;
SHMTX=zeros(2,NEVAB);
iNODE=ELNOD(iEDGE,1); % start node of the edge
for iODEG=1:NODEG
    kNODE=iNODE+iODEG-1;
    if kNODE>NNODE
        kNODE=1;
```

```
    end
    SHMTX(1,kNODE*2-1)=SHAPE(iODEG);
    SHMTX(2,kNODE*2)  =SHAPE(iODEG);
end


------------------------------------------------------
function [N_SET,T_SET]=TREFFTZ(sp,tp,kMATS,PROPS)
% Compute the Trefftz functions with specified terms
% Input parameters:
%   sp,tp: coordinates
%   kMATS:  Material number
%   PROPS:  Material properties
% Output parameters:
%   N_SET: Displacements Trefftz functions
%   T_SET: Stresses Trefftz functions
% ***********************************************
global NTREF NTYPE NNODE NDOFN;

% Check the number of terms of Trefftz functions
temp=NNODE*NDOFN-3;
if NTREF<temp
    error('Too small terms of Trefftz functions!');
end

YOUNG=PROPS(kMATS,1);
POISS=PROPS(kMATS,2);
if NTYPE==1 % plane stress
    KAMA=(3.0-POISS)/(1.0+POISS);
elseif NTYPE==2 % plane strain
    KAMA=3.0-4*POISS;
end
GMOD=YOUNG/(2*(1+POISS));

Z=complex(sp,tp);
ZC=conj(Z);
UNIT=complex(0,1);
% Trefftz functions
ZK=zeros(1,NTREF);
RK=zeros(1,NTREF);
SK=zeros(1,NTREF);
NK=zeros(1,NTREF);

k=1;
ZK(1)=KAMA*Z-Z; % KAMA*Z-Z
```

```
ZK(2)=UNIT*ZC;  % I*ZC
ZK(3)=-ZC;      % -ZC

RK(1)=2;        % 2
RK(2)=0;
RK(3)=0;

SK(1)=0;        % 0
SK(2)=UNIT;     % I
SK(3)=1;        % 1

NK(1:3)=k;
nt=4;
while nt<NTREF
    k=k+1;

    ZK(nt)=KAMA*UNIT*Z^k+k*UNIT*Z*ZC^(k-1);
    RK(nt)=2*UNIT*k*Z^(k-1);
    SK(nt)=UNIT*k*(k-1)*Z^(k-2)*ZC;
    NK(nt)=k;

    nt=nt+1;
    ZK(nt)=KAMA*Z^k-k*Z*ZC^(k-1);
    RK(nt)=2*k*Z^(k-1);
    SK(nt)=k*(k-1)*Z^(k-2)*ZC;
    NK(nt)=k;

    nt=nt+1;
    if nt>=NTREF
        break;
    end
    ZK(nt)=UNIT*ZC^k;
    RK(nt)=0;
    SK(nt)=UNIT*k*Z^(k-1);
    NK(nt)=k;

    nt=nt+1;
    ZK(nt)=-ZC^k;
    RK(nt)=0;
    SK(nt)=k*Z^(k-1);
    NK(nt)=k;

    nt=nt+1;
end
```

```
% Compute Trefftz functions N_SET and T_SET
N_SET=zeros(2,NTREF);
T_SET=zeros(3,NTREF);
for im=1:NTREF
    N_SET(1,im)=real(ZK(im))/(2*GMOD);
    N_SET(2,im)=imag(ZK(im))/(2*GMOD);
    k=NK(im);
    n1=4*(k-1)-3;
    n2=4*(k-1)-2;
    km=im-3;
    if (im==1)|(km==n1)|(km==n2)
        T_SET(1,im)=real(RK(im))-real(SK(im));
        T_SET(2,im)=real(RK(im))+real(SK(im));
        T_SET(3,im)=imag(SK(im));
    else
        T_SET(1,im)=-real(SK(im));
        T_SET(2,im)= real(SK(im));
        T_SET(3,im)= imag(SK(im));
    end
end


------------------------------------------------------
function [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,PROPS)
% Recovery of rigid body motion
% Input parameters:
%   ECOOD: Coordinates of element nodes
%   c_Ele: Coefficients of Trefftz interpolation
%   d_Ele: Displacement field at element nodes
%   kMATS:  Material number
%   PROPS:  Material properties
% Output parameters:
%   c0: Rigid body motion term
% *********************************
global NNODE;

RMATX=zeros(3,3);
rvect=zeros(3,1);
for iNODE=1: NNODE
    x1=ECOOD(iNODE, 1);
    x2=ECOOD(iNODE, 2);
    [N_SET, T_SET]=TREFFTZ(x1,x2,kMATS,PROPS);
    u=N_SET*c_Ele;

    du1=d_Ele(iNODE*2-1)-u(1);
```

```
    du2=d_Ele(iNODE*2  )-u(2);
    rvect(1)=rvect(1)+du1;
    rvect(2)=rvect(2)+du2;
    rvect(3)=rvect(3)+x2*du1-x1*du2;

    RMATX(1,3)=RMATX(1,3)+x2;
    RMATX(2,3)=RMATX(2,3)-x1;
    RMATX(3,3)=RMATX(3,3)+(x1^2+x2^2);
end
RMATX(3,1)=RMATX(1,3);
RMATX(3,2)=RMATX(2,3);
RMATX(1,1)=NNODE;
RMATX(2,2)=NNODE;

c0=RMATX\rvect;
```

6.8 C computer programming

```
/*
*********************************************************
* Mainfunction MAINFUN                                  *
* - Call other subroutines                              *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int NTREF,NTYPE,NNODE,NEDGE,NODEG,NDIME,NDOFN,NSTRE,
NMATS,NPROP,NGAUS;
void main()
{
    void DUCLEAN(); void ITCLEAN(); void INPUTDT();
    void TYPELEM(); void ELEPARS();
    void HMATRIX(); void GMATRIX(); void KMATRIX();
    void ASMSTIF(); void PVECTOR(); void INDISBC();
    void LSSOLVR(); void FIEDNOD(); void FIEDCEN();
    void OPRESUT();
    FILE *fp;
    int NEQNS,NPOIN,NELEM,NVFIX,NPLOD,NDLEG,TNFEG,NEVAB;
    int *MATNO,*LNODS,*NOFIX,*IFPRE,*LODPT,*NEASS,*NOPRS,
        *ELNOD;
```

```
double *COORD,*PRESC,*POINT,*PRESS,*PROPS;
double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*ESTIF,*GSTIF,
    *GLOAD,*UPOIN,*CECOD,*UCENP,*SCENP;
char dummy[201],TITLE[201],file[81];
int i,j,k,N,n1,n2,iELEM,kMATS;

printf("**********************************\n");
printf("        Hybrid Trefftz FEM\n");
printf("   for plane linear elastic problems\n");
printf("        without body forces\n");
printf("**********************************\n");
/** Input data from file **/
puts("Input file name < dir:fn.txt >: ");
gets(file);
if((fp=fopen(file,"r"))==NULL)
{
    printf("Warning! Can't open input file\n");
    exit(0);
}
// basic parameters
fgets(dummy,200,fp);
fgets(TITLE,200,fp);
fgets(dummy,200,fp);

fgets(dummy,200,fp);
fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
    &NPLOD,&NDLEG);

// element connectivity
MATNO=(int *)calloc(NELEM,sizeof(int));
ITCLEAN(NELEM,1,MATNO);
LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
ITCLEAN(NELEM,NNODE,LNODS);
```

```
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NELEM;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    MATNO[i]=n1-1;
    for(j=0;j<NNODE;j++)
    {
        fscanf(fp,"%d",&n2);
        LNODS[i*NNODE+j]=n2-1;
    }
    fscanf(fp,"\n");
}
// nodal coordinates
COORD=(double *)calloc(NPOIN*NDIME,sizeof(double));
DUCLEAN(NPOIN,NDIME,COORD);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NPOIN;i++)
{
    fscanf(fp,"%d",&N);
    for(j=0;j<NDIME;j++)
    {
        fscanf(fp,"%lf",&COORD[i*NDIME+j]);
    }
    fscanf(fp,"\n");
}
// specified nodal potential/displacement
NOFIX=(int *)calloc(NVFIX,sizeof(int));
ITCLEAN(NVFIX,1,NOFIX);
IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
ITCLEAN(NVFIX,NDOFN,IFPRE);
PRESC=(double *)calloc(NVFIX*NDOFN,sizeof(double));
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    NOFIX[i]=n1-1;
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
    }
    for(j=0;j<NDOFN;j++)
```

```
        {
            fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
        }
        fscanf(fp,"\n");
    }
    // specified concentrated loads at nodes
    fgets(dummy,200,fp);
    if(NPLOD>0)
    {
        LODPT=(int *)calloc(NPLOD*1,sizeof(int));
        ITCLEAN(NPLOD,1,LODPT);
        POINT=(double *)calloc(NPLOD*NDOFN,
            sizeof(double));
        DUCLEAN(NPLOD,NDOFN,POINT);
        fgets(dummy,200,fp);
        for(i=0;i<NPLOD;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            LODPT[i]=n1-1;
            for(j=0;j<NDOFN;j++)
            {
                fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
            }
            fscanf(fp,"\n");
        }
    }
    // specified distributed edge loads
    fgets(dummy,200,fp);
    if(NDLEG>0)
    {
        NEASS=(int *)calloc(NDLEG*1,sizeof(int));
        ITCLEAN(NDLEG,1,NEASS);
        NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
        ITCLEAN(NDLEG,NODEG,NOPRS);
        TNFEG=NODEG*NDOFN;
        PRESS=(double *)calloc(NDLEG*TNFEG,sizeof(double));
        DUCLEAN(NDLEG,TNFEG,PRESS);
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
```

```
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NPROP;j++)
        {
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }

    /** Establish local relations of nodes and edges **/
    ELNOD=(int *)calloc(NEDGE*NODEG,sizeof(int));
    ITCLEAN(NEDGE,NODEG,ELNOD);
    TYPELEM(ELNOD);

    /** Form stiffness matrix **/
    NEQNS=NPOIN*NDOFN;
    GSTIF=(double *)calloc(NEQNS*NEQNS,sizeof(double));
    DUCLEAN(NEQNS,NEQNS,GSTIF);
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        // Compute some quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
```

```
        HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
        // Compute G matrix
        NEVAB=NNODE*NDOFN;
        EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
        DUCLEAN(NTREF,NEVAB,EGMTX);
        GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
        // Compute element stiffness matrix
        ESTIF=(double *)calloc(NEVAB*NEVAB,sizeof(double));
        DUCLEAN(NEVAB,NEVAB,ESTIF);
        KMATRIX(EHMTX,EGMTX,ESTIF);
        // Assemble stiffness matrix
        ASMSTIF(iELEM,NEQNS,LNODS,ESTIF,GSTIF);
        free(EHMTX); free(EGMTX); free(ESTIF);
    }
    // Compute equivalent loads
    GLOAD=(double *)calloc(NEQNS*1,sizeof(double));
    DUCLEAN(NEQNS,1,GLOAD);
    PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,NOPRS,
        PRESS,NPLOD,LODPT,POINT,GLOAD);

    // Introduce constrained displacements
    INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,GSTIF,GLOAD);

    // Solve linear system of equations
    LSSOLVR(GSTIF,GLOAD,NEQNS);

    // Output nodal displacement
    UPOIN=(double *)calloc(NPOIN*NDOFN,sizeof(double));
    DUCLEAN(NPOIN,NDOFN,UPOIN);
    FIEDNOD(NPOIN,GLOAD,UPOIN);

    // Compute quantities at centroid of each element
    CECOD=(double *)calloc(NELEM*NDIME,sizeof(double));
    DUCLEAN(NELEM,NDIME,CECOD);
    UCENP=(double *)calloc(NELEM*NDOFN,sizeof(double));
    DUCLEAN(NELEM,NDOFN,UCENP);
    SCENP=(double *)calloc(NELEM*NSTRE,sizeof(double));
    DUCLEAN(NELEM,NSTRE,SCENP);
    FIEDCEN(NELEM,MATNO,LNODS,COORD,PROPS,ELNOD,GLOAD,
        CECOD,UCENP,SCENP);

    // Output results
    OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,
        NVFIX,NPLOD,NDLEG);
```

```
    free(COORD); free(LNODS); free(MATNO);
    free(NOFIX); free(IFPRE); free(PRESC);
    free(PROPS); free(ECOOD); free(CenCoord);
    free(GSTIF); free(GLOAD); free(UPOIN);
    free(CECOD); free(UCENP); free(SCENP);
    printf("------------ Finished ------------\n");
    return;
}


/*
*********************************************************
* Subroutine PVECTOR                                    *
* - Compute effective nodal force                       *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void PVECTOR(int MATNO[],double PROPS[],int LNODS[],
             double COORD[],int NDLEG,int NEASS[],
             int NOPRS[],double PRESS[],int NPLOD,
             int LODPT[],double POINT[],double GP[])
{
    void GAUSSQU();
    void SHAPFUN();
    void DUCLEAN();
    extern int NDIME,NDOFN,NGAUS,NPROP,NNODE,NEDGE,
        NODEG;
    double *POSGP,*WEIGP,*ELCOD,*EPRES,*PE,*SHAPE,
        *DSHAP,*CORGS,*DERGS,*PGASH;
    int iPLOD,iDLEG,iNODE,kNODE,iODEG,iDOFN,iDIME,
        jGAUS,kPOIN,kELEM,kMATS,kEQNS,iEVAB,n1,n2,
        NEVAB,TNFEG;
    double EXISP,THICK,DVOLU,DS1,DS2,PX,PY;
    // Add point loads at nodes to global load vector
    if(NPLOD>0)
    {
        for(iPLOD=0;iPLOD<NPLOD;iPLOD++)
        {
            for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
            {
                n1=LODPT[iPLOD]*NDOFN+iDOFN;
                n2=iPLOD*NDOFN+iDOFN;
                GP[n1]=GP[n1]+POINT[n2];
```

```
            }
        }
    }
    // Evaluate equivalent nodal force caused
    // by distributed edge load
    NEVAB=NNODE*NDOFN;
    TNFEG=NODEG*NDOFN;
    // Gaussian point and weight coefficients
    POSGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,POSGP);
    WEIGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,WEIGP);
    GAUSSQU(POSGP,WEIGP);
    for(iDLEG=0;iDLEG<NDLEG;iDLEG++)
    {
        kELEM=NEASS[iDLEG];
        // Material properties
        kMATS=MATNO[kELEM];
        THICK=PROPS[kMATS*NPROP+2];
        // Determine coordinates of nodes on the element edge
        ELCOD=(double *)calloc(NODEG*NDIME,sizeof(double));
        DUCLEAN(NODEG,NDIME,ELCOD);
        for(iODEG=0;iODEG<NODEG;iODEG++)
        {
            kPOIN=NOPRS[iDLEG*NODEG+iODEG];
            for(iDIME=0;iDIME<NDIME;iDIME++)
            {
                n1=iODEG*NDIME+iDIME;
                n2=kPOIN*NDIME+iDIME;
                ELCOD[n1]=COORD[n2];
            }
        }
        // Determine local nodal load intensity
        EPRES=(double *)calloc(NODEG*NDOFN,sizeof(double));
        DUCLEAN(NODEG,NDOFN,EPRES);
        for(iODEG=0;iODEG<NODEG;iODEG++)
        {
            for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
            {
                n1=iODEG*NDOFN+iDOFN;
                n2=iDLEG*TNFEG+n1;
                EPRES[n1]=PRESS[n2];
            }
        }
        // Integration along loaded edge
```

```
PE=(double *)calloc(NEVAB,sizeof(double));
DUCLEAN(NEVAB,1,PE);
for(jGAUS=0;jGAUS<NGAUS;jGAUS++)
{
    EXISP=POSGP[jGAUS];
    // shape functions and its derivatives
    SHAPE=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,SHAPE);
    DSHAP=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,DSHAP);
    SHAPFUN(EXISP,SHAPE,DSHAP);
    // Coordinates and derivatives of Gauss points
    // x=sum(Ni*xi) and y=sum(Ni*yi)
    // dx/dt=sum(dNi/dt*xi), dy/dt=sum(dNi/dt*yi)
    // DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    CORGS=(double *)calloc(NDIME,sizeof(double));
    DUCLEAN(1,NDIME,CORGS);
    DERGS=(double *)calloc(NDIME,sizeof(double));
    DUCLEAN(1,NDIME,DERGS);
    for(iDIME=0;iDIME<NDIME;iDIME++)
    {
        for(iODEG=0;iODEG<NODEG;iODEG++)
        {
            n1=iODEG*NDIME+iDIME;
            CORGS[iDIME]=CORGS[iDIME]+
                SHAPE[iODEG]*ELCOD[n1];
            DERGS[iDIME]=DERGS[iDIME]+
                DSHAP[iODEG]*ELCOD[n1];
        }
    }
    DVOLU=sqrt(pow(DERGS[0],2.0)+
        pow(DERGS[1],2.0));
    // Direction cosine at Gaussian points
    DS1= DERGS[1]/DVOLU;
    DS2=-DERGS[0]/DVOLU;
    // Gauss integration factor
    DVOLU=DVOLU*WEIGP[jGAUS];
    if((THICK+1)!=1)
    {
        DVOLU=DVOLU*THICK;
    }
    // Load intensity at Gaussian point
    //  pn=sum(Ni*pni),  pt=sum(Ni*pti)
    PGASH=(double *)calloc(NDOFN,sizeof(double));
    DUCLEAN(NDOFN,1,PGASH);
```

```
        for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
        {
            for(iODEG=0;iODEG<NODEG;iODEG++)
            {
                n1=iODEG*NDOFN+iDOFN;
                PGASH[iDOFN]=PGASH[iDOFN]+
                    SHAPE[iODEG]*EPRES[n1];
            }
        }
        // px=-pn*nx-pt*ny
        // py=-pn*ny+pt*nx
        PX=-DS1*PGASH[0]-DS2*PGASH[1];
        PY=-DS2*PGASH[0]+DS1*PGASH[1];
        PGASH[0]=PX;
        PGASH[1]=PY;
        // Compute equivalent force PE
        for(iNODE=0;iNODE<NNODE;iNODE++)
        {
            kPOIN=LNODS[kELEM*NNODE+iNODE];
            if(kPOIN==NOPRS[iDLEG*NODEG+0])
            {
                // iNODE is start node of the
                // loaded edge
                for(iODEG=0;iODEG<NODEG;iODEG++)
                {
                    kNODE=iNODE+iODEG;
                    if(kNODE>=NNODE)
                    {
                        kNODE=0;
                    }
                    for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
                    {
                        n1=kNODE*NDOFN+iDOFN;
                        PE[n1]=PE[n1]+SHAPE[iODEG]*
                            PGASH[iDOFN]*DVOLU;
                    }
                }
            }
        }
    }
    // Assemble PE into global load vector
    for(iNODE=0;iNODE<NNODE;iNODE++)
    {
        kPOIN=LNODS[kELEM*NNODE+iNODE];
        for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
```

```
            {
                kEQNS=NDOFN*kPOIN+iDOFN; // global DOF
                iEVAB=NDOFN*iNODE+iDOFN; // local DOF
                GP[kEQNS]=GP[kEQNS]+PE[iEVAB];
            }
        }
    }
    free(POSGP); free(WEIGP); free(ELCOD);
    free(EPRES); free(PE);  free(SHAPE);
    free(DSHAP); free(DERGS); free(PGASH);
    return;
}


/*
*******************************************************
* Subroutine FIEDCEN                                  *
* -Compute related fields at centroid of each element *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDCEN(int NELEM,int MATNO[],int LNODS[],
             double COORD[],double PROPS[],int ELNOD[],
             double ASDIS[],double CECOD[],double UCENP[],
             double SCENP[])
{
    void ELEPARS(); void DUCLEAN(); void HMATRIX();
    void GMATRIX(); void EDISNOD(); void CMATRIX();
    void RIGIDRV(); void TREFFTZ(); void MATMULT();
    extern int NTREF,NNODE,NDIME,NDOFN,NSTRE;
    int iELEM,kMATS,NEVAB;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*d_Ele,*c_Ele,
        *c0,*N_SET,*T_SET,*GDISP,*GSTRE,xp,yp;

    NEVAB=NNODE*NDOFN;
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        // Compute some quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
```

```
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
        HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
        // Compute G matrix
        EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
        DUCLEAN(NTREF,NEVAB,EGMTX);
        GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
        // Nodal displacements
        d_Ele=(double *)calloc(NEVAB,sizeof(double));
        DUCLEAN(NEVAB,1,d_Ele);
        EDISNOD(iELEM,LNODS,ASDIS,d_Ele);
        // Calculate the ce coefficients
        c_Ele=(double *)calloc(NTREF*1,sizeof(double));
        DUCLEAN(NTREF,1,c_Ele);
        CMATRIX(EHMTX,EGMTX,d_Ele,c_Ele);
        // Recover rigid displacement
        c0=(double *)calloc(3*1,sizeof(double));
        DUCLEAN(3,1,c0);
        RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,PROPS,c0);
        // Compute Trefftz internal fields at central point
        N_SET=(double *)calloc(NDOFN*NTREF,sizeof(double));
        DUCLEAN(NDOFN,NTREF,N_SET);
        T_SET=(double *)calloc(NSTRE*NTREF,sizeof(double));
        DUCLEAN(NSTRE,NTREF,T_SET);
        xp=0;
        yp=0;
        TREFFTZ(xp,yp,kMATS,PROPS,N_SET,T_SET);
        GDISP=(double *)calloc(NDOFN*1,sizeof(double));
        DUCLEAN(NDOFN,1,GDISP);
        GSTRE=(double *)calloc(NSTRE*1,sizeof(double));
        DUCLEAN(NSTRE,1,GSTRE);
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,GDISP);
        MATMULT(T_SET,c_Ele,NSTRE,NTREF,1,GSTRE);
        UCENP[iELEM*NDOFN+0]=GDISP[0]+c0[0]+yp*c0[2];
        UCENP[iELEM*NDOFN+1]=GDISP[1]+c0[1]-xp*c0[2];
        SCENP[iELEM*NSTRE+0]=GSTRE[0];
        SCENP[iELEM*NSTRE+1]=GSTRE[1];
        SCENP[iELEM*NSTRE+2]=GSTRE[2];
        // Coordinates of commputing point
        CECOD[iELEM*NDIME+0]=CenCoord[0]+xp;
        CECOD[iELEM*NDIME+1]=CenCoord[1]+yp;
    }
    free(ECOOD); free(CenCoord); free(EHMTX);
```

```
    free(EGMTX); free(d_Ele); free(c_Ele);
    free(N_SET); free(T_SET); free(GDISP);
    free(GSTRE);
}



/*
*********************************************************
* Subroutine QUANGAS                                    *
* - Evaluate quantities at Gaussian points              *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void QUANGAS(int iEDGE,double EXISP,double ECOOD[],
             int ELNOD[],double CORGS[],double *DVOLU,
             double AMTRX[],double SHMTX[])
{
    void SHAPFUN();
    void DUCLEAN();
    extern int NDIME,NDOFN,NNODE,NEDGE,NODEG;
    double *SHAPE,*DSHAP,*DERGS,DS1,DS2;
    int ii,jj,in,kn,NEVAB;

    NEVAB=NNODE*NDOFN;
    // shape functions and its derivatives
    SHAPE=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,SHAPE);
    DSHAP=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,DSHAP);
    SHAPFUN(EXISP,SHAPE,DSHAP);
    // Coordinates and derivatives of Gauss points
    //   x=sum(Ni*xi) and y=sum(Ni*yi)
    //   dx/dt=sum(dNi/dt*xi), dy/dt=sum(dNi/dt*yi)
    //   DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    DERGS=(double *)calloc(NDIME,sizeof(double));
    DUCLEAN(1,NDIME,DERGS);
    for(ii=0;ii<NDIME;ii++)
    {
        for(jj=0;jj<NODEG;jj++)
        {
            kn=ELNOD[iEDGE*NODEG+jj];
            CORGS[ii]=CORGS[ii]+
```

```
                SHAPE[jj]*ECOOD[kn*NDIME+ii];
            DERGS[ii]=DERGS[ii]+
                DSHAP[jj]*ECOOD[kn*NDIME+ii];
        }
    }
    *DVOLU=sqrt(pow(DERGS[0],2.0)+pow(DERGS[1],2.0));
    // Directional cosine at Gauss point
    //   nx = dy/dS,  ny = - dx/dS
    //   A=[n1,  0, n2
    //       0, n2, n1]
    DS1= DERGS[1]/(*DVOLU);
    DS2=-DERGS[0]/(*DVOLU);
    AMTRX[0*3+0]=DS1;
    AMTRX[0*3+2]=DS2;
    AMTRX[1*3+1]=DS2;
    AMTRX[1*3+2]=DS1;
    // Form shape function matrix of frame function
    in=ELNOD[iEDGE*NODEG+0];
    for(ii=0;ii<NODEG;ii++)
    {
        kn=in+ii;
        if(kn>=NNODE)
        {
            kn=0;
        }
        SHMTX[0*NEVAB+2*kn  ]=SHAPE[ii];
        SHMTX[1*NEVAB+2*kn+1]=SHAPE[ii];
    }
    return;
}


/*
*******************************************************
* Subroutine TREFFTZ                                  *
* - Evaluate Trefftz functions truncated with         *
*   specified terms number                            *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void TREFFTZ(double sp,double tp,int kMATS,
             double PROPS[],double N_SET[],
             double T_SET[])
```

```
{
    void DUCLEAN();
    void ITCLEAN();
    void ocpow();
    void ocmul();
    extern int NTREF,NTYPE,NNODE,NDOFN,NPROP;
    int temp,im,km,nt,k,n1,n2,*NK;
    double u1,v1,u2,v2,u3,v3,u4,v4,u5,v5;
    double YOUNG,POISS,KAMA,GMOD,*RZK,*RRK,*RSK,*IZK,*IRK,*ISK;
    // Check the number of terms of Trefftz functions
    temp=NNODE*NDOFN-3;
    if(NTREF<temp)
    {
        printf("Small number of Trefftz functions!");
        exit(0);
    }
    //
    YOUNG=PROPS[kMATS*NPROP+0];
    POISS=PROPS[kMATS*NPROP+1];
    if(NTYPE==1) // plane stress
    {
        KAMA=(3.0-POISS)/(1.0+POISS);
    }
    else if(NTYPE==2) // plane strain
    {
        KAMA=3.0-4*POISS;
    }
    GMOD=YOUNG/(2.0*(1+POISS));

    RZK=(double *)calloc(NTREF,sizeof(double));
    RRK=(double *)calloc(NTREF,sizeof(double));
    RSK=(double *)calloc(NTREF,sizeof(double));
    DUCLEAN(1,NTREF,RZK);
    DUCLEAN(1,NTREF,RRK);
    DUCLEAN(1,NTREF,RSK);
    IZK=(double *)calloc(NTREF,sizeof(double));
    IRK=(double *)calloc(NTREF,sizeof(double));
    ISK=(double *)calloc(NTREF,sizeof(double));
    DUCLEAN(1,NTREF,IZK);
    DUCLEAN(1,NTREF,IRK);
    DUCLEAN(1,NTREF,ISK);

    k=1;
    ocmul(KAMA-1,0.0,sp,tp,&RZK[0],&IZK[0]);
    ocmul( 0.0,1.0,sp,-tp,&RZK[1],&IZK[1]);
```

```
ocmul(-1.0,0.0,sp,-tp,&RZK[2],&IZK[2]);

RRK[0]=2.0; IRK[0]=0.0;
RRK[1]=0.0; IRK[1]=0.0;
RRK[2]=0.0; IRK[2]=0.0;

RSK[0]=0.0; ISK[0]=0.0;
RSK[1]=0.0; ISK[1]=1.0;
RSK[2]=1.0; ISK[2]=0.0;

NK=(int *)calloc(NTREF,sizeof(int));
ITCLEAN(1,NTREF,NK);
NK[0]=k; NK[1]=k; NK[2]=k;
nt=3;
while(nt<NTREF)
{
    k=k+1;

    ocpow(sp,tp,k,&u1,&v1);
    ocmul(0.0,KAMA,u1,v1,&u2,&v2);
    ocmul(0.0,1.0*k,sp,tp,&u3,&v3);
    ocpow(sp,-tp,k-1,&u4,&v4);
    ocmul(u3,v3,u4,v4,&u5,&v5);
    RZK[nt]=u2+u5; IZK[nt]=v2+v5;

    ocpow(sp,tp,k-1,&u1,&v1);
    ocmul(0.0,2.0*k,u1,v1,&u2,&v2);
    RRK[nt]=u2; IRK[nt]=v2;

    ocmul(0.0,1.0*k*(k-1),sp,-tp,&u1,&v1);
    ocpow(sp,tp,k-2,&u2,&v2);
    ocmul(u1,v1,u2,v2,&u3,&v3);
    RSK[nt]=u3; ISK[nt]=v3;
    NK[nt]=k;

    nt=nt+1;
    ocpow(sp,tp,k,&u1,&v1);
    ocpow(sp,-tp,k-1,&u2,&v2);
    ocmul(sp,tp,u2,v2,&u3,&v3);
    RZK[nt]=KAMA*u1-k*u3; IZK[nt]=KAMA*v1-k*v3;

    ocpow(sp,tp,k-1,&u1,&v1);
    RRK[nt]=2*k*u1; IRK[nt]=2*k*v1;

    ocpow(sp,tp,k-2,&u1,&v1);
```

```
        ocmul(u1,v1,sp,-tp,&u2,&v2);
        RSK[nt]=k*(k-1)*u2; ISK[nt]=k*(k-1)*v2;
        NK[nt]=k;

        nt=nt+1;
        if(nt>=NTREF)
        {
            break;
        }
        ocpow(sp,-tp,k,&u1,&v1);
        ocmul(0.0,1.0,u1,v1,&u2,&v2);
        RZK[nt]=u2; IZK[nt]=v2;

        RRK[nt]=0.0; IRK[nt]=0.0;

        ocpow(sp,tp,k-1,&u1,&v1);
        ocmul(0.0,k*1.0,u1,v1,&u2,&v2);
        RSK[nt]=u2; ISK[nt]=v2;
        NK[nt]=k;

        nt=nt+1;
        ocpow(sp,-tp,k,&u1,&v1);
        RZK[nt]=-u1; IZK[nt]=-v1;

        RRK[nt]=0.0; IRK[nt]=0.0;

        ocpow(sp,tp,k-1,&u1,&v1);
        RSK[nt]=k*u1; ISK[nt]=k*v1;
        NK[nt]=k;

        nt=nt+1;
    }
    for(im=0;im<NTREF;im++)
    {
        N_SET[0*NTREF+im]=RZK[im]/(2.0*GMOD);
        N_SET[1*NTREF+im]=IZK[im]/(2.0*GMOD);
        k=NK[im];
        n1=4*(k-2)+0;
        n2=4*(k-2)+1;
        km=im-3;
        if((im==0)||(km==n1)||(km==n2))
        {
            T_SET[0*NTREF+im]=RRK[im]-RSK[im];
            T_SET[1*NTREF+im]=RRK[im]+RSK[im];
            T_SET[2*NTREF+im]=ISK[im];
```

```
        }
        else
        {
            T_SET[0*NTREF+im]=-RSK[im];
            T_SET[1*NTREF+im]= RSK[im];
            T_SET[2*NTREF+im]= ISK[im];
        }
    }
    return;
}


/*
*******************************************************
* Subroutine RIGIDRV                                  *
* - Recovery of rigid body motion                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void RIGIDRV(double ECOOD[],double c_Ele[],
             double d_Ele[],int kMATS,double PROPS[],
             double rvect[])
{
    void DUCLEAN();
    void MATMULT();
    void TREFFTZ();
    void LSSOLVR();
    extern int NTREF,NDIME,NDOFN,NNODE,NSTRE;
    double x1,x2,*N_SET,*T_SET,*RMATX,du1,du2,*u;
    int iNODE;

    RMATX=(double *)calloc(3*3,sizeof(double));
    DUCLEAN(3,3,RMATX);
    N_SET=(double *)calloc(NDOFN*NTREF,sizeof(double));
    T_SET=(double *)calloc(NSTRE*NTREF,sizeof(double));
    for(iNODE=0;iNODE<NNODE;iNODE++)
    {
        x1=ECOOD[iNODE*NDIME+0];
        x2=ECOOD[iNODE*NDIME+1];
        DUCLEAN(NDOFN,NTREF,N_SET);
        DUCLEAN(NSTRE,NTREF,T_SET);
        TREFFTZ(x1,x2,kMATS,PROPS,N_SET,T_SET);
        u=(double *)calloc(NDOFN*1,sizeof(double));
```

```
        DUCLEAN(NDOFN,1,u);
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,u);

        du1=d_Ele[iNODE*2  ]-u[0];
        du2=d_Ele[iNODE*2+1]-u[1];
        rvect[0]=rvect[0]+du1;
        rvect[1]=rvect[1]+du2;
        rvect[2]=rvect[2]+x2*du1-x1*du2;

        RMATX[0*3+2]=RMATX[0*3+2]+x2;
        RMATX[1*3+2]=RMATX[1*3+2]-x1;
        RMATX[2*3+2]=RMATX[2*3+2]+(pow(x1,2)+pow(x2,2));
    }
    RMATX[2*3+0]=RMATX[0*3+2];
    RMATX[2*3+1]=RMATX[1*3+2];
    RMATX[0*3+0]=NNODE;
    RMATX[1*3+1]=NNODE;
    // R*c0=r
    LSSOLVR(RMATX,rvect,3);
    // after this the rvect stores c0
    free(N_SET); free(T_SET), free(RMATX);
    return;
}



/*
*********************************************************
* Subroutines ocdiv, oclog, ocexp, opowr, ocmul,      *
*             ocactan                                   *
* - complex algorithm                                   *
*********************************************************
*/
/* complex division (a+ib)/(c+id) */
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void ocdiv(double a,double b,double c,double d,
           double *e,double *f)
{
    double p,q,s,w;
    p=a*c;
    q=-b*d;
    s=(a+b)*(c-d);
    w=c*c+d*d;
```

```
    if((1+w)==1)
    {
        printf("Error, the denominator is zero!\n");
        exit(0);
    }
    *e=(p-q)/w;
    *f=(s-p-q)/w;
    return;
}

/* Complex logarithm ln((x+iy)) */
void oclog(double x,double y,double *u,double *v)
{
    double p;
    p=log(sqrt(x*x+y*y));
    *u=p;
    *v=atan2(y,x);
    return;
}

/* Complex exponent exp((x+iy)) */
void ocexp(double x,double y,double *u,double *v)
{
    double p;
    p=exp(x);
    *u=p*cos(y);
    *v=p*sin(y);
    return;
}

/* Complex power (x+iy)^n */
void ocpow(double x,double y,int n,double *u,double *v)
{
    double r,q;
    if(fabs(y/x)<1e-12)
    {
        y=0.0;
    }
    q=atan2(y,x);
    r=sqrt(x*x+y*y);
    if (r>1e-10)
    {
        r=n*log(r);
        r=exp(r);
    }
```

```
    *u=r*cos(n*q);
    *v=r*sin(n*q);
    return;
}

/* Complex multiply (a+ib)*(c+id) */
void ocmul(double a,double b,double c,double d,
           double *e,double *f)
{
    double p,q,s;
    p=a*c;
    q=b*d;
    s=(a+b)*(c+d);
    *e=p-q;
    *f=s-p-q;
    return;
}

/* Complex arctan((a+ib)) */
void ocactan(double a,double b,double *c,double *d)
{
  void ocdiv();
  void oclog();
  double e,f,u,v;
  ocdiv(1-b,a,1+b,-a,&e,&f);
  oclog(e,f,&u,&v);
  *c=v/2;
  *d=-u/2;
}
```

6.9 Numerical examples

Example 6.1 Square plate under uniform tension
In the first test problem, the plane stress application considered is the square plate shown in Figure 6.7, where the dimensions and boundary conditions are specified. The material properties assumed are elastic modulus $E = 1000$, Poisson's ratio $\nu = 0.3$, thickness $t = 1.0$, and coefficient of thermal expansion $\alpha = 0.001$.

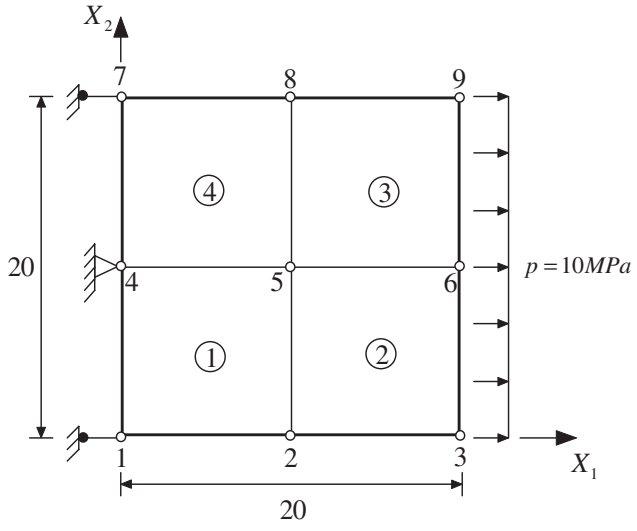


FIGURE 6.7
Square plate under simple uniform tension

The corresponding analytical solutions of displacements and stresses are given by

$$u = \frac{pX_1}{E}, \quad v = \frac{p\nu\left(\frac{a}{2} - X_2\right)}{E}$$
$$\sigma_{11} = p, \ \ \sigma_{22} = 0, \ \ \sigma_{12} = 0 \tag{6.91}$$

In our computation, only four 4-node regular elements are employed to discretise the square domain, and the terms of Trefftz functions are chosen to be 7 to meet the requirement specified in Eq. (6.61). The discretised data and numerical results obtained from the HT-FEM described in this chapter are presented below. It is observed that the results obtained are in good agreement with the analytical solutions. The results also show the linear variation of displacements and constant stress state within the solution domain. This matches the expected field distribution for a 2D elastic body to subject to a uniform tension.

```
*********************************************
HTFEM for plane stress square plate problems
*********************************************
NTREF NTYPE
7     1
NNODE NEDGE NODEG
4     4     2
NDIME NDOFN NSTRE
2     2     3
NMATS NPROP NGAUS
```

```
1     5     5
NPOIN NELEM NVFIX NPLOD NDLEG
9     4     3     0     2
-- Read elementary connections and material numbers
Elem# Mat# Node#1--->#NNODE
1     1    1   2   5   4
2     1    2   3   6   5
3     1    5   6   9   8
4     1    4   5   8   7
-- Read nodal coordinates
Node# Coord#1--->#NDIME
1      0.00    0.00
2     10.00    0.00
3     20.00    0.00
4      0.00   10.00
5     10.00   10.00
6     20.00   10.00
7      0.00   20.00
8     10.00   20.00
9     20.00   20.00
-- Read constrained boundary conditions
Num# Node# DOF#1--->DOF#NDOFN Val#1--->Val#NDOFN
1    1     1   0   0.00   0.00
2    4     1   1   0.00   0.00
3    7     1   0   0.00   0.00
-- No Concentrated loads
-- Distributed edge loads
Num# Ele# Node#1--->#NODEG Val#1--->#NODEG*NDOFN
1    2    3   6   -10.0   0.0   -10.0   0.0
2    3    6   9   -10.0   0.0   -10.0   0.0
-- Read material properties
Mat# Pro#1--->Pro#NPROP
1    1000.0   0.3   1.0   0.0   0.001


===============================================
**Basic parameters

NTREF=    7  NTYPE=    1

NDIME=    2  NDOFN=    2  NSTRE=    3

NNODE=    4  NEDGE=    4  NODEG=    2

NMATS=    1  NPROP=    5  NGAUS=    5
```

```
NPOIN=     9  NELEM=     4  NVFIX=     3

NPLOD=     0  NDLEG=     2


**Generalised displacements at nodes
-------------------------------------
Node# COD#1-->#NDIME DISPL#1-->#NODFN
-------------------------------------
    1     0.0000     0.0000     0.0000     0.0300
    2    10.0000     0.0000     0.1000     0.0300
    3    20.0000     0.0000     0.2000     0.0300
    4     0.0000    10.0000     0.0000    -0.0000
    5    10.0000    10.0000     0.1000    -0.0000
    6    20.0000    10.0000     0.2000    -0.0000
    7     0.0000    20.0000     0.0000    -0.0300
    8    10.0000    20.0000     0.1000    -0.0300
    9    20.0000    20.0000     0.2000    -0.0300


**Generalised Displacement at Element Centroid
-------------------------------------
Elem# COD#1-->#NDIME DISPL#1-->#NODFN
-------------------------------------
    1     5.0000     5.0000     0.0500     0.0150
    2    15.0000     5.0000     0.1500     0.0150
    3    15.0000    15.0000     0.1500    -0.0150
    4     5.0000    15.0000     0.0500    -0.0150

**Generalised Stress at Element Centroid
-------------------------
Elem#    STRES#1-->#NSTRE
-------------------------
    1    10.0000    -0.0000     0.0000
    2    10.0000     0.0000     0.0000
    3    10.0000     0.0000    -0.0000
    4    10.0000    -0.0000    -0.0000
```

Example 6.2 Thick circular cylinder under internal pressure [12]
The second problem considered is the case of a thick circular cylinder under internal pressure, which conforms to plane strain conditions. Due to axisymmetric properties, one quarter of the model is considered and the corresponding geometrical dimensions and boundary conditions are shown in Figure 6.8.

The related theoretical solutions are given as

$$u_r = \frac{1+\nu}{E}\left(-\frac{A}{r} + 2C(1-2\nu)r\right)$$
$$u_s = 0 \tag{6.92}$$

and

$$\sigma_r = \frac{A}{r^2} + 2C$$
$$\sigma_\theta = -\frac{A}{r^2} + 2C \tag{6.93}$$
$$\sigma_{r\theta} = 0$$

where

$$A = -\frac{r_i^2 r_o^2}{r_o^2 - r_i^2}p, \quad C = \frac{r_i^2}{2\left(r_o^2 - r_i^2\right)}p$$

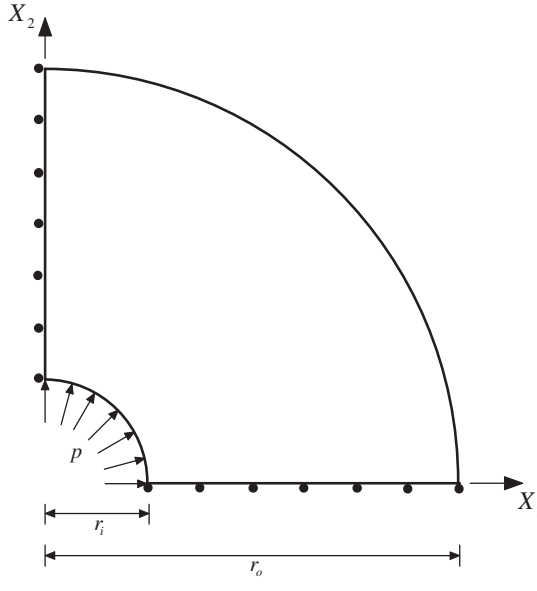


FIGURE 6.8
Configuration of thick cylinder under internal pressure

In the computation, the assumed material properties are the same as those in Example 6.1. The internal pressure of 10 units is considered, with the external boundary being unloaded. The element subdivision employed in the solution is illustrated in Figure 6.9 and it can be seen that nine parabolic elements have been utilised in the

computation. Additionally, taking into consideration the requirement of minimum terms of Trefftz functions, that is, $m_{\min} = 2 \times 8 - 3 = 13$, we set $m = 15$ in our calculation.

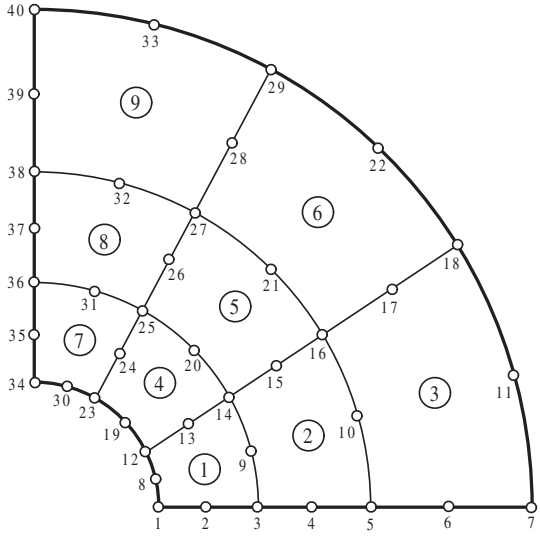


FIGURE 6.9
Mesh discretisation of a quarter of the thick cylinder

Radial displacement, radial and hoop stress distributions are obtained and compared with the analytical solutions in Figure 6.10 and Table 6.1, from which it can be seen that good agreement is achieved between the numerical results and theoretical solutions.

TABLE 6.1
Radial and hoop stress comparison for internal pressure loading

r	σ_r		σ_θ	
	Numerical	Theoretical	Numerical	Theoretical
6.4964	−5.7311	−5.6521	7.0593	6.9854
10.3941	−1.8172	−1.8016	3.1436	3.1350
16.0784	−0.3715	−0.3649	1.7017	1.6982

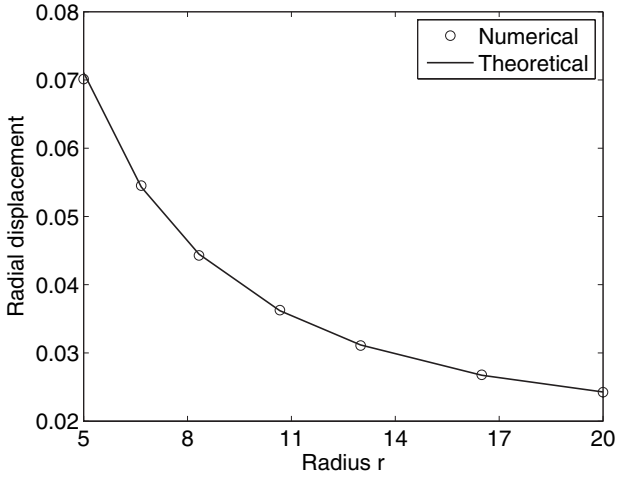


FIGURE 6.10
Radial displacement distribution due to internal pressure loading

Finally, the full input data and numerical results are displayed below.

```
*********************************************
HTFEM for plane strain thick cylinder problems
*********************************************
NTREF NTYPE
15    2
NNODE NEDGE NODEG
8     4     3
NDIME NDOFN NSTRE
2     2     3
NMATS NPROP NGAUS
1     5     5
NPOIN NELEM NVFIX NPLOD NDLEG
40    9     14    0     3
-- Read elementary connections and material numbers
Elem# Mat# Node#1-->#NNODE
1     1     1    2    3    9   14   13   12    8
2     1     3    4    5   10   16   15   14    9
3     1     5    6    7   11   18   17   16   10
4     1    12   13   14   20   25   24   23   19
5     1    14   15   16   21   27   26   25   20
6     1    16   17   18   22   29   28   27   21
7     1    23   24   25   31   36   35   34   30
8     1    25   26   27   32   38   37   36   31
```

```
9     1    27   28   29   33   40   39   38   32
-- Read nodal coordinates
Node# Coord#1-->#NDIME
 1     5.000      0.000
 2     6.667      0.000
 3     8.333      0.000
 4    10.667      0.000
 5    13.000      0.000
 6    16.500      0.000
 7    20.000      0.000
 8     4.830      1.294
 9     8.049      2.157
10    12.557      3.365
11    19.319      5.176
12     4.330      2.500
13     5.774      3.333
14     7.217      4.167
15     9.238      5.333
16    11.258      6.500
17    14.289      8.250
18    17.321     10.000
19     3.536      3.536
20     5.893      5.893
21     9.192      9.192
22    14.142     14.142
23     2.500      4.330
24     3.333      5.774
25     4.167      7.217
26     5.333      9.238
27     6.500     11.258
28     8.250     14.289
29    10.000     17.321
30     1.294      4.830
31     2.157      8.049
32     3.365     12.557
33     5.176     19.319
34     0.000      5.000
35     0.000      6.667
36     0.000      8.333
37     0.000     10.667
38     0.000     13.000
39     0.000     16.500
40     0.000     20.000
-- Read constrained boundary conditions
Num# Node# DOF#1-->#NDOFN Val#1-->#NDOFN
```

```
 1    1     0    1    0.0    0.0
 2    2     0    1    0.0    0.0
 3    3     0    1    0.0    0.0
 4    4     0    1    0.0    0.0
 5    5     0    1    0.0    0.0
 6    6     0    1    0.0    0.0
 7    7     0    1    0.0    0.0
 8   34     1    0    0.0    0.0
 9   35     1    0    0.0    0.0
10   36     1    0    0.0    0.0
11   37     1    0    0.0    0.0
12   38     1    0    0.0    0.0
13   39     1    0    0.0    0.0
14   40     1    0    0.0    0.0
-- No Concentrated load
-- Distributed edge load
Num# Ele# Node#1-->#NODEG Val#1-->#NODEG*NDOFN
1    1    12   8   1  10.0  0.0  10.0  0.0  10.0  0.0
2    4    23  19  12  10.0  0.0  10.0  0.0  10.0  0.0
3    7    34  30  23  10.0  0.0  10.0  0.0  10.0  0.0
-- Read material properties
Mat# Pro#1--->Pro#NPROP
1    1000.0   0.3   1.0   0.0   0.001


==========================================
**Basic parameters

NTREF=   15  NTYPE=    2

NDIME=    2  NDOFN=    2  NSTRE=    3

NNODE=    8  NEDGE=    4  NODEG=    3

NMATS=    1  NPROP=    5  NGAUS=    5

NPOIN=   40  NELEM=    9  NVFIX=   14

NPLOD=    0  NDLEG=    3


**Generalised displacements at nodes
-------------------------------------
Node# COD#1-->#NDIME DISPL#1-->#NODFN
-------------------------------------
```

1	5.0000	0.0000	0.0701	0.0000
2	6.6670	0.0000	0.0545	0.0000
3	8.3330	0.0000	0.0443	0.0000
4	10.6670	0.0000	0.0363	0.0000
5	13.0000	0.0000	0.0311	0.0000
6	16.5000	0.0000	0.0268	0.0000
7	20.0000	0.0000	0.0243	0.0000
8	4.8300	1.2940	0.0692	0.0185
9	8.0490	2.1570	0.0431	0.0116
10	12.5570	3.3650	0.0302	0.0081
11	19.3190	5.1760	0.0235	0.0063
12	4.3300	2.5000	0.0607	0.0351
13	5.7740	3.3330	0.0472	0.0273
14	7.2170	4.1670	0.0383	0.0221
15	9.2380	5.3330	0.0314	0.0181
16	11.2580	6.5000	0.0269	0.0155
17	14.2890	8.2500	0.0232	0.0134
18	17.3210	10.0000	0.0210	0.0121
19	3.5360	3.5360	0.0506	0.0506
20	5.8930	5.8930	0.0316	0.0316
21	9.1920	9.1920	0.0221	0.0221
22	14.1420	14.1420	0.0172	0.0172
23	2.5000	4.3300	0.0351	0.0607
24	3.3330	5.7740	0.0273	0.0472
25	4.1670	7.2170	0.0221	0.0383
26	5.3330	9.2380	0.0181	0.0314
27	6.5000	11.2580	0.0155	0.0269
28	8.2500	14.2890	0.0134	0.0232
29	10.0000	17.3210	0.0121	0.0210
30	1.2940	4.8300	0.0185	0.0692
31	2.1570	8.0490	0.0116	0.0431
32	3.3650	12.5570	0.0081	0.0302
33	5.1760	19.3190	0.0063	0.0235
34	0.0000	5.0000	0.0000	0.0701
35	0.0000	6.6670	0.0000	0.0545
36	0.0000	8.3330	0.0000	0.0443
37	0.0000	10.6670	0.0000	0.0363
38	0.0000	13.0000	0.0000	0.0311
39	0.0000	16.5000	0.0000	0.0268
40	0.0000	20.0000	0.0000	0.0243

```
**Generalised Displacement at Element Centroid
-------------------------------------
Elem# COD#1-->#NDIME DISPL#1-->#NODFN
```

```
-------------------------------------
    1     6.2750     1.6814     0.0536     0.0144
    2    10.0399     2.6903     0.0357     0.0096
    3    15.5305     4.1614     0.0262     0.0070
    4     4.5938     4.5938     0.0392     0.0392
    5     7.3498     7.3498     0.0261     0.0261
    6    11.3690    11.3690     0.0192     0.0192
    7     1.6814     6.2750     0.0144     0.0536
    8     2.6903    10.0399     0.0096     0.0357
    9     4.1614    15.5305     0.0070     0.0262

**Generalised Stress at Element Centroid
------------------------
Elem#    STRES#1-->#NSTRE
------------------------
    1    -4.8746     6.2027    -3.1972
    2    -1.4848     2.8112    -1.2403
    3    -0.2326     1.5628    -0.5183
    4     0.6652     0.6652    -6.3942
    5     0.6632     0.6632    -2.4803
    6     0.6651     0.6651    -1.0365
    7     6.2027    -4.8746    -3.1972
    8     2.8112    -1.4848    -1.2403
    9     1.5628    -0.2326    -0.5183
```

References

[1] Jirousek J, Teodorescu P (1982), Large finite elements method for the solution of problems in the theory of elasticity. *Comput Struct*, **15**: 575-587.

[2] Piltner R (1985), Special finite elements with holes and internal cracks. *Int J Numer Meth Eng*, **21**: 1471-1485.

[3] Jin WG, Cheung YK, Zienkiewicz OC (1990), Application of the Trefftz method in plane elasticity problems. *Int J Numer Meth Eng*, **30**: 1147-1161.

[4] Piltner R (1987), The use of complex valued functions for the solution of three-dimensional elasticity problems. *J Elasticity*, **18**: 191-225.

[5] Piltner R (1989), On the representation of three-dimensional elasticity solutions with the aid of complex valued functions. *J Elasticity*, **22**: 45-55.

[6] Jirousek J, Venkatesh A (1992), Hybrid Trefftz plane elasticity elements with *p*-method capabilities. *Int J Numer Meth Eng*, **35**: 1443-1472.

[7] Jirousek J, Wroblewski A (1996), T-elements: state of the art and future trends. *Arch Comput Method E*, **3**: 323-434.

[8] Qin QH (2005), Trefftz finite element method and its applications. *Appl Mech Rev*, **58**: 316-337.

[9] Timoshenko SP, Goodier JN (1951), *Theory of Elasticity* (2nd edition). New York: McGraw-Hill.

[10] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*. Southampton: WIT Press.

[11] Muskhelishvili NI (1953), *Some Basic Problems of the Mathematical Theory of Elasticity*. Groningen: P. Noordhoff Ltd.

[12] Hinton E, Owen DRJ (1977), *Finite Element Programming*. London: Academic Press.

7

Treatment of inhomogeneous terms using RBF approximation

7.1 Introduction

In Chapters 5 and 6, potential and plane elastic problems without generalised body force terms were considered. Absence of inhomogeneous terms such as body forces allows us to perform all integrals in element stiffness equations along the element boundary only. As a consequence, researchers can construct elements of any shape to meet their purposes. However, in practical engineering, there are many problems whose governing equations have a nonzero right-hand side. Typical examples are thermal sources in heat conduction, thermal loads caused by temperature change, body forces, inertial forces due to motion, and so on. For such problems, known as non-homogeneous problems, HT-FE formulation is disadvantageous compared to the conventional FEM in general, because domain integral and particular solutions are inevitably required and make the computational procedure more complex. This drawback weakens the main attraction of HT-FEM and is, in fact, a problem common to all boundary-type numerical methods.

In the computation of particular solutions, Qin [1] employed integrations of the corresponding source functions (also known as fundamental solutions) in potential problems. In the case of plane elastic problems, only constant body forces are discussed [1]. It is critical to develop effective approaches for the treatment of inhomogeneous terms that appear in the HT-FEM.

In this chapter, an approach to deal with arbitrary sources in potential problems and body force in elasticity is presented by way of radial basis functions (RBFs). The approach is based on the concept in dual-reciprocity boundary element method (DR-BEM) [2 - 8]. In DR-BEM, RBFs defined in terms of the Euclidean distance variable are used to approximate the distribution of the specified body forces, and then the related particular solution terms can be derived by means of an analytical integral procedure. Based on this procedure, the solution sought is first broken down into two parts: the particular part and the homogeneous or complementary part, and then RBF approximation and the HT-FEM are used to obtain the related particular and homogeneous solutions, respectively.

7.2 Radial basis functions

7.2.1 Basics of radial basis functions

In general, RBFs are defined as such functions that depend only on the Euclidean distance between any field point and a reference point (sometimes the central point of the element under consideration), so they are completely isotropic and can be easily used for scattered data interpolation in multi-dimensional space. Among the major advantages of RBFs are high accuracy, differentiability of the interpolation function, and absence of an underlying mesh. Applications of RBF interpolation include ocean-depth measurement, altitude measurement, rainfall interpolation, image warping, medical imaging, solving partial differential equations, and so on [9 - 13].

Traditionally, there are two categories of RBF. The first category is the piecewise polynomial compactly supported RBF (CS-RBF), which is defined in the local support domain chosen by the user [14 - 16] and positive definite on the definition domain for a given order smoothness. For instance, the normalized Wendland CS-RBF is written as

$$\phi_j(\mathbf{x}) = (1-r)_+^6 \left(35r^2 + 18r + 3\right) \tag{7.1}$$

with the notation

$$(1-r)_+^n = \begin{cases} (1-r)^n & if \;\; 0 \le r \le 1 \\ 0 & if \;\; r > 1 \end{cases} \tag{7.2}$$

The other category is the globally supported RBF (GS-RBF), which is defined in the entire domain under consideration. Table 7.1 lists some commonly used RBFs [17 - 19].

TABLE 7.1
Some commonly used radial basis functions (RBFs)

Piecewise smooth RBFs	Power spline (PS)*	$\phi_j(\mathbf{x}) = r_j^{2n-1}$
	Thin plate spline (TPS)	$\phi_j(\mathbf{x}) = r_j^{2n} \ln r_j$
Infinitely smooth RBFs	Multiquadric (MQ)	$\phi_j(\mathbf{x}) = \sqrt{r_j^2 + c^2}$
	Gaussian (GS)	$\phi_j(\mathbf{x}) = e^{-r_j^2}$

*Power spline (PS) RBF is also called a conical basis RBF in some references.

In the Table 7.1, $\phi_j(\mathbf{x}) = \phi(r_j)$ and r_j represents the Euclidean distance from the central point $\mathbf{x}_j$, also named as the reference point, to an arbitrary field point $\mathbf{x}$ in the

domain of interest (see Figure 7.1). For example, for a two-dimensional Cartesian coordinates system $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$, we have

$$r_j(x_1, x_2) = \sqrt{(x_1 - x_{j1})^2 + (x_2 - x_{j2})^2} \tag{7.3}$$

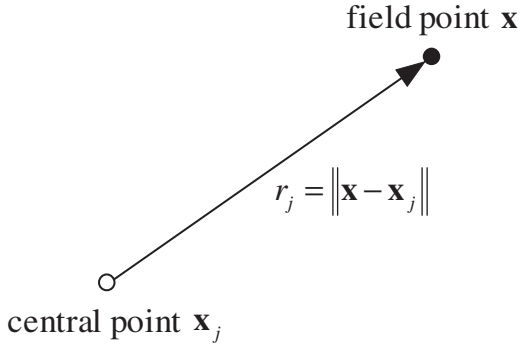


FIGURE 7.1
Definition of Euclidean distance in RBFs

Traditionally, linearly independent RBFs can easily be generated by choosing different central points. The ability to generate a large number of linearly independent functions may be considered an advantage of RBFs over polynomial functions. For the sake of convenience, only GS-RBFs are employed in this chapter.

7.2.2 RBF approximation

RBF approximation depends only on the Euclidean distance between two points, as shown in Figure 7.1. The distance is easy to compute in a problem with any number of space dimensions, so the presence of higher dimensions does not increase the difficulty in calculating RBFs. As a result, RBFs are widely used in practical engineering with various dimensions to approximate a given function.

In practice, RBF approximation works with points scattered throughout the domain of interest. As a consequence, any specified function can be interpolated by a linear combination of RBFs centered at the scattered points $\mathbf{x}_j$, that is,

$$f(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}) \tag{7.4}$$

where N is the number of central or reference points in the domain, ϕ_j represents a basis function centered at $\mathbf{x}_j$, and α_j denotes an unknown coefficient, which can usually be determined by making a match with given discrete data such as function

values, in a sense of, for example, satisfaction of the interpolation condition $f(\mathbf{x}_i) = f_i$ at each interior and boundary data point to yield linear algebraic equations

$$\begin{bmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_N(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_N(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \cdots & \phi_N(\mathbf{x}_N) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} \tag{7.5}$$

It is easily verified that the coefficient matrix in Eq. (7.5) is symmetric due to the relation

$$\phi_j(\mathbf{x}_i) = \phi_i(\mathbf{x}_j) = \phi\left(\left\|\mathbf{x}_i - \mathbf{x}_j\right\|\right) \tag{7.6}$$

To keep the collocation system unconditionally positive definite, the augmented interpolation strategy with monomial basis is also frequently employed by many researchers. For instance, let

$$f(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}) + \sum_{j=1}^{M} \beta_j p_j(\mathbf{x}) \tag{7.7}$$

where p_j represents a monomial basis and M is the number of terms of monomial basis, β_j is the coefficient to be determined by collocation. The monomial basis in two-dimensional cases consists of the family

$$\{p\} = \left\{1, \quad x_1, \quad x_2, \quad x_1^2, \quad x_2^2, \quad x_1 x_2, \quad x_1^3, \quad \ldots\right\} \tag{7.8}$$

Once the interpolation function is chosen, Eq. (7.7) is collocated at N properly chosen interior and boundary nodes $\mathbf{x}_i$ such that

$$f(\mathbf{x}_i) = \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}_i) + \sum_{j=1}^{M} \beta_j p_j(\mathbf{x}_i) \tag{7.9}$$

To guarantee the solvability of the system, M constraint equations are needed as

$$\sum_{j=1}^{M} \alpha_j p_k(\mathbf{x}_j) = 0 \qquad \text{for} \quad k = 1, 2, \cdots, M \tag{7.10}$$

The collocation system consisting of Eqs. (7.9) and (7.10) also can be written as in matrix form

$$\begin{bmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_N(\mathbf{x}_1) & p_1(\mathbf{x}_1) & p_2(\mathbf{x}_1) & \cdots & p_M(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_N(\mathbf{x}_2) & p_1(\mathbf{x}_2) & p_2(\mathbf{x}_2) & \cdots & p_M(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \cdots & \phi_N(\mathbf{x}_N) & p_1(\mathbf{x}_N) & p_2(\mathbf{x}_N) & \cdots & p_M(\mathbf{x}_N) \\ p_1(\mathbf{x}_1) & p_1(\mathbf{x}_2) & \cdots & p_1(\mathbf{x}_N) & 0 & 0 & \cdots & 0 \\ p_2(\mathbf{x}_1) & p_2(\mathbf{x}_2) & \cdots & p_2(\mathbf{x}_N) & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_M(\mathbf{x}_1) & p_M(\mathbf{x}_2) & \cdots & p_M(\mathbf{x}_N) & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_M \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{7.11}$$

7.2.3 Stability and convergence of RBF approximation

To demonstrate the accuracy, stability and convergence of GS-RBF interpolation here, several numerical tests in $\mathbb{R}^2$ space are performed.

The following function

$$\begin{aligned} f(x_1,x_2) = & \frac{-751\pi^2}{144}\sin\frac{\pi x_1}{6}\sin\frac{7\pi x_1}{4}\sin\frac{3\pi x_2}{4}\sin\frac{5\pi x_2}{4} \\ & + \frac{7\pi^2}{12}\cos\frac{\pi x_1}{6}\cos\frac{7\pi x_1}{4}\sin\frac{3\pi x_2}{4}\sin\frac{5\pi x_2}{4} \\ & + \frac{15\pi^2}{8}\sin\frac{\pi x_1}{6}\sin\frac{7\pi x_1}{4}\cos\frac{3\pi x_2}{4}\cos\frac{5\pi x_2}{4} \end{aligned} \tag{7.12}$$

is considered a standard test, having a distribution in a unit square domain as shown in Figure 7.2.

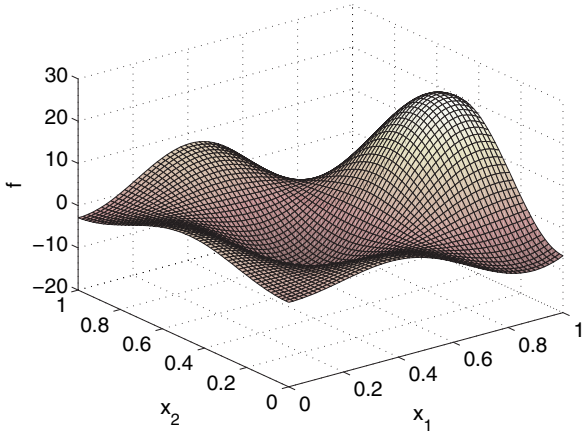


FIGURE 7.2
Distribution of test function f in the unit square domain

In our test, seven globally supported RBFs (GS-RBF) are employed (see Table 7.2) and different numbers of regularly distributed interpolation points are chosen to investigate the convergence of GS-RBF approximation for the given smooth function. The interpolation scheme without augmented terms is investigated first and the numerical results on average relative error (*Arerr*) defined as

$$Arerr(f) = \sqrt{\frac{\sum_{j=1}^{L}\left(f_j^{numerical} - f_j^{exact}\right)^2}{\sum_{j=1}^{L}\left(f_j^{exact}\right)^2}} \tag{7.13}$$

at 101×101 regular computing points and condition number are shown in Figure 7.3 where it can be seen that RBF1 and RBF2 (PS) and RBF3 and RBF4 (TPS) have a lower convergence rate than that of RBF5 and RBF6 (MQ). However, the selection of shape parameter c in MQ may largely affect the stability and convergence of the numerical results. Moreover, GS shows the worst results of all the RBFs used. The variations of condition number also show that PS-, TPS- and MQ-RBFs with small shape parameter have good stability as the number of interpolation point increases. In addition, the results in Figure 7.3 also indicate that highers order in PS- and TPS-RBFs can achieve better accuracy than lower order RBFs, but the related condition number becomes larger. It is also evident that TPS shows small improvement over PS-RBFs.

TABLE 7.2
Seven globally supported RBFs (GS-RBFs) for numerical test

Type	Expression
RBF1	r
RBF2	r^3
RBF3	$r^2 \ln r$
RBF4	$r^4 \ln r$
RBF5	$\sqrt{r^2+c^2}$ with $c = 0.5$
RBF6	$\sqrt{r^2+c^2}$ with $c = 1.0$
RBF7	e^{-r^2}

As reported in [7, 19], MQ converges exponentially, whereas TPS and PS converge at the rate of $O(h|\log h|)$ and $O\left(h^{1/2}\right)$, respectively, in the data spacing $X \equiv \{\mathbf{P}_j\}$ and $h = \max_{\mathbf{Q}\in R^n} \min_{\mathbf{P}\in X} \|\mathbf{P}-\mathbf{Q}\|$. However, for MQ interpolation, the shape parameter must be carefully chosen because over a small range (typically $0 < c < 10$) the accuracy of MQ interpolation can vary by three orders of magnitude, which means that the ratio of minimum and maximum of the average relative error could be as large as 10^3. For that reason TPS- and PS-RBFs are widely used in practical application, rather than MQ-RBFs.

Figure 7.4 shows comparisons between the results from the second interpolation strategy with augmented monomial basis and the first scheme described above when the PS and TPS bases are employed. It can be seen from Figure 7.4 that RBF with augmented monomial basis ($M > 0$) and pure RBF ($M = 0$) can produce similar

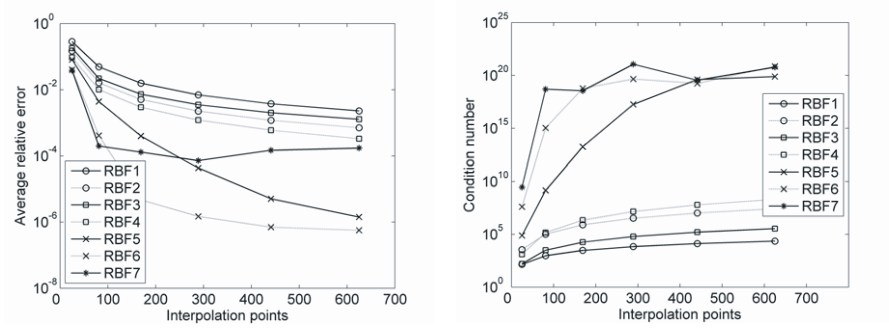


FIGURE 7.3
Variation of *Arerr* and condition numbers with the number of interpolation point

numerical results. Therefore only RBFs without augmented monomial basis are employed in the following computation.

7.3 Non-homogeneous problems

Non-homogeneous problems refer mainly to problems in which the corresponding governing equations have nonzero right-hand side terms, which gives rise to inconvenience in applying the HT-FE model because the kernel interpolation functions in the intra-element field, that is, Trefftz functions, are usually defined according to homogeneous governing equations, as seen in Chapters 5 and 6.

In this section, basic equations of non-homogeneous Poisson's equations and plane stress/strain problems are presented to provide notation and reference in later sections of this chapter.

7.3.1 Basic equations for Poisson's problems

The governing equation of Poisson's problems in a well-defined domain can be written as

$$\nabla^2 u = b(\mathbf{x}) \tag{7.14}$$

where u is the scalar potential field sought, and b represents the known source function distribution in terms of spatial variables $\mathbf{x}$ in the domain.

The boundary conditions of Poisson's equation (7.14) have the same form as given in Chapter 5, i.e.,

$$\begin{aligned} u &= \bar{u} \quad \text{on } \Gamma_u \\ q &= \frac{\partial u}{\partial n} = \bar{q} \quad \text{on } \Gamma_q \end{aligned} \tag{7.15}$$

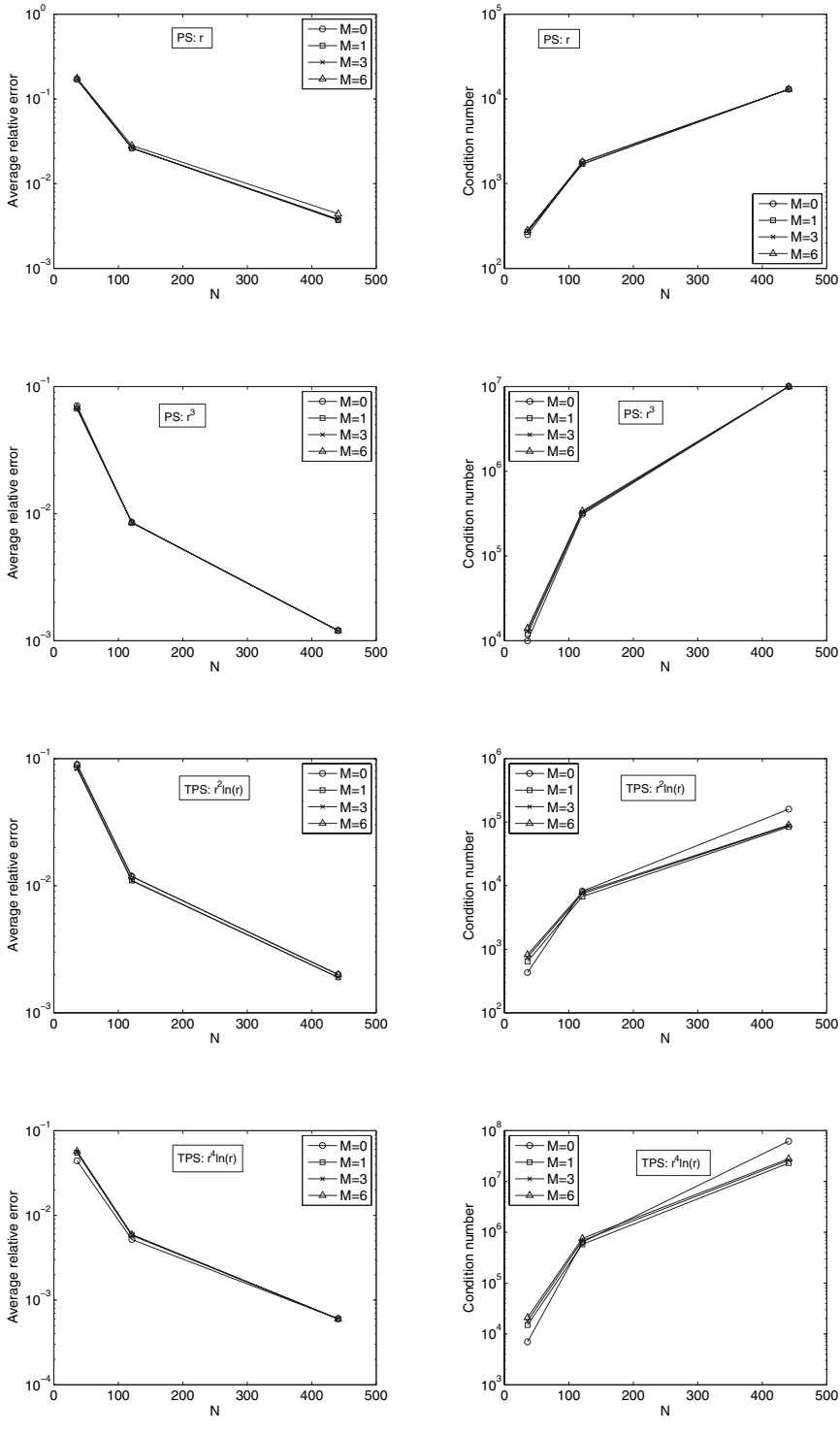


FIGURE 7.4
Variation of *Arerr* and condition numbers with the number of interpolation points

and are added to keep the system complete.

7.3.2 Basic equations for plane stress/strain problems

Consider again the isotropic homogeneous material described in Chapter 6. The equilibrium governing and geometric equations are defined by Eqs. (6.1) and (6.3), i.e.,

$$\mathbf{L}\sigma + \mathbf{b} = \mathbf{0} \tag{7.16}$$

$$\varepsilon = \mathbf{L}^{\mathrm{T}}\mathbf{u} \tag{7.17}$$

The constitutive relation (6.4) is modified to include thermal response:

$$\sigma = \mathbf{D}\varepsilon - \mathbf{m}\vartheta \tag{7.18}$$

where $\mathbf{L}$, σ, ε, and $\mathbf{D}$ are defined in Chapter 6, ϑ represents the temperature change in the domain, and $\mathbf{m}$ denotes effective coefficient vector of temperature change to stress distribution and is expressed as

$$\mathbf{m} = \tilde{m}\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \tag{7.19}$$

with

$$\begin{cases} \tilde{m} = \dfrac{E}{1-2\nu}\alpha & \text{for plane strain} \\ \tilde{m} = \dfrac{E}{1-\nu}\alpha & \text{for plane stress} \end{cases} \tag{7.20}$$

where E, ν and α represents Young's modulus, Poisson's ratio and coefficient of thermal expansion, respectively.

Combining Eqs. (7.16), (7.17) and (7.18) gives the following Navier equations in terms of displacements:

$$\mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{u} + \tilde{\mathbf{b}} = 0 \tag{7.21}$$

in which

$$\tilde{\mathbf{b}} = \mathbf{b} - \mathbf{L}\mathbf{m}\vartheta \tag{7.22}$$

represents the generalised body forces induced by both body forces $\mathbf{b}$ and temperature change ϑ.

For simplicity in the following writing, the so-called Cartesian tensor notation is employed throughout this chapter. This notation is not only a time-saver in writing long expressions, but also extremely useful in deriving formulation. Such notation makes use of number subscript indices $1, 2, 3, \ldots$ to represent components. Repeated subscript indices imply summation operation. For example, for two dimensional problems we have,

$$a_{k,k} = a_{1,1} + a_{2,2} \tag{7.23}$$

where a comma followed by an index represents space differentiation.

Using the index notation, Eqs. (7.16) - (7.18) can be written as

$$\sigma_{ij,j} + b_i = 0 \tag{7.24}$$

$$\sigma_{ij} = \tilde{\lambda}\,\delta_{ij}\varepsilon_{kk} + 2G\varepsilon_{ij} - \tilde{m}\delta_{ij}\vartheta \tag{7.25}$$

$$\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \tag{7.26}$$

where $\tilde{\lambda}$ and G are defined in Eq. (6.7).

Combining the above equations produces the following Navier equations in tensor form

$$Gu_{i,jj} + \frac{G}{1-2\tilde{\nu}}u_{j,ji} = -\tilde{b}_i \tag{7.27}$$

where $\tilde{b}_i = b_i - \tilde{m}\vartheta_{,i}$.

The corresponding boundary conditions (6.8) are now rewritten in index form as

$$\begin{aligned} &u_i = \bar{u}_i \qquad \text{on } \Gamma_u \\ &t_i = \sigma_{ij}n_j = \bar{t}_i \qquad \text{on } \Gamma_t \end{aligned} \tag{7.28}$$

7.4 Solution procedure of HT-FEM for non-homogeneous problems

To provide a background and thus enhance our understanding of the procedure described in the following sections, we briefly review the major processes of HT-FEM. For simplicity we take Poisson's problems as an example in this section.

7.4.1 Assumed fields

Unlike in the homogeneous potential problem, the particular solution u^p must be considered for constructing the intra-element field in Poisson's problem, that is,

$$u_e(\mathbf{x}) = u^p + \sum_{j=1}^{m} N_j(\mathbf{x})\,c_{ej} = u^p + \mathbf{N}_e(\mathbf{x})\,\mathbf{c}_e \tag{7.29}$$

The corresponding boundary flux is

$$q_e = \frac{\partial u_e}{\partial n} = q_e^p + \mathbf{Q}_e\mathbf{c}_e \tag{7.30}$$

where

$$q_e^p = \frac{\partial u^p}{\partial n}, \quad Q_{ej} = \frac{\partial N_{ej}}{\partial n} \tag{7.31}$$

Besides the intra-element field, the frame field is assumed on the element boundary

$$\tilde{u}_e(\mathbf{x}) = \tilde{\mathbf{N}}_e(\mathbf{x})\,\mathbf{d}_e \tag{7.32}$$

7.4.2 Variational functional

For a particular element, the functional (5.21) is rewritten as

$$\Psi_{me} = \frac{1}{2}\int_{\Omega_e}\left(q_1^2+q_2^2\right)t_e\mathrm{d}\Omega - \int_{\Gamma_e} q\tilde{u}t_e\mathrm{d}\Gamma + \int_{\Gamma_{eq}} \bar{q}\tilde{u}t_e\mathrm{d}\Gamma \tag{7.33}$$

Integrating the first term on the right-hand side of Eq. (7.33) by parts gives

$$\begin{aligned}\int_{\Omega_e}\left(q_1^2+q_2^2\right)t_e\mathrm{d}\Omega &= \int_{\Gamma_e} u\frac{\partial u}{\partial n}t_e\mathrm{d}\Gamma - \int_{\Omega_e}\left(\nabla^2 u\right)ut_e\mathrm{d}\Omega \\ &= \int_{\Gamma_e} uqt_e\mathrm{d}\Gamma - \int_{\Omega_e} but_e\mathrm{d}\Omega\end{aligned} \tag{7.34}$$

As a result, the functional (7.33) becomes

$$\Psi_{me} = \frac{1}{2}\int_{\Gamma_e} uqt_e\mathrm{d}\Gamma - \frac{1}{2}\int_{\Omega_e} but_e\mathrm{d}\Omega - \int_{\Gamma_e} q\tilde{u}t_e\mathrm{d}\Gamma + \int_{\Gamma_{eq}} \bar{q}\tilde{u}t_e\mathrm{d}\Gamma \tag{7.35}$$

Substituting Eqs. (7.29), (7.30) and (7.32) into (7.35) produces

$$\Psi_{me} = \frac{1}{2}\mathbf{c}_e^{\mathrm{T}}\mathbf{H}_e\mathbf{c}_e - \mathbf{c}_e^{\mathrm{T}}\mathbf{G}_e\mathbf{d}_e + \mathbf{d}_e^{\mathrm{T}}\mathbf{g}_e + \mathbf{c}_e^{\mathrm{T}}\mathbf{h}_e + \text{terms without } \mathbf{c}_e \text{ or } \mathbf{d}_e \tag{7.36}$$

where

$$\mathbf{H}_e = \int_{\Gamma_e}\mathbf{Q}_e^{\mathrm{T}}\mathbf{N}_e t_e\mathrm{d}\Gamma \tag{7.37}$$

$$\mathbf{G}_e = \int_{\Gamma_e}\mathbf{Q}_e^{\mathrm{T}}\tilde{\mathbf{N}}_e t_e\mathrm{d}\Gamma \tag{7.38}$$

$$\mathbf{g}_e = \int_{\Gamma_{eq}}\tilde{\mathbf{N}}_e^{\mathrm{T}}\bar{q}t_e\mathrm{d}\Gamma - \int_{\Gamma_e}\tilde{\mathbf{N}}_e^{\mathrm{T}}q^p t_e\mathrm{d}\Gamma \tag{7.39}$$

$$\mathbf{h}_e = \frac{1}{2}\int_{\Gamma_e}\mathbf{Q}_e^{\mathrm{T}}u^p t_e\mathrm{d}\Gamma + \frac{1}{2}\int_{\Gamma_e}\mathbf{N}_e^{\mathrm{T}}q^p t_e\mathrm{d}\Gamma - \frac{1}{2}\int_{\Omega_e}\mathbf{N}_e^{\mathrm{T}}bt_e\mathrm{d}\Omega \tag{7.40}$$

To enforce inter-element continuity on the common element boundary, the unknown vector $\mathbf{c}_e$ should be expressed in terms of nodal DOF $\mathbf{d}_e$. The minimization of the functional Ψ_{me} yields

$$\frac{\partial\Psi_{me}}{\partial\mathbf{c}_e^{\mathrm{T}}} = \mathbf{H}_e\mathbf{c}_e - \mathbf{G}_e\mathbf{d}_e + \mathbf{h}_e = \mathbf{0} \tag{7.41}$$

$$\frac{\partial\Psi_{me}}{\partial\mathbf{d}_e^{\mathrm{T}}} = -\mathbf{G}_e^{\mathrm{T}}\mathbf{c}_e + \mathbf{g}_e = \mathbf{0} \tag{7.42}$$

Eq. (7.41) determines the optional relationship between $\mathbf{c}_e$ and $\mathbf{d}_e$, that is

$$\mathbf{c}_e = \mathbf{H}_e^{-1}\left(\mathbf{G}_e\mathbf{d}_e - \mathbf{h}_e\right) \tag{7.43}$$

Substitution of Eq. (7.43) into Eq. (7.42) finally yields the following element stiffness equation

$$\mathbf{K}_e\mathbf{d}_e = \mathbf{p}_e \tag{7.44}$$

where

$$\mathbf{K}_e = \mathbf{G}_e^{\mathrm{T}} \mathbf{H}_e^{-1} \mathbf{G}_e \tag{7.45}$$

$$\mathbf{p}_e = \mathbf{g}_e + \mathbf{G}_e^{\mathrm{T}} \mathbf{H}_e^{-1} \mathbf{h}_e \tag{7.46}$$

stand for the element stiffness matrix and the equivalent nodal load vector, respectively.

7.4.3 Discussion

From the above review we can see that the existence of the non-zero right-hand side term b results in domain integral computation which lessens the advantage of HT-FEM. On the other hand, determination of the distribution of the particular solution u^p in Eq. (7.29) is sometimes complex. However, u^p can be obtained analytically for some simple right-hand non-homogeneous source functions b. For example, when the right-hand term b is a linear function

$$b = b_0 + x_1 b_1 + x_2 b_2 \tag{7.47}$$

we have

$$u^p = \frac{1}{4} b_0 \left(x_1^2 + x_2^2\right) + \frac{1}{6} \left(b_1 x_1^3 + b_2 x_2^3\right) + k_0 + k_1 x_1 + k_2 x_2 \tag{7.48}$$

where the linear part $k_0 + k_1 x_1 + k_2 x_2$ is arbitrary. From Eq. (7.48) we can see that the particular solution is not unique. If b is a complicated function of coordinates, it is difficult to obtain the corresponding particular solution. In hybrid Trefftz FEM (HT-FEM), the required particular solution is evaluated numerically by means of the singular integral in terms of the fundamental solution [1] (see also Chapter 1). In this case, it is necessary to perform the domain integrals for establishing the FE equation, which complicates the entire procedure.

In this chapter, a method based on RBF interpolation is introduced to determine the required particular solution numerically, which is then used to modify the boundary conditions. As a result, the domain integrals appearing in the HT-FEM are avoided completely. The detailed derivation is discussed in the following section.

7.5 Particular solutions in terms of RBFs

In this section, RBFs are employed to approximate the inhomogeneous terms appearing in governing equations, such as the source functions b in the potential equation (7.14) and generalised body force terms $\tilde{\mathbf{b}}$ in the plane stress/strain governing equations (7.21). The corresponding particular solutions are also derived by analytical integral procedure and RBF approximation.

7.5.1 Particular solutions for Poisson's equation

Due to the linear feature of the Laplace operator, the solution to Eq. (7.14) can be divided into two parts:

$$u(\mathbf{x}) = u^h(\mathbf{x}) + u^p(\mathbf{x}) \tag{7.49}$$

where u^h is the homogeneous solution satisfying

$$\nabla^2 u^h(\mathbf{x}) = 0 \tag{7.50}$$

which is subjected to a modified boundary conditions, and u^p stands for the particular solution satisfying

$$\nabla^2 u^p(\mathbf{x}) = b(\mathbf{x}) \tag{7.51}$$

without any restriction of boundary condition.

To determine the particular solution u^p, the right-hand side term $b(\mathbf{x})$ in Eq. (7.51) is approximated in terms of RBFs in the form

$$b(\mathbf{x}) \approx \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}) \tag{7.52}$$

where N is the number of reference points in the domain, ϕ_j represents a RBF centered at reference point $\mathbf{x}_j$, and α_j denotes an unknown coefficient.

The particular solution sought can also be written in a similar form

$$u^p(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \Phi_j(\mathbf{x}) \tag{7.53}$$

in which

$$\nabla^2 \Phi_j(\mathbf{x}) = \phi_j(\mathbf{x}) \tag{7.54}$$

in the solution domain, where $\Phi_j(\mathbf{x}) = \Phi(\mathbf{x}, \mathbf{x}_j)$ represents the approximated particular solution kernel.

In the following we focus on constructing Φ_j using the specified RBF ϕ_j.

Considering that RBFs are related to the Euclidean distance r only, we rewrite the Laplace operator in terms of the polar coordinates system as

$$\nabla^2 = \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right) \tag{7.55}$$

Substituting Eq. (7.55) into Eq. (7.54) and integrating it twice analytically, the corresponding approximate particular solution Φ_j can be determined as

$$\Phi_j = \frac{r^{2n+1}}{(2n+1)^2} \tag{7.56}$$

for a PS-type RBF in $\mathbb{R}^2$, that is, $\phi_j = r^{2n-1} \quad (n = 1,\ 2,\ 3,\ \ldots)$, and

$$\begin{aligned} \frac{\partial \Phi_j}{\partial x_1} &= \frac{r^{2n-1}}{(2n+1)}\left(x_1 - x_{1j}\right) \\ \frac{\partial \Phi_j}{\partial x_2} &= \frac{r^{2n-1}}{(2n+1)}\left(x_2 - x_{2j}\right) \end{aligned} \tag{7.57}$$

for TPS-type RBF in $\mathbb{R}^2$, that is, $\phi_j = r^{2n}\ln r\ (n = 1,\ 2,\ 3,\ \ldots)$, Φ_j has the following form

$$\Phi_j = \frac{(n+1)\ln r - 1}{4(n+1)^3} r^{2+2n} \tag{7.58}$$

and

$$\begin{aligned} \frac{\partial \Phi_j}{\partial x_1} &= \frac{2(n+1)\ln r - 1}{4(n+1)^2} r^{2n}\left(x_1 - x_{1j}\right) \\ \frac{\partial \Phi_j}{\partial x_2} &= \frac{2(n+1)\ln r - 1}{4(n+1)^2} r^{2n}\left(x_2 - x_{2j}\right) \end{aligned} \tag{7.59}$$

Using the solution obtained and setting the field point $\mathbf{x}$ in Eq. (7.52) to be N reference points in turn, we can produce linear algebraic equations (7.5), which can be solved for all unknowns, and then the particular solution at arbitrary field point $\mathbf{x}$ can be determined using Eq. (7.53).

Once the particular solution distribution in the domain is determined, the new homogeneous system

$$\nabla^2 u^h(\mathbf{x}) = 0 \tag{7.60}$$

with modified boundary conditions

$$\begin{aligned} u^h &= \bar{u} - u^p \quad \text{on } \Gamma_u \\ q^h &= \bar{q} - q^p \quad \text{on } \Gamma_q \end{aligned} \tag{7.61}$$

where

$$q^p = \frac{\partial u^p}{\partial n} = \sum_{j=1}^{N} \alpha_j \frac{\partial \Phi_j(\mathbf{x})}{\partial n} \tag{7.62}$$

can be solved using HT-FE formulation as presented in Chapter 5.

7.5.2 Particular solutions for plane stress/strain equations

Considering the plane elastic problem governed by Eq. (7.21), its solution $\mathbf{u}$ can be decomposed into two major parts: the homogeneous and particular parts, as was done for the potential problem:

$$\mathbf{u} = \mathbf{u}^h + \mathbf{u}^p \tag{7.63}$$

where

$$\mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{u}^h = 0 \tag{7.64}$$

and

$$\mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{u}^{p}+\tilde{\mathbf{b}}=0 \tag{7.65}$$

Regarding thermal loading, the treatment in Section 7.3 is introduced to convert the thermal load into an equivalent body force (see Eq. (7.22)). In this subsection, an alternative treatment is introduced in addition to the equivalent body force. Figure 7.5 shows the major steps of the procedure, in which the entire system variables including strain and stress fields are divided into a homogeneous part and a particular part. With the procedure, the boundary conditions must be changed as

$$\begin{aligned} u_i^h &= \bar{u}_i - u_i^p \qquad \text{on } \Gamma_u \\ t_i^h &= \bar{t}_i - t_i^p + \tilde{m}\vartheta n_i \qquad \text{on } \Gamma_t \end{aligned} \tag{7.66}$$

The homogeneous plane elastic problem defined by Eqs. (7.64) and (7.66) can be solved by the HT-FEM presented in Chapter 6.

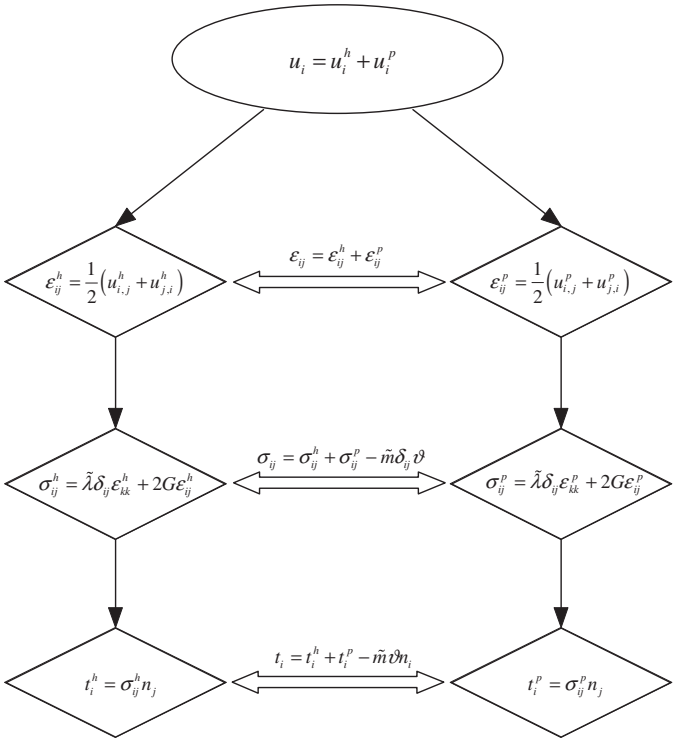


FIGURE 7.5
Decomposition of thermal-elastic system

Having determined the homogeneous solution $\mathbf{u}^h$, the next step is to determine the

particular solution $\mathbf{u}^p$ governed by Eq. (7.65). To this end, the right-hand generalised body forces $\tilde{\mathbf{b}}(\mathbf{x})$ in Eq. (7.65) are firstly approximated as

$$\tilde{b}_i(\mathbf{x}) \approx \sum_{n=1}^{N} \alpha_i^n \phi_n(\mathbf{x}) = \sum_{n=1}^{N} \alpha_l^n \delta_{li} \phi_n(\mathbf{x}) \qquad \mathbf{x} \in \Omega \tag{7.67}$$

where N is the number of reference points to be selected, ϕ_n represents the RBF with reference point $\mathbf{x}$, α_i^n denotes the unknown coefficient, and δ_{li} is the Kronecker delta function.

Setting the space variable $\mathbf{x}$ to be N interpolation points, $\mathbf{x}_i$ in turn produces

$$\tilde{b}_i(\mathbf{x}_i) \approx \sum_{n=1}^{N} \alpha_i^n \phi_n(\mathbf{x}_i) \tag{7.68}$$

or in matrix form

$$\begin{bmatrix} \phi_1(\mathbf{x}_1) & 0 & \phi_2(\mathbf{x}_1) & 0 & \cdots & \phi_N(\mathbf{x}_1) & 0 \\ 0 & \phi_1(\mathbf{x}_1) & 0 & \phi_2(\mathbf{x}_1) & \cdots & 0 & \phi_N(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & 0 & \phi_2(\mathbf{x}_2) & 0 & \cdots & \phi_N(\mathbf{x}_2) & 0 \\ 0 & \phi_1(\mathbf{x}_2) & 0 & \phi_2(\mathbf{x}_2) & \cdots & 0 & \phi_N(\mathbf{x}_2) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \phi_1(\mathbf{x}_N) & 0 & \phi_2(\mathbf{x}_N) & 0 & \cdots & \phi_N(\mathbf{x}_N) & 0 \\ 0 & \phi_1(\mathbf{x}_N) & 0 & \phi_2(\mathbf{x}_N) & \cdots & 0 & \phi_N(\mathbf{x}_N) \end{bmatrix} \begin{Bmatrix} \alpha_1^1 \\ \alpha_2^1 \\ \alpha_1^2 \\ \alpha_2^2 \\ \vdots \\ \alpha_1^N \\ \alpha_2^N \end{Bmatrix} = \begin{Bmatrix} b_1(\mathbf{x}_1) \\ b_2(\mathbf{x}_1) \\ b_1(\mathbf{x}_2) \\ b_2(\mathbf{x}_2) \\ \vdots \\ b_1(\mathbf{x}_N) \\ b_2(\mathbf{x}_N) \end{Bmatrix} \tag{7.69}$$

which is used to determine all unknown interpolation coefficients α_i^n.

Eq (7.53) for the particular $\mathbf{u}^p$ is now rewritten as

$$u_i^p(\mathbf{x}) = \sum_{n=1}^{N} \alpha_l^n \Phi_{li}^n(\mathbf{x}) \qquad \mathbf{x} \in \Omega \tag{7.70}$$

or in matrix form

$$\begin{Bmatrix} u_1^p(\mathbf{x}) \\ u_2^p(\mathbf{x}) \end{Bmatrix} = \begin{bmatrix} \Phi_{11}^1 & \Phi_{21}^1 & \Phi_{11}^2 & \Phi_{21}^2 & \cdots & \Phi_{11}^N & \Phi_{21}^N \\ \Phi_{12}^1 & \Phi_{22}^1 & \Phi_{12}^2 & \Phi_{22}^2 & \cdots & \Phi_{12}^N & \Phi_{22}^N \end{bmatrix} \begin{Bmatrix} \alpha_1^1 \\ \alpha_2^1 \\ \alpha_1^2 \\ \alpha_2^2 \\ \vdots \\ \alpha_1^N \\ \alpha_2^N \end{Bmatrix} \tag{7.71}$$

where $\Phi_{li}^n(\mathbf{x})$ is the approximated particular solution kernel of displacement.

Substitution of Eqs. (7.67) and (7.70) into Eq. (7.27) yields the following relation:

$$G\Phi_{li,kk}^n + \frac{G}{1-2\tilde{\nu}} \Phi_{lk,ki}^n = -\delta_{li}\phi_n \tag{7.72}$$

which is used to determine $\Phi_{li}^n(\mathbf{x})$ for any specified RBF ϕ_n. In doing this, the displacement particular solution kernel $\Phi_{li}^n(\mathbf{x})$ is expressed in terms of the Galerkin-Papkovich vectors [20]

$$\Phi_{li}^n = \frac{1-\tilde{\nu}}{G} F_{li,mm} - \frac{1}{2G} F_{mi,ml} \tag{7.73}$$

Substituting Eq. (7.73) into Eq. (7.72) yields the following bi-harmonic equation:

$$\begin{aligned}
&\frac{G}{1-2\tilde{\nu}}\Phi_{kl,ki}^n + G\Phi_{li,kk}^n \\
&= \frac{1-\tilde{\nu}}{1-2\tilde{\nu}} F_{ki,mmkl} - \frac{1}{2(1-2\tilde{\nu})} F_{mi,mkkl} + (1-\tilde{\nu}) F_{li,mmkk} - \frac{1}{2} F_{mi,mlkk} \\
&= \frac{1-\tilde{\nu}}{1-2\tilde{\nu}} F_{ki,mmkl} - \frac{1-\tilde{\nu}}{(1-2\tilde{\nu})} F_{mi,mkkl} + (1-\tilde{\nu}) F_{li,mmkk} \\
&= (1-\tilde{\nu}) F_{li,mmkk} \\
&= (1-\tilde{\nu}) \nabla^4 F_{li}
\end{aligned} \tag{7.74}$$

in which the relationship $F_{ki,mmkl} = F_{mi,kkml} = F_{mi,mkkl} = F_{mi,mlkk}$ has been used. Substituting Eq. (7.74) into Eq. (7.72) we have

$$\nabla^4 F_{li} = F_{li,mmkk} = -\frac{1}{1-\tilde{\nu}} \delta_{li} \phi_n \tag{7.75}$$

It is noted that the bi-harmonic operator ∇^4 has a simpler form in the axisymmetric system

$$\nabla^4 = \frac{\partial^4}{\partial x_1^4} + 2\frac{\partial^4}{\partial x_1^2 \partial x_2^2} + \frac{\partial^4}{\partial x_2^4} = \frac{1}{r}\frac{\mathrm{d}}{\mathrm{d}r}\left(r\frac{\mathrm{d}}{\mathrm{d}r}\left(\frac{1}{r}\frac{\mathrm{d}}{\mathrm{d}r}\left(r\frac{\mathrm{d}}{\mathrm{d}r}\right)\right)\right) \tag{7.76}$$

Making use of Eq. (7.76) and integrating Eq. (7.75) by parts, the solution F_{li} and in turn the displacement function Φ_{li}^n can be determined.

The corresponding stress particular solution is obtained by substituting Eq. (7.70) into Eq. (7.25), i.e.,

$$\sigma_{ij}^p = \tilde{\lambda}\delta_{ij}\varepsilon_{kk}^p + 2G\varepsilon_{ij}^p = \sum_{n=1}^{N} \alpha_l^n S_{lij}^n \tag{7.77}$$

where

$$S_{lij}^n = \left[\tilde{\lambda}\delta_{ij}\Phi_{lk,k}^n + G\left(\Phi_{li,j}^n + \Phi_{lj,i}^n\right)\right] \tag{7.78}$$

Using the relation (7.28), the traction of the particular solution can be obtained as

$$t_i^p = \sigma_{ij}^p n_j = \sum_{n=1}^{N} \alpha_l^n S_{lij}^n n_j \tag{7.79}$$

For the classic PS basis function, that is, $\phi_j = r^{2n-1} \quad (n = 1,\ 2,\ 3,\ \ldots)$ in R^2, we have†

$$F_{li} = -\frac{\delta_{li}}{1-\tilde{\nu}} \frac{r^{2n+3}}{(2n+1)^2 (2n+3)^2} \tag{7.80}$$

$$\Phi_{li}^k = -\frac{1}{2G(1-\tilde{\nu})} \frac{r^{2n+1}}{(2n+1)^2 (2n+3)} \left(A_1 \delta_{il} + A_2 r_{,i} r_{,l}\right) \tag{7.81}$$

$$S_{lij}^k = -\frac{1}{(1-\tilde{\nu})} \frac{r^{2n}}{(2n+1)(2n+3)} \begin{bmatrix} A_3 \delta_{ij} r_{,l} + A_4 r_{,i} r_{,j} r_{,l} \\ +A_5 \left(\delta_{il} r_{,j} + \delta_{lj} r_{,i}\right) \end{bmatrix} \tag{7.82}$$

where

$$\begin{aligned} A_1 &= 5 + 4n - 2\tilde{\nu}(2n+3) \\ A_2 &= -(2n+1) \\ A_3 &= \tilde{\nu}(2n+3) - 1 \\ A_4 &= 1 - 2n \\ A_5 &= 2n + 2 - \tilde{\nu}(2n+3) \end{aligned} \tag{7.83}$$

and

$$r = \|\mathbf{x} - \mathbf{x}_k\| \tag{7.84}$$

For the classic TPS basis function, that is, $\phi_k = r^{2n} \ln r \ (n = 1,\ 2,\ 3,\ \ldots)$ in R^2, we have

$$F_{li} = -\frac{\delta_{li}}{1-\tilde{\nu}} \frac{r^{2n+4}}{16(n+1)^2 (n+2)^2} \left[\ln r - \frac{2n+3}{(n+1)(n+2)}\right] \tag{7.85}$$

$$\Phi_{li}^k = -\frac{1}{32G(1-\tilde{\nu})} \frac{r^{2n+2}}{(n+1)^3 (n+2)^2} \left(B_1 \delta_{il} + B_2 r_{,i} r_{,l}\right) \tag{7.86}$$

$$S_{lij}^k = -\frac{1}{8(1-\nu)} \frac{r^{2n+1}}{(n+1)^2 (n+2)^2} \begin{bmatrix} B_3 \delta_{ij} r_{,l} + B_4 r_{,i} r_{,j} r_{,l} \\ +B_5 \left(\delta_{il} r_{,j} + \delta_{lj} r_{,i}\right) \end{bmatrix} \tag{7.87}$$

with

$$\begin{aligned} B_1 &= -\left(8n^2 + 29n + 27\right) + 8\tilde{\nu}(n+2)^2 + 2(n+1)(n+2)\left[4n + 7 - 4\tilde{\nu}(n+2)\right] \ln r \\ B_2 &= 2(n+1)(2n+3) - 4(n+1)^2 (n+2) \ln r \\ B_3 &= 2n + 3 - 2\tilde{\nu}(n+2)^2 + 2(n+1)(n+2)(2\tilde{\nu}n + 4\tilde{\nu} - 1) \ln r \\ B_4 &= 2\left(n^2 - 2\right) - 4n(n+1)(n+2) \ln r \\ B_5 &= -\left(2n^2 + 6n + 5\right) + 2\tilde{\nu}(n+2)^2 - 2(n+1)(n+2)\left[-(2n+3) + 2\tilde{\nu}(n+2)\right] \ln r \end{aligned} \tag{7.88}$$

†The following relations have been used in the derivation: $r_{,i} = (x_i - x_{ik})/r$, $r_{,k} r_{,k} = 1$, $r_{,ij} = (\delta_{ij} - r_{,i} r_{,j})/r$

Having determined the particular and homogeneous solutions for displacement and stress components at specified points, the full displacement and stress fields can be calculated by

$$u_i = u_i^p + u_i^h \tag{7.89}$$

and

$$\sigma_{ij} = \sigma_{ij}^p + \sigma_{ij}^h - \tilde{m}\delta_{ij}\vartheta \tag{7.90}$$

It is worth pointing out that the modified term $\tilde{m}\delta_{ij}\vartheta$ in Eq. (7.90) should be included in the stresses in thermoelastic analysis. This relation is also shown in Figure 7.5.

7.6 Modification of the program structure

Because the internal source distribution in Poisson's problems and the generalised body forces involving body forces and thermal response are considered in this chapter, the program structure presented in Chapters 5 and 6 for homogeneous problems must be modified by adding special modules related to RBF interpolations and the treatment of inhomogeneous terms. In the previous sections of this chapter, the homogeneous system is obtained with modified boundary conditions by excluding the particular solution part. In the program presented here, the modification of boundary conditions is not necessarily performed separately before forming stiffness equations, because the modification of flux or traction on the boundary requires the directional cosine of outward normal to be known, which is not convenient for computation. Our strategy is that to modify the displacement and traction boundary conditions in the process of forming the equivalent nodal flux or load vector. As an example, we consider Poisson's problems. The modification of boundary conditions is performed in the modules for forming equivalent nodal flux and introducing specified potential conditions at nodes. The treatment is now described in detail.

7.6.1 Forming equivalent nodal flux

Since p_{ie} and $\bar{q}$ in Eq. (5.58) are now replaced by p_{ie}^h and $\bar{q}^h$, Eq. (5.58) should be modified as

$$p_{ie}^h = \int_{\Gamma_e} N_i \bar{q}^h \mathrm{d}\Gamma = \int_{\Gamma_e} N_i \left(\bar{q} - \bar{q}^p\right) \mathrm{d}\Gamma \tag{7.91}$$

Obviously, p_{ie}^h in Eq. (7.91) represents the equivalent nodal flux for the homogeneous system defined in (7.60) and (7.61).

Using Gaussian numerical integral method, Eq. (7.91) can be further written as

$$p_{ie}^h = \int_{-1}^{+1} N_i \left(\bar{q} - \bar{q}^p\right) J \mathrm{d}\xi \approx \sum_{k=1}^{n_s} w_k N_i \left(\xi_k\right) \left[\bar{q}\left(\xi_k\right) - \bar{q}^p\left(\xi_k\right)\right] J\left(\xi_k\right) \tag{7.92}$$

Eq. (7.92) shows that only the particular solutions at Gaussian points are required, instead of those at nodal points.

7.6.2 Introducing nodal potential condition

Having obtained the stiffness equations including equivalent loading vectors, removal of the singularity of the stiffness equation system should be done by introducing specified potential conditions at nodes. In contrast to the modification of flux condition, modification of potential condition is performed in a more direct way. For example, if the potential field at node i is constrained with $\bar{u}|_{\text{at node } i}$, then the corresponding homogeneous potential can be obtained by

$$\bar{u}^h\Big|_{\text{at node } i} = \bar{u}|_{\text{at node } i} - \bar{u}^p|_{\text{at node } i} \tag{7.93}$$

Finally, it should be mentioned that the full solution is obtained by combining the homogeneous solution and particular solution part.

The modification of boundary conditions in thermoelastic problems is similar to that described above. The entire solution procedure is illustrated in Figure 7.6 for reference.

7.7 MATLAB functions for particular solutions

In contrast to the procedures in Chapter 5 and 6, the existence of body forces increases the complication of computer programming. However, the approach employed in this chapter, that is, the separation of homogeneous solutions and particular solutions, enables us easily to add and modify related modules based on the routines listed in Chapter 5 and 6. For simplicity, only those routines, which are required to modify, are provided in this chapter.

In addition, in order to give the right-hand source field b or equivalent body forces $\mathbf{b}$ appeared in the governing equations, the subroutine `USERFUN` is introduced. What the user needs to do is to modify the related part in function `USERFUN` according to problems under consideration. The same function is also provided in C programming for this purpose.

7.7.1 Two-dimensional Poisson's problems

```
function MAINFUN
% Main program using HTFEM with RBF approximation for
% 2D Poisson's problems
% ******************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;
```

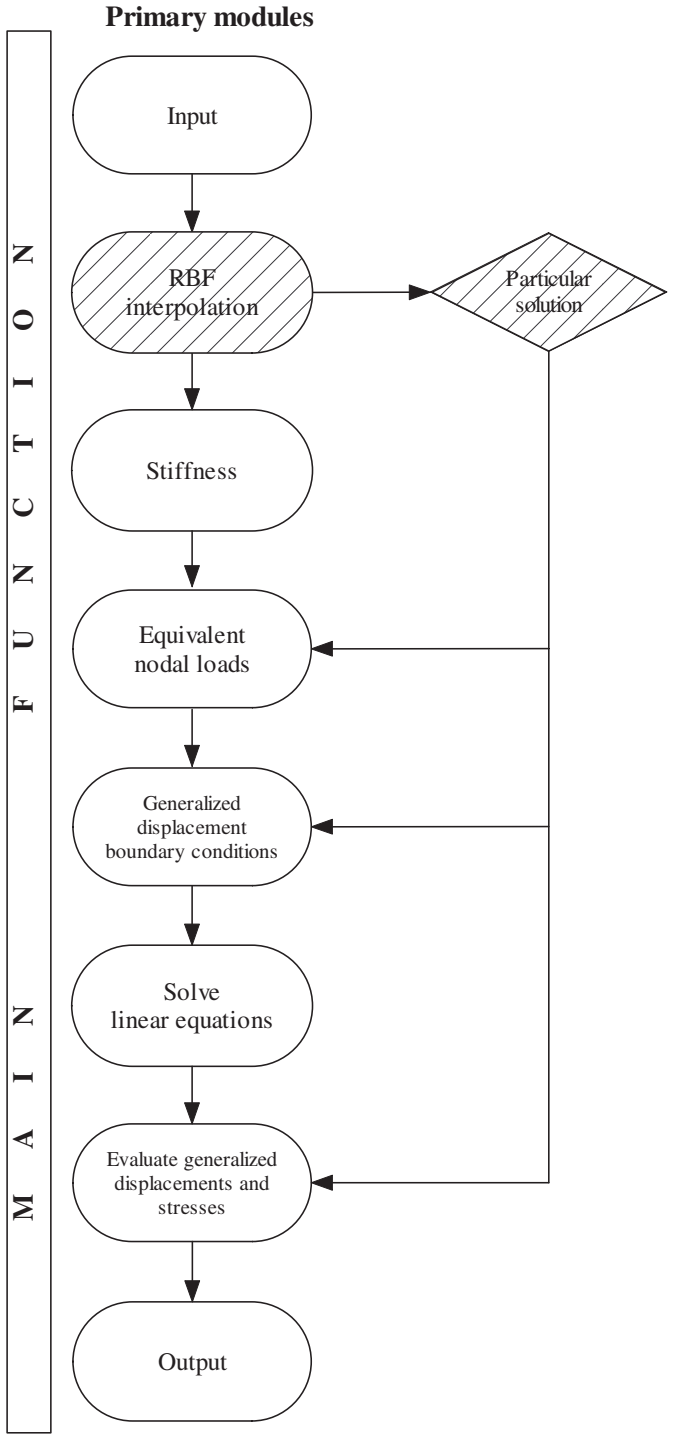


FIGURE 7.6
Illustration of entire solution procedure

```
disp('*******************************************');
disp('             Hybrid Trefftz FEM');
disp('          for 2D Poisson problems');
disp('*******************************************');

% Input data from file
[NPOIN,NELEM,COORD,MATNO,LNODS,NVFIX,NOFIX,IFPRE,...
    PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,NOPRS,PRESS,...
    PROPS]=INPUTDT;
% Generate local relations of nodes and edges
[ELNOD]=TYPELEM;
% Compute the coefficient of RBF interpolation
[NINTP,IPCOD,CRBFI]=RBFINTP(NPOIN,NELEM,COORD,LNODS);
% Global stiffness matrix
NEQNS=NPOIN*NDOFN;
GSTIF=zeros(NEQNS,NEQNS);
GLOAD=zeros(NEQNS,1);
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute element stiffnesss matrix
    [ESTIF]=KMATRIX(EHMTX,EGMTX);
    % Assemble stiffness matrix
    [GSTIF]=ASMSTIF(iELEM,LNODS,ESTIF,GSTIF);
end
% Compute equivalent nodal forces
[GLOAD]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,...
    NOPRS,PRESS,NPLOD,LODPT,POINT,GLOAD,NINTP,IPCOD,...
    CRBFI);
% Introduce constrained displacements and point loads
[GSTIF,GLOAD]=INDISBC(COORD,NEQNS,NVFIX,NOFIX,IFPRE,...
    PRESC,GSTIF,GLOAD,NINTP,IPCOD,CRBFI);
% Solve linear system of equations and store
% displacements of each node
% in the array ASDIS
[ASDIS]=LSSOLVR(GSTIF,GLOAD,NEQNS);
% Output nodal displacements
[UPOIN]=FIEDNOD(NPOIN,COORD,ASDIS,NINTP,IPCOD,CRBFI);
% Compute displacement and stress components at
```

```
% central point of element
[CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,LNODS,COORD,...
    PROPS,ELNOD,ASDIS,NINTP,IPCOD,CRBFI);
%-- Output results
OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,...
    NVFIX,NPLOD,NDLEG);
disp('--- All procedure are finished ---');


------------------------------------------------------
function [NINTP,IPCOD,CRBFI]=RBFINTP(NPOIN,NELEM,COORD,LNODS)
% RBF interpolation for source field in the domain
% Input parameters:
%   NELEM:  Number of elements
%   NPOIN:  Number of nodes in the domain
%   COORD:  Coordinates of nodes
%   LNODS:  Element connectivity
% Output parameters:
%   NINTP: Number of interpolation points
%   IPCOD: Coordinates of interpolation points
%   CRBFI: Coefficient of RBF interpolation
% ***********************************************
global NDIME NDOFN NNODE;

% Generate interpolation points: node+centroid
NINTP=NPOIN+NELEM;
IPCOD=zeros(NINTP,NDIME);
for iPOIN=1:NPOIN
    IPCOD(iPOIN,:)=COORD(iPOIN,:);
end
for iELEM=1:NELEM
    sum0=zeros(1,NDIME);
    for iNODE=1:NNODE
        kPOIN=LNODS(iELEM,iNODE);
        sum0(1,:)=sum0(1,:)+COORD(kPOIN,:);
    end
    IPCOD(NPOIN+iELEM,:)=sum0(1,:)/NNODE;
end

NVARB=NINTP*NDOFN;
FMATX=zeros(NVARB,NVARB);
bvect=zeros(NVARB,1);
for iINTP=1:NINTP
    xi=IPCOD(iINTP,1);
    yi=IPCOD(iINTP,2);
```

```
    % User function is used to provide the internal
    % source distribution
    [bvect(iINTP)]=USERFUN(xi,yi);
    for jINTP=1:NINTP
        xj=IPCOD(jINTP,1);
        yj=IPCOD(jINTP,2);
        r=sqrt((xi-xj)^2+(yi-yj)^2);
        % PS: r^(2*n-1)
        n=2;
        FMATX(iINTP,jINTP)=r^(2*n-1);
    end
end
% Solve linear algebraic equations
[CRBFI]=LSSOLVR(FMATX,bvect,NVARB);


------------------------------------------------------
function [GP]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,...
    NEASS,NOPRS,PRESS,NPLOD,LODPT,POINT,GP,NINTP,...
    IPCOD,CRBFI)
% Compute effective nodal forces
% Input parameters:
%   MATNO: Material index of each element
%   PROPS: Properties of materials
%   LNODS: Element connections
%   COORD: Coordinates of nodes
%   NPLOD: Number of concentrated loads
%   LODPT: Global index of nodes at which concentrated
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NDLEG: Number of loaded edges
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   GP:    Global effective nodal forces
%   NPOIN: Number of RBF interpolation points, which is
%          chosen as nodes in our application
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   GP:    Global effective nodal forces
% ***************************************
global NDIME NDOFN NNODE NEDGE NODEG NGAUS;

% Total number of local DOF of each element
```

```
NEVAB=NNODE*NDOFN;
% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);

% Evaluate equivalent nodal force on the distributed
% loaded edge
for iDLEG=1:NDLEG
    kELEM=NEASS(iDLEG);
    % Material properties
    kMATS=MATNO(kELEM);
    THICK=PROPS(kMATS,3);
    % Determine coordinates of nodes on the element edge
    ELCOD=zeros(NODEG,NDIME);
    for iODEG=1:NODEG
        kPOIN=NOPRS(iDLEG,iODEG);
        ELCOD(iODEG,:)=COORD(kPOIN,:);
    end
    % Determine local nodal load intensity
    EPRES=zeros(NODEG,NDOFN);
    for iODEG=1:NODEG
        for iDOFN=1:NDOFN
            ii=(iODEG-1)*NDOFN+iDOFN;
            EPRES(iODEG,iDOFN)=PRESS(iDLEG,ii);
        end
    end
    % Integration along the loaded edge
    %    P(iODEG)(iDOFN)=integral(N(iODEG)*p(iDOFN)*dS)
    % dS is arc-length
    PE=zeros(NEVAB,1);
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        % Shape function and its derivatives for 1D element
        SHAPE=zeros(1,NODEG);
        DSHAP=zeros(1,NODEG);
        [SHAPE,DSHAP]=SHAPFUN(EXISP);
        % Coordinates and derivatives of Gauss points
        %  x=sum(Ni*xi) and y=sum(Ni*yi)
        %  dx/dt=sum(dNi/dt*xi) and dy/dt=sum(dNi/dt*yi)
        CORGS=zeros(1,NDIME);
        DERGS=zeros(1,NDIME);
        for iDIME=1:NDIME
            for iODEG=1:NODEG
                CORGS(iDIME)=CORGS(iDIME)+...
                    SHAPE(iODEG)*ELCOD(iODEG,iDIME);
                DERGS(iDIME)=DERGS(iDIME)+...
```

```
            DSHAP(iODEG)*ELCOD(iODEG,iDIME);
    end
end
% DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
DVOLU=sqrt(DERGS(1)^2+DERGS(2)^2);
% Directional cosine at Gauss point
%  nx = dy/dS      ny = - dx/dS
nx= DERGS(2)/DVOLU;
ny=-DERGS(1)/DVOLU;
% Gauss integration factor
DVOLU=DVOLU*WEIGP(iGAUS);
if THICK+1~=1
    DVOLU=DVOLU*THICK;
end
% Load intensity at Gaussian point
%          p=sum(Ni*pi)
% for potential problems, PGASH is scalar
% denoting normal flux
PGASH=zeros(NDOFN,1);
for iODEG=1:NODEG
    PGASH(1)=PGASH(1)+...
        SHAPE(iODEG)*EPRES(iODEG,1);
end
% Modify boundary load
xp=CORGS(1);
yp=CORGS(2);
[ups,sps]=PARSOLU(xp,yp,NINTP,IPCOD,CRBFI);
PGASH(1)=PGASH(1)-(nx*sps(1)+ny*sps(2));
% Compute equivalent nodal load vector PE
for iNODE=1:NNODE
    kPOIN=LNODS(kELEM,iNODE);
    if kPOIN==NOPRS(iDLEG,1)
        % iNODE is start point of loaded edge
        for iODEG=1:NODEG
            kNODE=iNODE+iODEG-1;
            if kNODE>NNODE
                kNODE=1;
            end
            for iDOFN=1:NDOFN
                iEVAB=(kNODE-1)*NDOFN+iDOFN;
                PE(iEVAB,1)=PE(iEVAB,1)+...
                    SHAPE(iODEG)*...
                    PGASH(iDOFN)*DVOLU;
            end
        end
```

```
            end
        end
    end
    % Assemble PE into global load term
    for iNODE=1:NNODE
        kPOIN=LNODS(kELEM,iNODE);
        for iDOFN=1:NDOFN
            kEQNS=NDOFN*(kPOIN-1)+iDOFN; % global DOF
            iEVAB=NDOFN*(iNODE-1)+iDOFN; % local DOF
            GP(kEQNS,1)=GP(kEQNS,1)+PE(iEVAB,1);
        end
    end
end
clear ELCOD EPRES PE SHAPE DSHAP PGASH POSGP WEIGP;


------------------------------------------------------
function [GSTIF,GLOAD]=INDISBC(COORD,NEQNS,NVFIX,...
    NOFIX,IFPRE,PRESC,GSTIF,GLOAD,NINTP,IPCOD,CRBFI)
% Modify global stiffness matrix GSTIF and equivalent
% force vector GLOAD by the penalty approach
% Input parameters:
%   COORD: Coordinates of nodes
%   NEQNS: Total number of equations
%   NVFIX: Number of boundary nodes at which specified
%          DOF is restricted
%   NOFIX: Global index of nodes at which specified DOF
%          is restricted
%   IFPRE: Types of constraints of each DOF
%   PRESC: Specified values
%   GSTIF: Global stiffness matrix
%   GLOAD: Global equivalent nodal load vector
%   NINTP: Number of RBF interpolation points
%   IPCOD: Coordinates of RBF interpolation points
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   GSTIF: Modified global stiffness matrix
%   GLOAD: Modified global equivalent nodal load vector
% **********************************
global NDOFN;

if NVFIX>0
    % Decide penalty parameter CNST
    CNST=max(max(abs(GSTIF)));
    if CNST+1==1
```

```
        error('**Singular stiffness matrix GSTIF!');
    end
    CNST=CNST*1000000;
    % Modify GSTIF and GLOAD for specified nodal
    % displacements
    for iVFIX=1:NVFIX
        kPOIN=NOFIX(iVFIX);
        xp=COORD(kPOIN,1);
        yp=COORD(kPOIN,2);
        for iDOFN=1:NDOFN
            iGR=(kPOIN-1)*NDOFN+iDOFN;
            ii=IFPRE(iVFIX,iDOFN);
            % ii=1 indicates a constrained degree of
            % freedom
            if ii==1
                disv=PRESC(iVFIX,iDOFN);
                [ups,sps]=PARSOLU(xp,yp,NINTP,...
                    IPCOD,CRBFI);
                % Modify generalised displacement BC
                mdisv=disv-ups(iDOFN);
                GSTIF(iGR,iGR)=GSTIF(iGR,iGR)+CNST;
                GLOAD(iGR)=GLOAD(iGR)+CNST*mdisv;
            end
        end
    end
end


------------------------------------------------------
function [UNODE]=FIEDNOD(NPOIN,COORD,ASDIS,NINTP,...
    IPCOD,CRBFI)
% Generate nodal generalised displacement field
% Input parameters:
%   NPOIN: Number of nodes in domain
%   COORD: Coordinates of nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
%   NINTP: Number of RBF interpolation points
%   IPCOD: Coordinates of RBF interpolation points
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   UNODE: Nodal generalised displacement field
% *********************************
global NDOFN NDIME;
```

```
UNODE=zeros(NPOIN,NDOFN);
for iPOIN=1:NPOIN
    xp=COORD(iPOIN,1);
    yp=COORD(iPOIN,2);
    % particular solution
    [ups,sps]=PARSOLU(xp,yp,NINTP,IPCOD,CRBFI);
    % full solution
    NRT=(iPOIN-1)*NDOFN;
    for iDOFN=1: NDOFN
        NR=NRT+iDOFN;
        UNODE(iPOIN,iDOFN)=ASDIS(NR)+ups(iDOFN);
    end
end


------------------------------------------------------
function [CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    LNODS,COORD,PROPS,ELNOD,ASDIS,NINTP,IPCOD,CRBFI)
% Compute potential and flux at central point of each
% element
% Input parameters:
%   NELEM: Number of elements in domain
%   MATNO: Material index of each element
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
%   PROPS: Properties of materials
%   ELNOD: Local relation of edge and nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
%   NINTP: Number of RBF interpolation points
%   IPCOD: Coordinates of RBF interpolation points
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
% **********************************
global NDIME NDOFN NNODE NEDGE NODEG NSTRE NMATS NGAUS;

CECOD=zeros(NELEM,NDIME);
UCENP=zeros(NELEM,NDOFN);
SCENP=zeros(NELEM,NSTRE);
% Loop for all nodes
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
```

```
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Identify nodal field of the specified element
    [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Calculate the ce coefficients: m by 1
    [c_Ele]=CMATRIX(EHMTX,EGMTX,d_Ele);
    % Recover rigid displacement
    [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele);
    % Compute homogeneous solutions at specified
    % internal point (local coordinates)
    xp=0;
    yp=0;
    [N_SET,T_SET]=TREFFTZ(xp,yp);
    GDISP=N_SET*c_Ele;
    GSTRE=T_SET*c_Ele;
    % particular solution
    gxp=CenCoord(1)+xp;
    gyp=CenCoord(2)+yp;
    [ups,sps]=PARSOLU(gxp,gyp,NINTP,IPCOD,CRBFI);
    % Full solution
    UCENP(iELEM,1)=GDISP(1)+c0+ups(1);
    SCENP(iELEM,1)=GSTRE(1)+sps(1);
    SCENP(iELEM,2)=GSTRE(2)+sps(2);
    % Coordinates of commputing point
    CECOD(iELEM,:)=[gxp,gyp];
end
clear EHMTX EGMTX c_Ele d_Ele N_SET T_SET GDISP GSTRE;


-------------------------------------------------------
function [ups,sps]=PARSOLU(xp,yp,NINTP,IPCOD,CRBFI)
% Evaluate approximated particular solutions at given
% point (xp,yp)
% Input parameters:
%   xp,yp:  Coordinates of computing point
%   NINTP:  Number of interpolation points
%   IPCOD:  Coordinates of interpolation points
%   CRBFI:  Coefficient of RBF interpolation
% Output parameters:
%   ups  :  displacement particular solutions
%   sps  :  stress particular solutions
```

```
% ************************************************
global NDIME NDOFN;

up=0;
uxp=0;
uyp=0;
for jINTP=1:NINTP
    xj=IPCOD(jINTP,1);
    yj=IPCOD(jINTP,2);
    x=xp-xj;
    y=yp-yj;
    r=sqrt(x^2+y^2);
    % PS: r^(2n-1)
    n=2;
    Phi =r^(2*n+1)/((2*n+1)^2);
    Phix=r^(2*n-1)/(2*n+1)*x;
    Phiy=r^(2*n-1)/(2*n+1)*y;

    up =up +CRBFI(jINTP)*Phi;
    uxp=uxp+CRBFI(jINTP)*Phix;
    uyp=uyp+CRBFI(jINTP)*Phiy;
end
ups=up;
sps=[uxp;uyp];


-------------------------------------------------------
function [b]=USERFUN(x,y)
% Compute the source value at given point (x,y)
% Input parameters:
%   x, y:  Coordinates of computing point
% Output parameters:
%   b   :  Value of source field at point (x,y)
% ************************************************
% source field user provided according to the problem
% under consideration
b=0;
```

7.7.2 Plane stress/strain problems

```
function MAINFUN
% Main program using HTFEM with RBF approximation for
% 2D thermal-elasticity with arbitrary body forces and
% temperature change
```

```
% ******************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;

disp('*******************************************');
disp('             Hybrid Trefftz FEM');
disp('   for 2D linear thermal-elastic problems');
disp('         with arbitrary body forces');
disp('*******************************************');

% Input data from file
[NPOIN,NELEM,COORD,MATNO,LNODS,NVFIX,NOFIX,IFPRE,...
    PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,NOPRS,PRESS,...
    PROPS]=INPUTDT;
% Generate local relations of nodes and edges
[ELNOD]=TYPELEM;
% Compute the coefficient of RBF interpolation
[NINTP,IPCOD,CRBFI]=RBFINTP(NPOIN,NELEM,COORD,LNODS,...
    PROPS);
% Global stiffness matrix
NEQNS=NPOIN*NDOFN;
GSTIF=zeros(NEQNS,NEQNS);
GLOAD=zeros(NEQNS,1);
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute element stiffnesss matrix
    [ESTIF]=KMATRIX(EHMTX,EGMTX);
    % Assemble stiffness matrix
    [GSTIF]=ASMSTIF(iELEM,NNODE,LNODS,ESTIF,GSTIF);
end
% Compute equivalent forces
[GLOAD]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,...
    NOPRS,PRESS,GLOAD,NINTP,IPCOD,CRBFI);
% Introduce constrained displacements and point loads
[GSTIF,GLOAD]=INDISBC(NEQNS,PROPS,COORD,NVFIX,...
    NOFIX,IFPRE,PRESC,GSTIF,GLOAD,NINTP,IPCOD,CRBFI);
% Solve linear system of equations and store
% displacements of each node in the array ASDIS
[ASDIS]=LSSOLVR(GSTIF,GLOAD,NEQNS);
```

```
% Output nodal potential
[UPOIN]=FIEDNOD(NPOIN,COORD,ASDIS,NINTP,IPCOD,...
    CRBFI,PROPS);
% Compute potential and flux components at central
% point of element
[CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,LNODS,...
    COORD,PROPS,ELNOD,ASDIS,NINTP,IPCOD,CRBFI);
% Output results
OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,...
    NVFIX,NPLOD,NDLEG);
disp('--- All procedures are finished ---');


------------------------------------------------------
function [NINTP,IPCOD,CRBFI]=RBFINTP(NPOIN,NELEM,...
    COORD,LNODS,PROPS)
% ** RBF interpolation in the domain
% Input parameters:
%   NPOIN:  Number of nodes in the domain
%   NELEM:  Number of elements
%   COORD:  Coordinates of nodes
%   LNODS:  Element connectivity
%   PROPS:  Material properties
% Output parameters:
%   NINTP:  Number of RBF interpolation points
%   IPCOD:  Coordinates of RBF interpolation points
%   CRBFI:  Coefficients of RBF interpolation
% ***********************************************
global NDIME NDOFN NNODE NTYPE;

% Material properties
YOUNG=PROPS(1,1);
POISS=PROPS(1,2);
THICK=PROPS(1,3);
TEXPN=PROPS(1,4);
DENST=PROPS(1,5);
if NTYPE==1 %plane stress
    gm=YOUNG*TEXPN/(1-POISS);
elseif NTYPE==2
    gm=YOUNG*TEXPN/(1-2*POISS);
end

% Generate interpolation points: node+centroid
NINTP=NPOIN+NELEM;
IPCOD=zeros(NINTP,NDIME);
```

```
for iPOIN=1:NPOIN
    IPCOD(iPOIN,:)=COORD(iPOIN,:);
end
% Compute central coordinates of the given element
for iELEM=1:NELEM
    sum0=zeros(1,NDIME);
    for iNODE=1:NNODE
        kPOIN=LNODS(iELEM,iNODE);
        sum0(1,:)=sum0(1,:)+COORD(kPOIN,:);
    end
    IPCOD(NPOIN+iELEM,:)=sum0(1,:)/NNODE;
end

NTVAR=NINTP*NDOFN;
FMATX=zeros(NTVAR,NTVAR);
bvect=zeros(NTVAR,1);
for iINTP=1:NINTP
    xi=IPCOD(iINTP,1);
    yi=IPCOD(iINTP,2);
    % User function is used to provide body forces and
    % temperature
    [GBFOR,TEMPR]=USERFUN(xi,yi,PROPS);
    bvect(2*iINTP-1)=GBFOR(1);
    bvect(2*iINTP)  =GBFOR(2);
    for jINTP=1:NINTP
        xj=IPCOD(jINTP,1);
        yj=IPCOD(jINTP,2);
        r=sqrt((xi-xj)^2+(yi-yj)^2);
        % TPS: r^(2*n)ln(r)
        n=2;
        if 1+r==1
            r=0;
            lnr=0;
        else
            lnr=log(r);
        end
        temp=r^(2*n)*lnr;
        FMATX(2*iINTP-1,2*jINTP-1)=r^(2*n)*lnr;
        FMATX(2*iINTP,  2*jINTP)  =r^(2*n)*lnr;
    end
end
% Solve linear algebraic equations
[CRBFI]=LSSOLVR(FMATX,bvect,NTVAR);
```

```
-----------------------------------------------------
function [GP]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,...
    NEASS,NOPRS,PRESS,GP,NINTP,IPCOD,CRBFI)
% Compute effective nodal forces
% Input parameters:
%   MATNO: Material index of each element
%   PROPS: Properties of materials
%   LNODS: Element connections
%   COORD: Coordinates of nodes
%   NPLOD: Number of concentrated loads
%   LODPT: Global index of nodes at which concentrated
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NDLEG: Number of loaded edges
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   GP:    Global effective nodal forces
%   NPOIN: Number of RBF interpolation points
%   IPCOD: Coordinates of interpolation points
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   GP:    Global effective nodal forces
% ***************************************

global NDIME NDOFN NNODE NEDGE NODEG NTYPE NGAUS;

% Material properties
YOUNG=PROPS(1,1);
POISS=PROPS(1,2);
THICK=PROPS(1,3);
TEXPN=PROPS(1,4);
DENST=PROPS(1,5);
if NTYPE==1 %plane stress
    gm=YOUNG*TEXPN/(1-POISS);
elseif NTYPE==2 %plane strain
    gm=YOUNG*TEXPN/(1-2*POISS);
end

NEVAB=NNODE*NDOFN;
% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);

% Calculate equivalent nodal loads on the distributed loaded edge
```

```
for iDLEG=1:NDLEG
    kELEM=NEASS(iDLEG);
    % Determine coordinates of nodes on the element edge
    ELCOD=zeros(NODEG,NDIME);
    for iODEG=1:NODEG
        kPOIN=NOPRS(iDLEG,iODEG);
        for iDIME=1:NDIME
            ELCOD(iODEG,iDIME)=COORD(kPOIN,iDIME);
        end
    end
    % Determine local nodal load intensity
    EPRES=zeros(NODEG,NDOFN);
    for iODEG=1:NODEG
        for iDOFN=1:NDOFN
            ii=(iODEG-1)*NDOFN+iDOFN;
            EPRES(iODEG,iDOFN)=PRESS(iDLEG,ii);
        end
    end
    % Integration along the loaded edge
    %    P(iODEG)(iDOFN)=integral(N(iODEG)*p(iDOFN)*dS)
    % dS is arc-length
    PE=zeros(NEVAB,1);
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        % Shape functions and its derivatives for 1D
        % line element
        SHAPE=zeros(1,NODEG);
        DSHAP=zeros(1,NODEG);
        [SHAPE,DSHAP]=SHAPFUN(EXISP);

        % Coordinates and derivatives of Gaussian
        % points
        %  x=sum(Ni*xi) and y=sum(Ni*yi)
        %  dx/dt=sum(dNi/dt*xi)
        %  dy/dt=sum(dNi/dt*yi)
        %  DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
        CORGS=zeros(1,NDIME);
        DERGS=zeros(1,NDIME);
        for iDIME=1:NDIME
            for iODEG=1:NODEG
                CORGS(iDIME)=CORGS(iDIME)+...
                    SHAPE(iODEG)*ELCOD(iODEG,iDIME);
                DERGS(iDIME)=DERGS(iDIME)+...
                    DSHAP(iODEG)*ELCOD(iODEG,iDIME);
            end
```

```
end
DVOLU=sqrt(DERGS(1)^2+DERGS(2)^2);
% Directional cosine at Gaussian point
%  nx = dy/dS,   ny = - dx/dS
DS1= DERGS(2)/DVOLU;
DS2=-DERGS(1)/DVOLU;

% Gaussian integration factor
DVOLU=DVOLU*WEIGP(iGAUS)*THICK;

% Load intensity at Gaussian point
%          p=sum(Ni*pi)
% for plane elastic problems, that is, NDOFN=2,
%   PGASH(1) is normal pressure, which is
%   assumed to be positive if it acts in a
%   direction into the element
%   PGASH(2) is tangential load, which is
%   assumed to be positive if it acts in an
%   anticlockwise direction with respect to
%   the loaded element
PGASH=zeros(NDOFN,1);
for iDOFN=1:NDOFN % pn, pt
    for iODEG=1:NODEG
        PGASH(iDOFN)=PGASH(iDOFN)+...
            SHAPE(iODEG)*EPRES(iODEG,iDOFN);
    end
end
%px=-pn*nx-pt*ny
%py=-pn*ny+pt*nx
PX=-DS1*PGASH(1)-DS2*PGASH(2); % tx
PY=-DS2*PGASH(1)+DS1*PGASH(2); % ty
PGASH(1)=PX;
PGASH(2)=PY; % PGASH is reset value
% Modify boundary tractions at Gaussian points
% th(i)=t(i)-tp(i)+m*T*n(i)
xp=CORGS(1);
yp=CORGS(2);
% particular solution
[ups,sps]=PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,...
    PROPS);
% Specified temperature at Gaussian point
[GBFOR,TEMPR]=USERFUN(xp,yp,PROPS);
% Evaluate homogeneous tractions
PGASH(1)=PGASH(1)-(sps(1)*DS1+sps(3)*DS2)+...
    gm*TEMPR*DS1;
```

```
        PGASH(2)=PGASH(2)-(sps(3)*DS1+sps(2)*DS2)+...
            gm*TEMPR*DS2;
        % Compute equivalent nodal force vector PE
        for iNODE=1:NNODE
            kPOIN=LNODS(kELEM,iNODE);
            if kPOIN==NOPRS(iDLEG,1)
                % iNODE is start point of loaded edge
                for iODEG=1:NODEG
                    kNODE=iNODE+iODEG-1;
                    if kNODE>NNODE
                        kNODE=1;
                    end
                    for iDOFN=1:NDOFN
                        iEVAB=(kNODE-1)*NDOFN+iDOFN;
                        PE(iEVAB,1)=PE(iEVAB,1)+...
                            SHAPE(iODEG)*...
                            PGASH(iDOFN)*DVOLU;
                    end
                end
            end
        end
    end
    % Assemble PE into global load term
    for iNODE=1:NNODE
        kPOIN=LNODS(kELEM,iNODE);
        for iDOFN=1:NDOFN
            kEQNS=NDOFN*(kPOIN-1)+iDOFN; % global DOF
            iEVAB=NDOFN*(iNODE-1)+iDOFN; % local DOF
            GP(kEQNS,1)=GP(kEQNS,1)+PE(iEVAB,1);
        end
    end
end
clear ELCOD EPRES PE SHAPE DSHAP PGASH POSGP WEIGP;


------------------------------------------------------
function [GK,GP]=INDISBC(NEQNS,PROPS,COORD,NVFIX,...
    NOFIX,IFPRE,PRESC,GK,GP,NINTP,IPCOD,CRBFI)
% Modify global stiffness matrix GSTIF and equivalent
% load vector GLOAD by the penalty approach
% Input parameters:
%   NEQNS: Number of equations
%   PROPS: Material properties
%   COORD: Coordinates of nodes
%   NVFIX: Number of boundary nodes at which specified
```

```
%          DOF is restricted
%   NOFIX: Global index of nodes at which specified DOF
%          is restricted
%   IFPRE: Types of constraints of each DOF
%   PRESC: Specified values
%   GK   : Global stiffness matrix
%   GP   : Global equivalent nodal load vector
%   NINTP: Number of RBF interpolation points
%   IPCOD: Coordinates of RBF interpolation points
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   GK   : Modified global stiffness matrix
%   GP   : Modified global equivalent nodal load vector
% ********************************
global NDOFN;

if NVFIX>0
    % Decide penalty parameter CNST
    CNST=max(max(abs(GK)));
    if CNST+1==1
        error('**Singular stiffness matrix GK!');
    end
    CNST=CNST*1000000;
    % Modify GK and GP for specified nodal
    % displacements
    for iVFIX=1:NVFIX
        kPOIN=NOFIX(iVFIX);
        xp=COORD(kPOIN,1);
        yp=COORD(kPOIN,2);
        for iDOFN=1: NDOFN
            NR=(kPOIN-1)*NDOFN+iDOFN;
            ii=IFPRE(iVFIX,iDOFN);
            % unit value indicates a constrained DOF
            if ii==1
                disv=PRESC(iVFIX,iDOFN);
                % particular solution
                [ups,sps]=PARSOLU(xp,yp,NINTP,...
                    IPCOD,CRBFI,PROPS);
                % Evaluate homogeneous displacement BC
                mdisv=disv-ups(iDOFN);
                GK(NR,NR)=GK(NR,NR)+CNST;
                GP(NR)=GP(NR)+CNST*mdisv;
            end
        end
    end
```

```
end


------------------------------------------------------
function [UNODE]=FIEDNOD(NPOIN,COORD,ASDIS,NINTP,...
    IPCOD,CRBFI,PROPS)
% Generate nodal generalised displacement field
% Input parameters:
%   NPOIN: Number of nodes in domain
%   COORD: Coordinates of nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
%   NINTP: Number of RBF interpolation points
%   IPCOD: Coordinates of RBF interpolation points
%   CRBFI: Coefficient of RBF interpolation
%   PROPS: Material properties
% Output parameters:
%   UNODE: Nodal generalised displacement field
% **********************************
global NDOFN NDIME;

UNODE=zeros(NPOIN,NDOFN);
for iPOIN=1:NPOIN
    xp=COORD(iPOIN,1);
    yp=COORD(iPOIN,2);
    % particular solution
    [ups,sps]=PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,PROPS);
    % full solution
    NRT=(iPOIN-1)*NDOFN;
    for iDOFN=1:NDOFN
        NR=NRT+iDOFN;
        UNODE(iPOIN,iDOFN)=ASDIS(NR)+ups(iDOFN);
    end
end


------------------------------------------------------
function [CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    LNODS,COORD,PROPS,ELNOD,ASDIS,NINTP,IPCOD,...
    CRBFI)
% Compute potential and flux at central point of each
% element
% Input parameters:
%   NELEM: Number of elements in domain
%   MATNO: Material index of each element
```

```
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
%   PROPS: Properties of materials
%   ELNOD: Local relation of edge and nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
%   NINTP: Number of RBF interpolation points
%   IPCOD: Coordinates of RBF interpolation points
%   CRBFI: Coefficient of RBF interpolation
% Output parameters:
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
% **********************************
global NDIME NDOFN NNODE NEDGE NODEG NSTRE NTYPE NGAUS;

% Material properties
YOUNG=PROPS(1,1);
POISS=PROPS(1,2);
THICK=PROPS(1,3);
TEXPN=PROPS(1,4);
DENST=PROPS(1,5);
if NTYPE==1 %plane stress
    gm=YOUNG*TEXPN/(1-POISS);
elseif NTYPE==2
    gm=YOUNG*TEXPN/(1-2*POISS);
end

CECOD=zeros(NELEM,NDIME);
UCENP=zeros(NELEM,NDOFN);
SCENP=zeros(NELEM,NSTRE);
% Loop for all nodes
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Identify nodal field of the specified element
    [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,PROPS);
    % Calculate the ce coefficients: m by 1
    [c_Ele]=CMATRIX(EHMTX,EGMTX,d_Ele);
    % Recover rigid-body motion vector: 3 by 1
```

```
    [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,PROPS);
    % Compute Trefftz internal fields at central point
    xp=0;
    yp=0;
    [N_SET,T_SET]=TREFFTZ(xp,yp,kMATS,PROPS);
    GDISP=N_SET*c_Ele;
    GSTRE=T_SET*c_Ele;
    % particular solution
    gxp=CenCoord(1)+xp;
    gyp=CenCoord(2)+yp;
    [ups,sps]=PARSOLU(gxp,gyp,NINTP,IPCOD,CRBFI,PROPS);
    % Temperature effect
    [GBFOR,TEMPR]=USERFUN(gxp,gyp,PROPS);
    % Full solution
    UCENP(iELEM,1)=GDISP(1)+c0(1)+yp*c0(3)+ups(1);
    UCENP(iELEM,2)=GDISP(2)+c0(2)-xp*c0(3)+ups(2);
    SCENP(iELEM,1)=GSTRE(1)+sps(1)-gm*TEMPR;
    SCENP(iELEM,2)=GSTRE(2)+sps(2)-gm*TEMPR;
    SCENP(iELEM,3)=GSTRE(3)+sps(3);
    % Coordinates of computing points
    CECOD(iELEM,:)=[gxp,gyp];
end
clear EHMTX EGMTX c_Ele d_Ele;


------------------------------------------------------
function [ups,sps]=PARSOLU(xp,yp,NINTP,IPCOD,...
    CRBFI,PROPS)
% ** Evaluate approximated particular solutions at
%    given point (x,y)
% Input parameters:
%   xp,yp:  Coordinates of computing point
%   NINTP:  Number of interpolation points
%   IPCOD:  Coordinates of interpolation points
%   CRBFI:  Coefficient of RBF interpolation
%   PROPS:  Material properties
% Output parameters:
%   ups: Displacement vector
%   sps: Stress vector
% ***********************************************
global NDIME NDOFN NTYPE NSTRE;

% Material properties
YOUNG=PROPS(1,1);
POISS=PROPS(1,2);
```

```
THICK=PROPS(1,3);
TEXPN=PROPS(1,4);
DENST=PROPS(1,5);
G=YOUNG/(2*(1+POISS));
if NTYPE==1 %plane stress
    mu=POISS/(1+POISS);
elseif NTYPE==2 %plane strain
    mu=POISS;
end
% Particular solutions
ups=zeros(1,NDOFN);
sps=zeros(1,NSTRE);
for jINTP=1:NINTP
    xj=IPCOD(jINTP,1);
    yj=IPCOD(jINTP,2);
    x=xp-xj;
    y=yp-yj;
    r=sqrt(x^2+y^2);
    % TPS: r^(2*n)ln(r)
    if 1+r==1
        r=0;
        lnr=0;
        r1=0;
        r2=0;
    else
        lnr=log(r);
        r1=x/r; % dr/dx
        r2=y/r; % dr/dy
    end
    n=2;
    A1=-(8*n^2+29*n+27)+8*mu*(n+2)^2+...
        2*(n+1)*(n+2)*(4*n+7-4*mu*(n+2))*lnr;
    A2=2*(n+1)*(2*n+3)-4*(n+1)^2*(n+2)*lnr;
    A3=2*n+3-2*mu*(n+2)^2+...
        2*(n+1)*(n+2)*(2*mu*n+4*mu-1)*lnr;
    A4=2*(n^2-2)-4*n*(n+1)*(n+2)*lnr;
    A5=-(2*n^2+6*n+5)+2*mu*(n+2)^2-...
        2*(n+1)*(n+2)*(-(2*n+3)+2*mu*(n+2))*lnr;
    temp1=-1/(32*G*(1-mu))*r^(2*n+2)/((n+1)^3*(n+2)^2);
    temp2=-1/(8*(1-mu))*r^(2*n+1)/((n+1)^2*(n+2)^2);

    u11=temp1*(A1+A2*r1*r1);
    u12=temp1*(A2*r1*r2);
    u22=temp1*(A1+A2*r2*r2);
    s111=temp2*(A3*r1+A4*r1*r1*r1+A5*(2*r1));
```

```
    s112=temp2*(A3*r2+A4*r1*r1*r2);
    s121=temp2*(A4*r1*r2*r1+A5*r2);
    s122=temp2*(A4*r1*r2*r2+A5*r1);
    s221=temp2*(A3*r1+A4*r2*r2*r1);
    s222=temp2*(A3*r2+A4*r2*r2*r2+A5*(2*r2));

    ups(1)=ups(1)+CRBFI(2*jINTP-1)*u11 +...
        CRBFI(2*jINTP)*u12;
    ups(2)=ups(2)+CRBFI(2*jINTP-1)*u12 +...
        CRBFI(2*jINTP)*u22;
    sps(1)=sps(1)+CRBFI(2*jINTP-1)*s111+...
        CRBFI(2*jINTP)*s112;
    sps(2)=sps(2)+CRBFI(2*jINTP-1)*s221+...
        CRBFI(2*jINTP)*s222;
    sps(3)=sps(3)+CRBFI(2*jINTP-1)*s121+...
        CRBFI(2*jINTP)*s122;
end


------------------------------------------------------
function [GBFOR,TEMPR]=USERFUN(x,y,PROPS)
% Compute the source value at given point (x,y)
% Input parameters:
%   x, y :  Coordinates of computing point
%   PROPS:  Material properties
% Output parameters:
%   GBFOR:  generalised body forces at point (x,y)
%   TEMPR:  Temperature at point (x,y)
% ***********************************************
global NTYPE;

% Material properties
YOUNG=PROPS(1,1);
POISS=PROPS(1,2);
THICK=PROPS(1,3);
TEXPN=PROPS(1,4);
DENST=PROPS(1,5);
%
if NTYPE==1 %plane stress
    gm=YOUNG*TEXPN/(1-POISS);
elseif NTYPE==2
    gm=YOUNG*TEXPN/(1-2*POISS);
end

% Body forces user provided according to the problem
```

```
% under consideration
BODYF(1)=0; % bx
BODYF(2)=0; % by
% Temperature user provided according to the problem
% under consideration
TEMPR=0;    % T
DTEMP(1)=0; % dT/dx
DTEMP(2)=0; % dT/dy

% Generalised body forces
GBFOR=zeros(2,1);
GBFOR(1)=BODYF(1)-gm*DTEMP(1);
GBFOR(2)=BODYF(2)-gm*DTEMP(2);
```

7.8 C programming

Considering the practical characteristic of the interpolation matrix in RBF approximation, the algorithm of Gaussian elimination with column pivoting employed to solve the final stiffness equation in C programming maybe is failed for obtaining unknown coefficients. At this time, the solver LAS_SVD for linear equations with singular value decomposition is used. For simplify, we don't list it in the following, the reader can refer to the disc for detail.

7.8.1 Two-dimensional Poisson's problems

```
/*
*********************************************************
* Mainfunction MAINFUN                                  *
* - Call other subroutines                              *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int NTREF,NTYPE,NNODE,NEDGE,NODEG,NDIME,NDOFN,NSTRE,
NMATS,NPROP,NGAUS;
void main()
{
    void DUCLEAN(); void ITCLEAN(); void INPUTDT();
```

```
void TYPELEM(); void ELEPARS();
void HMATRIX(); void GMATRIX(); void KMATRIX();
void ASMSTIF(); void PVECTOR(); void INDISBC();
void LSSOLVR(); void FIEDNOD(); void FIEDCEN();
void OPRESUT(); void RBFINTP();
FILE *fp;
int NEQNS,NPOIN,NELEM,NVFIX,NPLOD,NDLEG,TNFEG,NEVAB,NINTP,NTVAR;
int *MATNO,*LNODS,*NOFIX,*IFPRE,*LODPT,*NEASS,*NOPRS,
    *ELNOD;
double *COORD,*PRESC,*POINT,*PRESS,*PROPS;
double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*ESTIF,*GSTIF,
    *GLOAD,*UPOIN,*CECOD,*UCENP,*SCENP,*IPCOD,*CRBFI;
char dummy[201],TITLE[201],file[81];
int i,j,k,N,n1,n2,iELEM,kMATS;

printf("************************************\n");
printf("        Hybrid Trefftz FEM\n");
printf("      for 2D Poisson's problems\n");
printf("************************************\n");
/** Input data from file **/
puts("Input file name < dir:fn.txt >: ");
gets(file);
if((fp=fopen(file,"r"))==NULL)
{
    printf("Warning! Can't open input file\n");
    exit(0);
}
// basic parameters
fgets(dummy,200,fp);
fgets(TITLE,200,fp);
fgets(dummy,200,fp);

fgets(dummy,200,fp);
fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

fgets(dummy,200,fp);
```

```
fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
    &NPLOD,&NDLEG);

// element connectivity
MATNO=(int *)calloc(NELEM,sizeof(int));
ITCLEAN(NELEM,1,MATNO);
LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
ITCLEAN(NELEM,NNODE,LNODS);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NELEM;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    MATNO[i]=n1-1;
    for(j=0;j<NNODE;j++)
    {
        fscanf(fp,"%d",&n2);
        LNODS[i*NNODE+j]=n2-1;
    }
    fscanf(fp,"\n");
}
// nodal coordinates
COORD=(double *)calloc(NPOIN*NDIME,sizeof(double));
DUCLEAN(NPOIN,NDIME,COORD);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NPOIN;i++)
{
    fscanf(fp,"%d",&N);
    for(j=0;j<NDIME;j++)
    {
        fscanf(fp,"%lf",&COORD[i*NDIME+j]);
    }
    fscanf(fp,"\n");
}
// specified nodal potential/displacement
NOFIX=(int *)calloc(NVFIX,sizeof(int));
ITCLEAN(NVFIX,1,NOFIX);
IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
ITCLEAN(NVFIX,NDOFN,IFPRE);
PRESC=(double *)calloc(NVFIX*NDOFN,sizeof(double));
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
```

```
{
    fscanf(fp,"%d %d",&N,&n1);
    NOFIX[i]=n1-1;
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
    }
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
    }
    fscanf(fp,"\n");
}
// specified concentrated loads at nodes
fgets(dummy,200,fp);
if(NPLOD>0)
{
    LODPT=(int *)calloc(NPLOD*1,sizeof(int));
    ITCLEAN(NPLOD,1,LODPT);
    POINT=(double *)calloc(NPLOD*NDOFN,
        sizeof(double));
    DUCLEAN(NPLOD,NDOFN,POINT);
    fgets(dummy,200,fp);
    for(i=0;i<NPLOD;i++)
    {
        fscanf(fp,"%d %d",&N,&n1);
        LODPT[i]=n1-1;
        for(j=0;j<NDOFN;j++)
        {
            fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
        }
        fscanf(fp,"\n");
    }
}
// specified distributed edge loads
fgets(dummy,200,fp);
if(NDLEG>0)
{
    NEASS=(int *)calloc(NDLEG*1,sizeof(int));
    ITCLEAN(NDLEG,1,NEASS);
    NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
    ITCLEAN(NDLEG,NODEG,NOPRS);
    TNFEG=NODEG*NDOFN;
    PRESS=(double *)calloc(NDLEG*TNFEG,sizeof(double));
    DUCLEAN(NDLEG,TNFEG,PRESS);
```

```
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NPROP;j++)
        {
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }

    /** Establish local relations of nodes and edges **/
    ELNOD=(int *)calloc(NEDGE*NODEG,sizeof(int));
    ITCLEAN(NEDGE,NODEG,ELNOD);
    TYPELEM(ELNOD);

    /** Compute the coefficient of RBF interpolation **/
    NINTP=NPOIN+NELEM;
    IPCOD=(double *)calloc(NINTP*NDIME,sizeof(double));
    DUCLEAN(NINTP,NDIME,IPCOD);
    NTVAR=NINTP*NDOFN;
    CRBFI=(double *)calloc(NTVAR,sizeof(double));
    DUCLEAN(NTVAR,1,CRBFI);
    RBFINTP(NPOIN,NELEM,COORD,LNODS,NINTP,IPCOD,CRBFI);
```

```
/** Form stiffness matrix **/
NEQNS=NPOIN*NDOFN;
GSTIF=(double *)calloc(NEQNS*NEQNS,sizeof(double));
DUCLEAN(NEQNS,NEQNS,GSTIF);
for(iELEM=0;iELEM<NELEM;iELEM++)
{
    kMATS=MATNO[iELEM];
    // Compute some quantities related to each element
    ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
    DUCLEAN(NNODE,NDIME,ECOOD);
    CenCoord=(double *)calloc(1*NDIME,sizeof(double));
    DUCLEAN(1,NDIME,CenCoord);
    ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
    // Compute H matrix
    EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
    DUCLEAN(NTREF,NTREF,EHMTX);
    HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
    // Compute G matrix
    NEVAB=NNODE*NDOFN;
    EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
    DUCLEAN(NTREF,NEVAB,EGMTX);
    GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
    // Compute element stiffness matrix
    ESTIF=(double *)calloc(NEVAB*NEVAB,sizeof(double));
    DUCLEAN(NEVAB,NEVAB,ESTIF);
    KMATRIX(EHMTX,EGMTX,ESTIF);
    // Assemble stiffness matrix
    ASMSTIF(iELEM,NEQNS,LNODS,ESTIF,GSTIF);
    free(EHMTX); free(EGMTX); free(ESTIF);
}
// Compute equivalent loads
GLOAD=(double *)calloc(NEQNS*1,sizeof(double));
DUCLEAN(NEQNS,1,GLOAD);
PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,
    NOPRS,PRESS,NPLOD,LODPT,POINT,GLOAD,NINTP,
    IPCOD,CRBFI);

// Introduce constrained displacements
INDISBC(COORD,NEQNS,NVFIX,NOFIX,IFPRE,PRESC,
    GSTIF,GLOAD,NINTP,IPCOD,CRBFI);
// Solve linear system of equations
LSSOLVR(GSTIF,GLOAD,NEQNS);

// Output nodal displacement
```

```
    UPOIN=(double *)calloc(NPOIN*NDOFN,sizeof(double));
    DUCLEAN(NPOIN,NDOFN,UPOIN);
    FIEDNOD(NPOIN,COORD,GLOAD,NINTP,IPCOD,CRBFI,UPOIN);

    // Compute quantities at centroid of each element
    CECOD=(double *)calloc(NELEM*NDIME,sizeof(double));
    DUCLEAN(NELEM,NDIME,CECOD);
    UCENP=(double *)calloc(NELEM*NDOFN,sizeof(double));
    DUCLEAN(NELEM,NDOFN,UCENP);
    SCENP=(double *)calloc(NELEM*NSTRE,sizeof(double));
    DUCLEAN(NELEM,NSTRE,SCENP);
    FIEDCEN(NELEM,MATNO,LNODS,COORD,PROPS,ELNOD,GLOAD,
        NINTP,IPCOD,CRBFI,CECOD,UCENP,SCENP);

    // Output results
    OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,
        NVFIX,NPLOD,NDLEG);

    free(COORD); free(LNODS); free(MATNO);
    free(NOFIX); free(IFPRE); free(PRESC);
    free(PROPS); free(ECOOD); free(CenCoord);
    free(GSTIF); free(GLOAD); free(UPOIN);
    free(CECOD); free(UCENP); free(SCENP);
    free(IPCOD); free(CRBFI);
    printf("------------ Finished ------------\n");
    return;
}


/*
*******************************************************
* Subroutine RBFINTP                                  *
* -Compute coefficients of RBF interpolation          *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void RBFINTP(int NPOIN,int NELEM,double COORD[],
             int LNODS[],int NINTP,double IPCOD[],
             double bvect[])
{
    void DUCLEAN();
    void LSSOLVR();
    void USERFUN();
```

```
extern int NDIME,NDOFN,NNODE;
int iP,iD,iE,iN,kP,iT,jT,n1,n,NVARB;
double *sum0,*FMATX,xi,yi,xj,yj,r;
// Generate interpolation points: node+centroid
for(iP=0;iP<NPOIN;iP++)
{
    for(iD=0;iD<NDIME;iD++)
    {
        IPCOD[iP*NDIME+iD]=COORD[iP*NDIME+iD];
    }
}
sum0=(double *)calloc(NDIME,sizeof(double));
for(iE=0;iE<NELEM;iE++)
{
    n1=NPOIN+iE;
    // Form nodal coordinates of the given element
    DUCLEAN(1,NDIME,sum0);
    for(iD=0;iD<NDIME;iD++)
    {
        for(iN=0;iN<NNODE;iN++)
        {
            kP=LNODS[iE*NNODE+iN];
            sum0[iD]=sum0[iD]+COORD[kP*NDIME+iD];
        }
        IPCOD[n1*NDIME+iD]=sum0[iD]/NNODE;
    }
}
// Form coefficient matrix
NVARB=NINTP*NDOFN;
FMATX=(double *)calloc(NVARB*NVARB,sizeof(double));
DUCLEAN(NVARB,NVARB,FMATX);
for(iT=0;iT<NINTP;iT++)
{
    xi=IPCOD[iT*NDIME+0];
    yi=IPCOD[iT*NDIME+1];
    // compute right-hand term b
    USERFUN(xi,yi,&(bvect[iT]));
    for(jT=0;jT<NINTP;jT++)
    {
        xj=IPCOD[jT*NDIME+0];
        yj=IPCOD[jT*NDIME+1];
        r=sqrt(pow((xi-xj),2)+pow((yi-yj),2));
        // PS: r^(2n-1)
        n=2;
        FMATX[iT*NVARB+jT]=pow(r,(2*n-1));
```

```
        }
    }
    // Solve linear algebraic equations
    LAS_SVD(FMATX,bvect,NVARB);
    free(FMATX); free(sum0);
    return;
}


/*
*******************************************************
* Subroutine PVECTOR                                  *
* - Compute effective nodal force                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void PVECTOR(int MATNO[],double PROPS[],int LNODS[],
             double COORD[],int NDLEG,int NEASS[],
             int NOPRS[],double PRESS[],int NPLOD,
             int LODPT[],double POINT[],double GP[],
             int NINTP,double IPCOD[],double CRBFI[])
{
    void GAUSSQU();
    void SHAPFUN();
    void DUCLEAN();
    void PARSOLU();
    extern int NDIME,NDOFN,NGAUS,NPROP,NNODE,NEDGE,
        NODEG,NSTRE;
    double *POSGP,*WEIGP,*ELCOD,*EPRES,*PE,*SHAPE,
        *DSHAP,*CORGS,*DERGS,*PGASH;
    int iDLEG,iNODE,kNODE,iODEG,iDOFN,iDIME,
        jGAUS,kPOIN,kELEM,kMATS,kEQNS,iEVAB,n1,n2,
        NEVAB,TNFEG;
    double EXISP,THICK,DVOLU,xp,yp,*ups,*sps,nx,ny;

    // Evaluate equivalent nodal force caused
    // by distributed edge load
    NEVAB=NNODE*NDOFN;
    TNFEG=NODEG*NDOFN;
    // Gaussian point and weight coefficients
    POSGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,POSGP);
    WEIGP=(double *)calloc(NGAUS,sizeof(double));
```

```
DUCLEAN(NGAUS,1,WEIGP);
GAUSSQU(POSGP,WEIGP);
for(iDLEG=0;iDLEG<NDLEG;iDLEG++)
{
    kELEM=NEASS[iDLEG];
    // Material properties
    kMATS=MATNO[kELEM];
    THICK=PROPS[kMATS*NPROP+2];
    // Determine coordinates of nodes on the element edge
    ELCOD=(double *)calloc(NODEG*NDIME,sizeof(double));
    DUCLEAN(NODEG,NDIME,ELCOD);
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        kPOIN=NOPRS[iDLEG*NODEG+iODEG];
        for(iDIME=0;iDIME<NDIME;iDIME++)
        {
            n1=iODEG*NDIME+iDIME;
            n2=kPOIN*NDIME+iDIME;
            ELCOD[n1]=COORD[n2];
        }
    }
    // Determine local nodal load intensity
    EPRES=(double *)calloc(NODEG*NDOFN,sizeof(double));
    DUCLEAN(NODEG,NDOFN,EPRES);
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
        {
            n1=iODEG*NDOFN+iDOFN;
            n2=iDLEG*TNFEG+n1;
            EPRES[n1]=PRESS[n2];
        }
    }
    // Integration along the loaded edge
    PE=(double *)calloc(NEVAB,sizeof(double));
    DUCLEAN(NEVAB,1,PE);
    for(jGAUS=0;jGAUS<NGAUS;jGAUS++)
    {
        EXISP=POSGP[jGAUS];
        // shape functions and its derivatives
        SHAPE=(double *)calloc(NODEG,sizeof(double));
        DUCLEAN(1,NODEG,SHAPE);
        DSHAP=(double *)calloc(NODEG,sizeof(double));
        DUCLEAN(1,NODEG,DSHAP);
        SHAPFUN(EXISP,SHAPE,DSHAP);
```

```
// Coordinates and derivatives of Gauss points
// x=sum(Ni*xi) and y=sum(Ni*yi)
// dx/dt=sum(dNi/dt*xi), dy/dt=sum(dNi/dt*yi)
// DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
CORGS=(double *)calloc(NDIME,sizeof(double));
DUCLEAN(1,NDIME,CORGS);
DERGS=(double *)calloc(NDIME,sizeof(double));
DUCLEAN(1,NDIME,DERGS);
for(iDIME=0;iDIME<NDIME;iDIME++)
{
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        n1=iODEG*NDIME+iDIME;
        CORGS[iDIME]=CORGS[iDIME]+
            SHAPE[iODEG]*ELCOD[n1];
        DERGS[iDIME]=DERGS[iDIME]+
            DSHAP[iODEG]*ELCOD[n1];
    }
}
DVOLU=sqrt(pow(DERGS[0],2.0)+
    pow(DERGS[1],2.0));
// Directional cosine at Gaussian point
//  nx = dy/dS   ny = - dx/dS
nx= DERGS[1]/DVOLU;
ny=-DERGS[0]/DVOLU;
// Gauss integration factor
DVOLU=DVOLU*WEIGP[jGAUS];
if((THICK+1)!=1)
{
    DVOLU=DVOLU*THICK;
}
// flux intensity at Gauss point
//          p=sum(Ni*pi)
PGASH=(double *)calloc(NDOFN,sizeof(double));
DUCLEAN(NDOFN,1,PGASH);
for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
{
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        n1=iODEG*NDOFN+iDOFN;
        PGASH[iDOFN]=PGASH[iDOFN]+
            SHAPE[iODEG]*EPRES[n1];
    }
}
// Modify boundary load
```

```
            xp=CORGS[0];
            yp=CORGS[1];
            ups=(double *)calloc(NDOFN,sizeof(double));
            DUCLEAN(NDOFN,1,ups);
            sps=(double *)calloc(NSTRE,sizeof(double));
            DUCLEAN(NSTRE,1,sps);
            PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,ups,sps);
            PGASH[0]=PGASH[0]-(nx*sps[0]+ny*sps[1]);
            // Compute equivalent nodal force PE
            for(iNODE=0;iNODE<NNODE;iNODE++)
            {
                kPOIN=LNODS[kELEM*NNODE+iNODE];
                if(kPOIN==NOPRS[iDLEG*NODEG+0])
                {
                    // iNODE is start node of the
                    // loaded edge
                    for(iODEG=0;iODEG<NODEG;iODEG++)
                    {
                        kNODE=iNODE+iODEG;
                        if(kNODE>=NNODE)
                        {
                            kNODE=0;
                        }
                        for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
                        {
                            n1=kNODE*NDOFN+iDOFN;
                            PE[n1]=PE[n1]+SHAPE[iODEG]*
                                PGASH[iDOFN]*DVOLU;
                        }
                    }
                }
            }
        }
        // Assemble PE into global load vector
        for(iNODE=0;iNODE<NNODE;iNODE++)
        {
            kPOIN=LNODS[kELEM*NNODE+iNODE];
            for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
            {
                kEQNS=NDOFN*kPOIN+iDOFN; // global DOF
                iEVAB=NDOFN*iNODE+iDOFN; // local DOF
                GP[kEQNS]=GP[kEQNS]+PE[iEVAB];
            }
        }
    }
```

```
    free(POSGP); free(WEIGP); free(ELCOD);
    free(EPRES); free(PE);  free(SHAPE);
    free(DSHAP); free(DERGS); free(PGASH);
    return;
}


/*
*********************************************************
* Subroutine INDISBC                                    *
* - Introduce displacement constraints by the penalty *
*   approach                                            *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void INDISBC(double COORD[],int NEQNS,int NVFIX,
             int NOFIX[],int IFPRE[],double PRESC[],
             double GSTIF[],double GLOAD[],int NINTP,
             double IPCOD[],double CRBFI[])
{
    void DUCLEAN();
    void PARSOLU();
    extern int NDIME,NDOFN,NSTRE;
    int ii,jj,kp,iGR,n1;
    double temp,CNST,disv,xp,yp,mdisv,*ups,*sps;
    // Decide penalty parameter CNST
    CNST=0.0;
    for(ii=0;ii<NEQNS;ii++)
    {
        for(jj=0;jj<NEQNS;jj++)
        {
            temp=fabs(GSTIF[ii*NEQNS+jj]);
            if(temp>CNST)
            {
                CNST=temp;
            }
        }
    }
    if((CNST+1)==1)
    {
        printf("Singular coefficient matrix GSTIF!");
        exit(0);
    }
```

```
    CNST=CNST*1000000.0;
    // Modify GSTIF and GLOAD for specified nodal
    // displacements
    for(ii=0;ii<NVFIX;ii++)
    {
        kp=NOFIX[ii];
        xp=COORD[kp*NDIME+0];
        yp=COORD[kp*NDIME+1];
        for(jj=0;jj<NDOFN;jj++)
        {
            iGR=kp*NDOFN+jj;
            // 1 indicates a constrained DOF
            if(IFPRE[ii*NDOFN+jj]==1)
            {
                disv=PRESC[ii*NDOFN+jj];
                ups=(double *)calloc(NDOFN,
                    sizeof(double));
                DUCLEAN(NDOFN,1,ups);
                sps=(double *)calloc(NSTRE,
                    sizeof(double));
                DUCLEAN(NSTRE,1,sps);
                PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,
                    ups,sps);
                // Modify generalised displacement BC
                mdisv=disv-ups[jj];
                n1=iGR*NEQNS+iGR;
                GSTIF[n1]=GSTIF[n1]+CNST;
                GLOAD[iGR]=GLOAD[iGR]+CNST*mdisv;
            }
        }
    }
    return;
}


/*
*******************************************************
* Subroutine FIEDNOD                                  *
* - Generate nodal generalised displacement field     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDNOD(int NPOIN,double COORD[],double ASDIS[],
```

```
              int NINTP,double IPCOD[],double CRBFI[],
              double UPOIN[])
{
    void DUCLEAN();
    void PARSOLU();
    extern int NDIME,NDOFN,NSTRE;
    int ii,jj,NR;
    double xp,yp,*ups,*sps;
    for(ii=0;ii<NPOIN;ii++)
    {
        xp=COORD[ii*NDIME+0];
        yp=COORD[ii*NDIME+1];
        ups=(double *)calloc(NDOFN,sizeof(double));
        DUCLEAN(NDOFN,1,ups);
        sps=(double *)calloc(NSTRE,sizeof(double));
        DUCLEAN(NSTRE,1,sps);
        PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,ups,sps);
        for(jj=0;jj<NDOFN;jj++)
        {
            NR=ii*NDOFN+jj;
            UPOIN[ii*NDOFN+jj]=ASDIS[NR]+ups[jj];
        }
    }
    return;
}


/*
*********************************************************
* Subroutine FIEDCEN                                    *
* -Compute related fields at centroid of each element *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDCEN(int NELEM,int MATNO[],int LNODS[],
              double COORD[],double PROPS[],
              int ELNOD[],double ASDIS[],int NINTP,
              double IPCOD[],double CRBFI[],
              double CECOD[],double UCENP[],
              double SCENP[])
{
    void ELEPARS(); void DUCLEAN(); void HMATRIX();
    void GMATRIX(); void EDISNOD(); void CMATRIX();
```

```
void RIGIDRV(); void TREFFTZ(); void MATMULT();
void PARSOLU();
extern int NTREF,NNODE,NDIME,NDOFN,NSTRE;
int iELEM,kMATS,NEVAB;
double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*d_Ele,*c_Ele,
    c0,*N_SET,*T_SET,*GDISP,*GSTRE,xp,yp,*ups,*sps,
    gxp,gyp;

NEVAB=NNODE*NDOFN;
for(iELEM=0;iELEM<NELEM;iELEM++)
{
    kMATS=MATNO[iELEM];
    // Compute some quantities related to each element
    ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
    DUCLEAN(NNODE,NDIME,ECOOD);
    CenCoord=(double *)calloc(1*NDIME,sizeof(double));
    DUCLEAN(1,NDIME,CenCoord);
    ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
    // Compute H matrix
    EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
    DUCLEAN(NTREF,NTREF,EHMTX);
    HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
    // Compute G matrix
    EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
    DUCLEAN(NTREF,NEVAB,EGMTX);
    GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
    // Nodal displacements
    d_Ele=(double *)calloc(NEVAB,sizeof(double));
    DUCLEAN(NEVAB,1,d_Ele);
    EDISNOD(iELEM,LNODS,ASDIS,d_Ele);
    // Calculate the ce coefficients
    c_Ele=(double *)calloc(NTREF*1,sizeof(double));
    DUCLEAN(NTREF,1,c_Ele);
    CMATRIX(EHMTX,EGMTX,d_Ele,c_Ele);
    // Recover rigid displacement
    RIGIDRV(ECOOD,c_Ele,d_Ele,&c0);
    // Compute Trefftz internal fields at central point
    N_SET=(double *)calloc(NDOFN*NTREF,sizeof(double));
    DUCLEAN(NDOFN,NTREF,N_SET);
    T_SET=(double *)calloc(NSTRE*NTREF,sizeof(double));
    DUCLEAN(NSTRE,NTREF,T_SET);
    xp=0;
    yp=0;
    TREFFTZ(xp,yp,N_SET,T_SET);
    GDISP=(double *)calloc(NDOFN*1,sizeof(double));
```

```
        DUCLEAN(NDOFN,1,GDISP);
        GSTRE=(double *)calloc(NSTRE*1,sizeof(double));
        DUCLEAN(NSTRE,1,GSTRE);
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,GDISP);
        MATMULT(T_SET,c_Ele,NSTRE,NTREF,1,GSTRE);
        // particular solution
        gxp=CenCoord[0]+xp;
        gyp=CenCoord[1]+yp;
        ups=(double *)calloc(NDOFN,sizeof(double));
        DUCLEAN(NDOFN,1,ups);
        sps=(double *)calloc(NSTRE,sizeof(double));
        DUCLEAN(NSTRE,1,sps);
        PARSOLU(gxp,gyp,NINTP,IPCOD,CRBFI,ups,sps);
        // full solution
        UCENP[iELEM*NDOFN+0]=GDISP[0]+c0+ups[0];
        SCENP[iELEM*NSTRE+0]=GSTRE[0]+sps[0];
        SCENP[iELEM*NSTRE+1]=GSTRE[1]+sps[1];
        // Coordinates of commputing point
        CECOD[iELEM*NDIME+0]=gxp;
        CECOD[iELEM*NDIME+1]=gyp;
    }
    free(ECOOD); free(CenCoord); free(EHMTX);
    free(EGMTX); free(d_Ele); free(c_Ele);
    free(N_SET); free(T_SET); free(GDISP);
    free(GSTRE);
}


/*
*********************************************************
* Subroutine PARSOLU                                    *
* -Evaluate approximated particular solutions           *
*  at given point (x,y)                                 *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void PARSOLU(double xp,double yp,int NINTP,
             double IPCOD[],double CRBFI[],
             double ups[],double sps[])
{
    extern int NDIME,NDOFN;
    double up,uxp,uyp,x,y,xj,yj,r,Phi,Phix,Phiy;
    int jT,n;
```

```
    up=0;
    uxp=0;
    uyp=0;
    for(jT=0;jT<NINTP;jT++)
    {
        xj=IPCOD[jT*NDIME+0];
        yj=IPCOD[jT*NDIME+1];
        x=xp-xj;
        y=yp-yj;
        r=sqrt(pow(x,2)+pow(y,2));
        // PS: r^(2n-1)
        n=2;
        Phi =pow(r,(2*n+1))/(pow((2*n+1),2));
        Phix=pow(r,(2*n-1))/(2*n+1)*x;
        Phiy=pow(r,(2*n-1))/(2*n+1)*y;

        up =up +CRBFI[jT]*Phi;
        uxp=uxp+CRBFI[jT]*Phix;
        uyp=uyp+CRBFI[jT]*Phiy;
    }
    ups[0]=up;
    sps[0]=uxp;
    sps[1]=uyp;
    return;
}


/*
*********************************************************
* Subroutine USERFUN                                    *
* - Compute the source value at given point (x,y)       *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void USERFUN(double x, double y,double *b)
{
    *b=0;
    return;
}
```

7.8.2 Plane stress/strain problems

```
/*
*****************************************************
* Mainfunction MAINFUN                               *
* - Call other subroutines                           *
*****************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int NTREF,NTYPE,NNODE,NEDGE,NODEG,NDIME,NDOFN,NSTRE,
NMATS,NPROP,NGAUS;
void main()
{
    void DUCLEAN(); void ITCLEAN(); void INPUTDT();
    void TYPELEM(); void ELEPARS();
    void HMATRIX(); void GMATRIX(); void KMATRIX();
    void ASMSTIF(); void PVECTOR(); void INDISBC();
    void LSSOLVR(); void FIEDNOD(); void FIEDCEN();
    void OPRESUT(); void RBFINTP();
    FILE *fp;
    int NEQNS,NPOIN,NELEM,NVFIX,NPLOD,NDLEG,TNFEG,NEVAB,NINTP,NTVAR;
    int *MATNO,*LNODS,*NOFIX,*IFPRE,*LODPT,*NEASS,*NOPRS,
        *ELNOD;
    double *COORD,*PRESC,*POINT,*PRESS,*PROPS;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*ESTIF,*GSTIF,
        *GLOAD,*UPOIN,*CECOD,*UCENP,*SCENP,*IPCOD,*CRBFI;
    char dummy[201],TITLE[201],file[81];
    int i,j,k,N,n1,n2,iELEM,kMATS;

    printf("*********************************\n");
    printf("        Hybrid Trefftz FEM\n");
    printf("      for plane elastic problems\n");
    printf("      with arbitrary body forces\n");
    printf("      and temperature change\n");
    printf("*********************************\n");
    /** Input data from file **/
    puts("Input file name < dir:fn.txt >: ");
    gets(file);
    if((fp=fopen(file,"r"))==NULL)
    {
        printf("Warning! Can't open input file\n");
        exit(0);
    }
```

```
// basic parameters
fgets(dummy,200,fp);
fgets(TITLE,200,fp);
fgets(dummy,200,fp);

fgets(dummy,200,fp);
fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
    &NPLOD,&NDLEG);

// element connectivity
MATNO=(int *)calloc(NELEM,sizeof(int));
ITCLEAN(NELEM,1,MATNO);
LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
ITCLEAN(NELEM,NNODE,LNODS);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NELEM;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    MATNO[i]=n1-1;
    for(j=0;j<NNODE;j++)
    {
        fscanf(fp,"%d",&n2);
        LNODS[i*NNODE+j]=n2-1;
    }
    fscanf(fp,"\n");
}
// nodal coordinates
COORD=(double *)calloc(NPOIN*NDIME,sizeof(double));
DUCLEAN(NPOIN,NDIME,COORD);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NPOIN;i++)
```

```
{
    fscanf(fp,"%d",&N);
    for(j=0;j<NDIME;j++)
    {
        fscanf(fp,"%lf",&COORD[i*NDIME+j]);
    }
    fscanf(fp,"\n");
}
// specified nodal potential/displacement
NOFIX=(int *)calloc(NVFIX,sizeof(int));
ITCLEAN(NVFIX,1,NOFIX);
IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
ITCLEAN(NVFIX,NDOFN,IFPRE);
PRESC=(double *)calloc(NVFIX*NDOFN,sizeof(double));
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    NOFIX[i]=n1-1;
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
    }
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
    }
    fscanf(fp,"\n");
}
// specified concentrated loads at nodes
fgets(dummy,200,fp);
if(NPLOD>0)
{
    LODPT=(int *)calloc(NPLOD*1,sizeof(int));
    ITCLEAN(NPLOD,1,LODPT);
    POINT=(double *)calloc(NPLOD*NDOFN,
        sizeof(double));
    DUCLEAN(NPLOD,NDOFN,POINT);
    fgets(dummy,200,fp);
    for(i=0;i<NPLOD;i++)
    {
        fscanf(fp,"%d %d",&N,&n1);
        LODPT[i]=n1-1;
```

```
            for(j=0;j<NDOFN;j++)
            {
                fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
            }
            fscanf(fp,"\n");
        }
    }
    // specified distributed edge loads
    fgets(dummy,200,fp);
    if(NDLEG>0)
    {
        NEASS=(int *)calloc(NDLEG*1,sizeof(int));
        ITCLEAN(NDLEG,1,NEASS);
        NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
        ITCLEAN(NDLEG,NODEG,NOPRS);
        TNFEG=NODEG*NDOFN;
        PRESS=(double *)calloc(NDLEG*TNFEG,sizeof(double));
        DUCLEAN(NDLEG,TNFEG,PRESS);
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NPROP;j++)
        {
```

```
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }

    /** Establish local relations of nodes and edges **/
    ELNOD=(int *)calloc(NEDGE*NODEG,sizeof(int));
    ITCLEAN(NEDGE,NODEG,ELNOD);
    TYPELEM(ELNOD);

    /** Compute the coefficient of RBF interpolation **/
    NINTP=NPOIN+NELEM;
    IPCOD=(double *)calloc(NINTP*NDIME,sizeof(double));
    DUCLEAN(NINTP,NDIME,IPCOD);
    NTVAR=NINTP*NDOFN;
    CRBFI=(double *)calloc(NTVAR,sizeof(double));
    DUCLEAN(NTVAR,1,CRBFI);
    RBFINTP(NPOIN,NELEM,COORD,LNODS,PROPS,NINTP,
        IPCOD,CRBFI);

    /** Form stiffness matrix **/
    NEQNS=NPOIN*NDOFN;
    GSTIF=(double *)calloc(NEQNS*NEQNS,sizeof(double));
    DUCLEAN(NEQNS,NEQNS,GSTIF);
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        // Compute some quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,
            sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,
            sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
        HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
        // Compute G matrix
        NEVAB=NNODE*NDOFN;
        EGMTX=(double *)calloc(NTREF*NEVAB,
            sizeof(double));
        DUCLEAN(NTREF,NEVAB,EGMTX);
```

```
        GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
        // Compute element stiffness matrix
        ESTIF=(double *)calloc(NEVAB*NEVAB,
            sizeof(double));
        DUCLEAN(NEVAB,NEVAB,ESTIF);
        KMATRIX(EHMTX,EGMTX,ESTIF);
        // Assemble stiffness matrix
        ASMSTIF(iELEM,NEQNS,LNODS,ESTIF,GSTIF);
        free(EHMTX); free(EGMTX); free(ESTIF);
    }
    // Compute equivalent loads
    GLOAD=(double *)calloc(NEQNS*1,sizeof(double));
    DUCLEAN(NEQNS,1,GLOAD);
    PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,
        NOPRS,PRESS,GLOAD,NINTP,IPCOD,CRBFI);

    // Introduce constrained displacements
    INDISBC(NEQNS,PROPS,COORD,NVFIX,NOFIX,IFPRE,PRESC,
        GSTIF,GLOAD,NINTP,IPCOD,CRBFI);
    // Solve linear system of equations
    LSSOLVR(GSTIF,GLOAD,NEQNS);

    // Output nodal displacement
    UPOIN=(double *)calloc(NPOIN*NDOFN,sizeof(double));
    DUCLEAN(NPOIN,NDOFN,UPOIN);
    FIEDNOD(NPOIN,COORD,GLOAD,NINTP,IPCOD,CRBFI,UPOIN,PROPS);

    // Compute quantities at centroid of each element
    CECOD=(double *)calloc(NELEM*NDIME,sizeof(double));
    DUCLEAN(NELEM,NDIME,CECOD);
    UCENP=(double *)calloc(NELEM*NDOFN,sizeof(double));
    DUCLEAN(NELEM,NDOFN,UCENP);
    SCENP=(double *)calloc(NELEM*NSTRE,sizeof(double));
    DUCLEAN(NELEM,NSTRE,SCENP);
    FIEDCEN(NELEM,MATNO,LNODS,COORD,PROPS,ELNOD,GLOAD,
        NINTP,IPCOD,CRBFI,CECOD,UCENP,SCENP);

    // Output results
    OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,
        NVFIX,NPLOD,NDLEG);

    free(COORD); free(LNODS); free(MATNO);
    free(NOFIX); free(IFPRE); free(PRESC);
    free(PROPS); free(ECOOD); free(CenCoord);
    free(GSTIF); free(GLOAD); free(UPOIN);
```

```
    free(CECOD); free(UCENP); free(SCENP);
    free(IPCOD); free(CRBFI);
    printf("------------ Finished ------------\n");
    return;
}


/*
*********************************************************
* Subroutine RBFINTP                                    *
* -Compute coefficients of RBF interpolation            *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void RBFINTP(int NPOIN,int NELEM,double COORD[],
             int LNODS[],double PROPS[],int NINTP,
             double IPCOD[],double bvect[])
{
    void DUCLEAN();
    void LAS_SVD();
    void USERFUN();
    extern int NDIME,NDOFN,NNODE,NPROP,NTYPE;
    int iP,iD,iE,iN,kP,iT,jT,n1,n,NTVAR,n2;
    double *sum0,*FMATX,xi,yi,xj,yj,r,lnr,temp,
        YOUNG,POISS,THICK,TEXPN,DENST,gm,*GBF,TEM;

    // Material properties
    YOUNG=PROPS[0*NPROP+0];
    POISS=PROPS[0*NPROP+1];
    THICK=PROPS[0*NPROP+2];
    TEXPN=PROPS[0*NPROP+3];
    DENST=PROPS[0*NPROP+4];
    if(NTYPE==1) //plane stress
    {
        gm=YOUNG*TEXPN/(1.0-POISS);
    }
    else if(NTYPE==2)
    {
        gm=YOUNG*TEXPN/(1.0-2*POISS);
    }

    // Generate interpolation points: node+centroid
    for(iP=0;iP<NPOIN;iP++)
```

```
{
    for(iD=0;iD<NDIME;iD++)
    {
        IPCOD[iP*NDIME+iD]=COORD[iP*NDIME+iD];
    }
}
sum0=(double *)calloc(NDIME,sizeof(double));
for(iE=0;iE<NELEM;iE++)
{
    n1=NPOIN+iE;
    // Form nodal coordinates of the given element
    DUCLEAN(1,NDIME,sum0);
    for(iD=0;iD<NDIME;iD++)
    {
        for(iN=0;iN<NNODE;iN++)
        {
            kP=LNODS[iE*NNODE+iN];
            sum0[iD]=sum0[iD]+COORD[kP*NDIME+iD];
        }
        IPCOD[n1*NDIME+iD]=sum0[iD]/NNODE;
    }
}
// Form coefficient matrix
NTVAR=NINTP*NDOFN;
FMATX=(double *)calloc(NTVAR*NTVAR,sizeof(double));
DUCLEAN(NTVAR,NTVAR,FMATX);
for(iT=0;iT<NINTP;iT++)
{
    xi=IPCOD[iT*NDIME+0];
    yi=IPCOD[iT*NDIME+1];
    // compute right-hand term b
    GBF=(double *)calloc(NDOFN*1,sizeof(double));
    DUCLEAN(NDOFN,1,GBF);
    USERFUN(xi,yi,PROPS,GBF,&TEM);
    for(iD=0;iD<NDOFN;iD++)
    {
        bvect[iT*NDOFN+iD]=GBF[iD];
    }
    //bvect[iT*NDOFN+0]=GBF[0];
    //bvect[iT*NDOFN+1]=GBF[1];
    //
    for(jT=0;jT<NINTP;jT++)
    {
        xj=IPCOD[jT*NDIME+0];
        yj=IPCOD[jT*NDIME+1];
```

```
            r=sqrt(pow((xi-xj),2)+pow((yi-yj),2));
            // TPS: r^(2*n)ln(r)
            n=2;
            if((1+r)==1)
            {
                r=0.0;
                lnr=0.0;
            }
            else
            {
                lnr=log(r);
            }
            temp=pow(r,(2*n))*lnr;
            for(iD=0;iD<NDOFN;iD++)
            {
                n1=(iT*NDOFN+iD)*NTVAR+(jT*NDOFN+iD);
                FMATX[n1]=temp;
            }
            /*
            n1=(iT*NDOFN+0)*NTVAR+(jT*NDOFN+0);
            FMATX[n1]=temp;
            n2=(iT*NDOFN+1)*NTVAR+(jT*NDOFN+1);
            FMATX[n2]=temp;
            */
        }
    }
    // Solve linear algebraic equations
    LAS_SVD(FMATX,bvect,NTVAR);
    free(FMATX); free(sum0);
    return;
}


/*
*******************************************************
* Subroutine PVECTOR                                  *
* - Compute effective nodal force                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void PVECTOR(int MATNO[],double PROPS[],int LNODS[],
             double COORD[],int NDLEG,int NEASS[],
             int NOPRS[],double PRESS[],double GP[],
```

```
              int NINTP,double IPCOD[],double CRBFI[])
{
    void GAUSSQU();
    void SHAPFUN();
    void DUCLEAN();
    void PARSOLU();
    void USERFUN();
    extern int NDIME,NDOFN,NGAUS,NPROP,NNODE,NEDGE,
        NODEG,NTYPE,NSTRE;
    double *POSGP,*WEIGP,*ELCOD,*EPRES,*PE,*SHAPE,
        *DSHAP,*CORGS,*DERGS,*PGASH;
    int iDLEG,iNODE,kNODE,iODEG,iDOFN,iDIME,
        jGAUS,kPOIN,kELEM,kMATS,kEQNS,iEVAB,n1,n2,
        NEVAB,TNFEG;
    double EXISP,DVOLU,DS1,DS2,PX,PY,*ups,*sps,
        *GBF,TEM,YOUNG,POISS,THICK,TEXPN,DENST,
        gm,xp,yp;

    // Material properties
    YOUNG=PROPS[0*NPROP+0];
    POISS=PROPS[0*NPROP+1];
    THICK=PROPS[0*NPROP+2];
    TEXPN=PROPS[0*NPROP+3];
    DENST=PROPS[0*NPROP+4];
    if(NTYPE==1) //plane stress
    {
        gm=YOUNG*TEXPN/(1.0-POISS);
    }
    else if(NTYPE==2)
    {
        gm=YOUNG*TEXPN/(1.0-2*POISS);
    }
    // Evaluate equivalent nodal force caused
    // by distributed edge load
    NEVAB=NNODE*NDOFN;
    TNFEG=NODEG*NDOFN;
    // Gaussian point and weight coefficients
    POSGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,POSGP);
    WEIGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,WEIGP);
    GAUSSQU(POSGP,WEIGP);
    for(iDLEG=0;iDLEG<NDLEG;iDLEG++)
    {
        kELEM=NEASS[iDLEG];
```

```
// Material properties
kMATS=MATNO[kELEM];
THICK=PROPS[kMATS*NPROP+2];
// Determine coordinates of nodes on the element edge
ELCOD=(double *)calloc(NODEG*NDIME,sizeof(double));
DUCLEAN(NODEG,NDIME,ELCOD);
for(iODEG=0;iODEG<NODEG;iODEG++)
{
    kPOIN=NOPRS[iDLEG*NODEG+iODEG];
    for(iDIME=0;iDIME<NDIME;iDIME++)
    {
        n1=iODEG*NDIME+iDIME;
        n2=kPOIN*NDIME+iDIME;
        ELCOD[n1]=COORD[n2];
    }
}
// Determine local nodal load intensity
EPRES=(double *)calloc(NODEG*NDOFN,sizeof(double));
DUCLEAN(NODEG,NDOFN,EPRES);
for(iODEG=0;iODEG<NODEG;iODEG++)
{
    for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
    {
        n1=iODEG*NDOFN+iDOFN;
        n2=iDLEG*TNFEG+n1;
        EPRES[n1]=PRESS[n2];
    }
}
// Integration along loaded edge
PE=(double *)calloc(NEVAB,sizeof(double));
DUCLEAN(NEVAB,1,PE);
for(jGAUS=0;jGAUS<NGAUS;jGAUS++)
{
    EXISP=POSGP[jGAUS];
    // shape functions and its derivatives
    SHAPE=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,SHAPE);
    DSHAP=(double *)calloc(NODEG,sizeof(double));
    DUCLEAN(1,NODEG,DSHAP);
    SHAPFUN(EXISP,SHAPE,DSHAP);
    // Coordinates and derivatives of Gauss points
    // x=sum(Ni*xi) and y=sum(Ni*yi)
    // dx/dt=sum(dNi/dt*xi), dy/dt=sum(dNi/dt*yi)
    // DVOLU=sqrt((dx/dt)^2+(dy/dt)^2)
    CORGS=(double *)calloc(NDIME,sizeof(double));
```

```
DUCLEAN(1,NDIME,CORGS);
DERGS=(double *)calloc(NDIME,sizeof(double));
DUCLEAN(1,NDIME,DERGS);
for(iDIME=0;iDIME<NDIME;iDIME++)
{
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        n1=iODEG*NDIME+iDIME;
        CORGS[iDIME]=CORGS[iDIME]+
            SHAPE[iODEG]*ELCOD[n1];
        DERGS[iDIME]=DERGS[iDIME]+
            DSHAP[iODEG]*ELCOD[n1];
    }
}
DVOLU=sqrt(pow(DERGS[0],2.0)+
    pow(DERGS[1],2.0));
// Direction cosine at Gaussian points
DS1= DERGS[1]/DVOLU;
DS2=-DERGS[0]/DVOLU;
// Gaussian integration factor
DVOLU=DVOLU*WEIGP[jGAUS];
if((THICK+1)!=1)
{
    DVOLU=DVOLU*THICK;
}
// Load intensity at Gaussian point
//  pn=sum(Ni*pni),  pt=sum(Ni*pti)
PGASH=(double *)calloc(NDOFN,sizeof(double));
DUCLEAN(NDOFN,1,PGASH);
for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
{
    for(iODEG=0;iODEG<NODEG;iODEG++)
    {
        n1=iODEG*NDOFN+iDOFN;
        PGASH[iDOFN]=PGASH[iDOFN]+
            SHAPE[iODEG]*EPRES[n1];
    }
}
// px=-pn*nx-pt*ny
// py=-pn*ny+pt*nx
PX=-DS1*PGASH[0]-DS2*PGASH[1];
PY=-DS2*PGASH[0]+DS1*PGASH[1];
PGASH[0]=PX;
PGASH[1]=PY;
// Modify boundary tractions at Gaussian points
```

```
// th(i)=t(i)-tp(i)+m*T*n(i)
xp=CORGS[0];
yp=CORGS[1];
// particular solution
ups=(double *)calloc(NDOFN,
    sizeof(double));
DUCLEAN(NDOFN,1,ups);
sps=(double *)calloc(NSTRE,
    sizeof(double));
DUCLEAN(NSTRE,1,sps);
PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,PROPS,
    ups,sps);
// Specified temperature at Gaussian point
GBF=(double *)calloc(NDOFN,
    sizeof(double));
DUCLEAN(NDOFN,1,GBF);
USERFUN(xp,yp,PROPS,GBF,&TEM);
// Evaluate homogeneous tractions
PGASH[0]=PGASH[0]-
    (sps[0]*DS1+sps[2]*DS2)+gm*TEM*DS1;
PGASH[1]=PGASH[1]-
    (sps[2]*DS1+sps[1]*DS2)+gm*TEM*DS2;
// Compute equivalent force PE
for(iNODE=0;iNODE<NNODE;iNODE++)
{
    kPOIN=LNODS[kELEM*NNODE+iNODE];
    if(kPOIN==NOPRS[iDLEG*NODEG+0])
    {
        // iNODE is start node of the
        // loaded edge
        for(iODEG=0;iODEG<NODEG;iODEG++)
        {
            kNODE=iNODE+iODEG;
            if(kNODE>=NNODE)
            {
                kNODE=0;
            }
            for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
            {
                n1=kNODE*NDOFN+iDOFN;
                PE[n1]=PE[n1]+SHAPE[iODEG]*
                    PGASH[iDOFN]*DVOLU;
            }
        }
    }
```

```
            }
        }
        // Assemble PE into global load vector
        for(iNODE=0;iNODE<NNODE;iNODE++)
        {
            kPOIN=LNODS[kELEM*NNODE+iNODE];
            for(iDOFN=0;iDOFN<NDOFN;iDOFN++)
            {
                kEQNS=NDOFN*kPOIN+iDOFN; // global DOF
                iEVAB=NDOFN*iNODE+iDOFN; // local DOF
                GP[kEQNS]=GP[kEQNS]+PE[iEVAB];
            }
        }
    }
    free(POSGP); free(WEIGP); free(ELCOD);
    free(EPRES); free(PE);  free(SHAPE);
    free(DSHAP); free(DERGS); free(PGASH);
    return;
}


/*
*********************************************************
* Subroutine INDISBC                                    *
* - Introduce displacement constraints by the penalty *
*   approach                                            *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void INDISBC(int NEQNS,double PROPS[],double COORD[],
             int NVFIX,int NOFIX[],int IFPRE[],
             double PRESC[],double GSTIF[],
             double GLOAD[],int NINTP,
             double IPCOD[],double CRBFI[])
{
    void DUCLEAN();
    void PARSOLU();
    extern int NDIME,NDOFN,NSTRE;
    int ii,jj,kp,iGR,n1;
    double temp,CNST,disv,xp,yp,mdisv,*ups,*sps;
    // Decide penalty parameter CNST
    CNST=0.0;
    for(ii=0;ii<NEQNS;ii++)
```

```
{
    for(jj=0;jj<NEQNS;jj++)
    {
        temp=fabs(GSTIF[ii*NEQNS+jj]);
        if(temp>CNST)
        {
            CNST=temp;
        }
    }
}
if((CNST+1)==1)
{
    printf("Singular coefficient matrix GSTIF!");
    exit(0);
}
CNST=CNST*1000000.0;
// Modify GSTIF and GLOAD for specified nodal
// displacements
for(ii=0;ii<NVFIX;ii++)
{
    kp=NOFIX[ii];
    xp=COORD[kp*NDIME+0];
    yp=COORD[kp*NDIME+1];
    for(jj=0;jj<NDOFN;jj++)
    {
        iGR=kp*NDOFN+jj;
        // 1 indicates a constrained DOF
        if(IFPRE[ii*NDOFN+jj]==1)
        {
            disv=PRESC[ii*NDOFN+jj];
            ups=(double *)calloc(NDOFN,
                sizeof(double));
            DUCLEAN(NDOFN,1,ups);
            sps=(double *)calloc(NSTRE,
                sizeof(double));
            DUCLEAN(NSTRE,1,sps);
            PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,
                PROPS,ups,sps);
            // Modify generalised displacement BC
            mdisv=disv-ups[jj];
            n1=iGR*NEQNS+iGR;
            GSTIF[n1]=GSTIF[n1]+CNST;
            GLOAD[iGR]=GLOAD[iGR]+CNST*mdisv;
        }
    }
```

```
    }
    return;
}


/*
*********************************************************
* Subroutine FIEDNOD                                    *
* - Generate nodal generalised displacement field       *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDNOD(int NPOIN,double COORD[],double ASDIS[],
             int NINTP,double IPCOD[],double CRBFI[],
             double UPOIN[],double PROPS[])
{
    void DUCLEAN();
    void PARSOLU();
    extern int NDIME,NDOFN,NSTRE;
    int ii,jj,NR;
    double xp,yp,*ups,*sps;
    for(ii=0;ii<NPOIN;ii++)
    {
        xp=COORD[ii*NDIME+0];
        yp=COORD[ii*NDIME+1];
        ups=(double *)calloc(NDOFN,sizeof(double));
        DUCLEAN(NDOFN,1,ups);
        sps=(double *)calloc(NSTRE,sizeof(double));
        DUCLEAN(NSTRE,1,sps);
        PARSOLU(xp,yp,NINTP,IPCOD,CRBFI,PROPS,ups,sps);
        for(jj=0;jj<NDOFN;jj++)
        {
            NR=ii*NDOFN+jj;
            UPOIN[ii*NDOFN+jj]=ASDIS[NR]+ups[jj];
        }
    }
    return;
}


/*
*********************************************************
* Subroutine FIEDCEN                                    *
```

```
* -Compute related fields at centroid of each element *
*****************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDCEN(int NELEM,int MATNO[],int LNODS[],
             double COORD[],double PROPS[],int ELNOD[],
             double ASDIS[],int NINTP,double IPCOD[],
             double CRBFI[],double CECOD[],
             double UCENP[],double SCENP[])
{
    void ELEPARS(); void DUCLEAN(); void HMATRIX();
    void GMATRIX(); void EDISNOD(); void CMATRIX();
    void RIGIDRV(); void TREFFTZ(); void MATMULT();
    void PARSOLU(); void USERFUN();
    extern int NTREF,NNODE,NDIME,NDOFN,NSTRE,NTYPE,NPROP;
    int iELEM,kMATS,NEVAB;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*d_Ele,
        *c_Ele,*c0,*N_SET,*T_SET,*GDISP,*GSTRE,
        xp,yp,YOUNG,POISS,THICK,TEXPN,DENST,
        gm,gxp,gyp,*ups,*sps,*GBF,TEM;

    // Material properties
    YOUNG=PROPS[0*NPROP+0];
    POISS=PROPS[0*NPROP+1];
    THICK=PROPS[0*NPROP+2];
    TEXPN=PROPS[0*NPROP+3];
    DENST=PROPS[0*NPROP+4];
    if(NTYPE==1) //plane stress
    {
        gm=YOUNG*TEXPN/(1.0-POISS);
    }
    else if(NTYPE==2)
    {
        gm=YOUNG*TEXPN/(1.0-2*POISS);
    }

    NEVAB=NNODE*NDOFN;
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        // Compute some quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
```

```
CenCoord=(double *)calloc(1*NDIME,sizeof(double));
DUCLEAN(1,NDIME,CenCoord);
ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
// Compute H matrix
EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
DUCLEAN(NTREF,NTREF,EHMTX);
HMATRIX(ECOOD,ELNOD,kMATS,PROPS,EHMTX);
// Compute G matrix
EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
DUCLEAN(NTREF,NEVAB,EGMTX);
GMATRIX(ECOOD,ELNOD,kMATS,PROPS,EGMTX);
// Nodal displacements
d_Ele=(double *)calloc(NEVAB,sizeof(double));
DUCLEAN(NEVAB,1,d_Ele);
EDISNOD(iELEM,LNODS,ASDIS,d_Ele);
// Calculate the ce coefficients
c_Ele=(double *)calloc(NTREF*1,sizeof(double));
DUCLEAN(NTREF,1,c_Ele);
CMATRIX(EHMTX,EGMTX,d_Ele,c_Ele);
// Recover rigid displacement
c0=(double *)calloc(3*1,sizeof(double));
DUCLEAN(3,1,c0);
RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,PROPS,c0);
// Compute Trefftz internal fields at central point
N_SET=(double *)calloc(NDOFN*NTREF,sizeof(double));
DUCLEAN(NDOFN,NTREF,N_SET);
T_SET=(double *)calloc(NSTRE*NTREF,sizeof(double));
DUCLEAN(NSTRE,NTREF,T_SET);
xp=0;
yp=0;
TREFFTZ(xp,yp,kMATS,PROPS,N_SET,T_SET);
GDISP=(double *)calloc(NDOFN*1,sizeof(double));
DUCLEAN(NDOFN,1,GDISP);
GSTRE=(double *)calloc(NSTRE*1,sizeof(double));
DUCLEAN(NSTRE,1,GSTRE);
MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,GDISP);
MATMULT(T_SET,c_Ele,NSTRE,NTREF,1,GSTRE);
// particular solution
gxp=CenCoord[0]+xp;
gyp=CenCoord[1]+yp;
ups=(double *)calloc(NDOFN,sizeof(double));
DUCLEAN(NDOFN,1,ups);
sps=(double *)calloc(NSTRE,sizeof(double));
DUCLEAN(NSTRE,1,sps);
PARSOLU(gxp,gyp,NINTP,IPCOD,CRBFI,PROPS,ups,sps);
```

```
        // Temperature effect
        GBF=(double *)calloc(NDOFN,sizeof(double));
        DUCLEAN(NDOFN,1,GBF);
        USERFUN(gxp,gyp,PROPS,GBF,&TEM);
        // Full solution
        UCENP[iELEM*NDOFN+0]=GDISP[0]+c0[0]+yp*c0[2]+ups[0];
        UCENP[iELEM*NDOFN+1]=GDISP[1]+c0[1]-xp*c0[2]+ups[1];
        SCENP[iELEM*NSTRE+0]=GSTRE[0]+sps[0]-gm*TEM;
        SCENP[iELEM*NSTRE+1]=GSTRE[1]+sps[1]-gm*TEM;
        SCENP[iELEM*NSTRE+2]=GSTRE[2]+sps[2];
        // Coordinates of commputing point
        CECOD[iELEM*NDIME+0]=gxp;
        CECOD[iELEM*NDIME+1]=gyp;
    }
    free(ECOOD); free(CenCoord); free(EHMTX);
    free(EGMTX); free(d_Ele); free(c_Ele);
    free(N_SET); free(T_SET); free(GDISP);
    free(GSTRE);
}


/*
*********************************************************
* Subroutine PARSOLU                                    *
* -Evaluate approximated particular solutions           *
*  at given point (x,y)                                 *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void PARSOLU(double xp,double yp,int NINTP,
             double IPCOD[],double CRBFI[],
             double PROPS[],double ups[],double sps[])
{
    extern int NDIME,NDOFN,NTYPE,NPROP;
    double x,y,xj,yj,r,lnr,r1,r2,A1,A2,A3,A4,A5,
        temp1,temp2,u11,u12,u22,s111,s112,s221,
        s222,s121,s122,YOUNG,POISS,THICK,TEXPN,DENST,
        mu,G;
    int jT,n;

    // Material properties
    YOUNG=PROPS[0*NPROP+0];
    POISS=PROPS[0*NPROP+1];
```

```
THICK=PROPS[0*NPROP+2];
TEXPN=PROPS[0*NPROP+3];
DENST=PROPS[0*NPROP+4];
G=YOUNG/(2*(1+POISS));
if(NTYPE==1) //plane stress
{
    mu=POISS/(1+POISS);
}
else if(NTYPE==2)
{
    mu=POISS;
}
// particular solutions
for(jT=0;jT<NINTP;jT++)
{
    xj=IPCOD[jT*NDIME+0];
    yj=IPCOD[jT*NDIME+1];
    x=xp-xj;
    y=yp-yj;
    r=sqrt(pow(x,2)+pow(y,2));
    // TPS: r^(2*n)ln(r)
    if(1+r==1)
    {
        r=0;
        lnr=0;
        r1=0;
        r2=0;
    }
    else
    {
        lnr=log(r);
        r1=x/r; // dr/dx
        r2=y/r; // dr/dy
    }
    n=2;
    A1=-(8*pow(n,2)+29*n+27)+8*mu*pow((n+2),2)+
        2*(n+1)*(n+2)*(4*n+7-4*mu*(n+2))*lnr;
    A2=2*(n+1)*(2*n+3)-4*pow((n+1),2)*(n+2)*lnr;
    A3=2*n+3-2*mu*pow((n+2),2)+
        2*(n+1)*(n+2)*(2*mu*n+4*mu-1)*lnr;
    A4=2*(pow(n,2)-2)-4*n*(n+1)*(n+2)*lnr;
    A5=-(2*pow(n,2)+6*n+5)+2*mu*pow((n+2),2)-
        2*(n+1)*(n+2)*(-(2*n+3)+2*mu*(n+2))*lnr;
    temp1=-1.0/(32*G*(1-mu))*pow(r,(2*n+2))
        /(pow((n+1),3)*pow((n+2),2));
```

```
        temp2=-1.0/(8*(1-mu))*pow(r,(2*n+1))
            /(pow((n+1),2)*pow((n+2),2));

        u11=temp1*(A1+A2*r1*r1);
        u12=temp1*(A2*r1*r2);
        u22=temp1*(A1+A2*r2*r2);
        s111=temp2*(A3*r1+A4*r1*r1*r1+A5*(2*r1));
        s112=temp2*(A3*r2+A4*r1*r1*r2);
        s121=temp2*(A4*r1*r2*r1+A5*r2);
        s122=temp2*(A4*r1*r2*r2+A5*r1);
        s221=temp2*(A3*r1+A4*r2*r2*r1);
        s222=temp2*(A3*r2+A4*r2*r2*r2+A5*(2*r2));

        ups[0]=ups[0]+CRBFI[2*jT]*u11 +
            CRBFI[2*jT+1]*u12;
        ups[1]=ups[1]+CRBFI[2*jT]*u12 +
            CRBFI[2*jT+1]*u22;
        sps[0]=sps[0]+CRBFI[2*jT]*s111+
            CRBFI[2*jT+1]*s112;
        sps[1]=sps[1]+CRBFI[2*jT]*s221+
            CRBFI[2*jT+1]*s222;
        sps[2]=sps[2]+CRBFI[2*jT]*s121+
            CRBFI[2*jT+1]*s122;
    }
    return;
}


/*
*******************************************************
* Subroutine USERFUN                                  *
* - Compute the source value at given point (x,y)     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void USERFUN(double x, double y,double PROPS[],
             double GB[],double *T)
{
    extern int NPROP,NTYPE;
    double YOUNG,POISS,THICK,TEXPN,DENST,
        gm,BF1,BF2,DT1,DT2;

    // Material properties
```

```
    YOUNG=PROPS[0*NPROP+0];
    POISS=PROPS[0*NPROP+1];
    THICK=PROPS[0*NPROP+2];
    TEXPN=PROPS[0*NPROP+3];
    DENST=PROPS[0*NPROP+4];
    if(NTYPE==1) //plane stress
    {
        gm=YOUNG*TEXPN/(1.0-POISS);
    }
    else if(NTYPE==2)
    {
        gm=YOUNG*TEXPN/(1.0-2*POISS);
    }
    // Body forces
    BF1=0.0; // bx
    BF2=0.0; // by
    // Temperature change
    *T= 0.0;  // T
    DT1=0.0;  // dT/dx
    DT2=0.0;  // dT/dy
    // Generalised body forces
    GB[0]=BF1-gm*DT1;
    GB[1]=BF2-gm*DT2;
    return;
}
```

7.9 Numerical examples

To illustrate applications of the proposed approach in dealing with domain integrals induced by internal source fields in potential problems or by generalised body forces in plane stress/stain cases, four numerical examples are considered in this section. Moreover, to simplify the process of generating RBF interpolation points and reduce the time required to prepare input data, the nodes and central points of each element are chosen as RBF interpolation points.

7.9.1 Poisson's problems

Example 7.1 Saint Venant torsion of a prismatic beam
Generally the Saint Venant torsion of a prismatic beam is governed by a partial dif-

ferential equation of the type

$$\frac{\partial}{\partial X_1}\left(\frac{1}{G}\frac{\partial u}{\partial X_1}\right)+\frac{\partial}{\partial X_2}\left(\frac{1}{G}\frac{\partial u}{\partial X_2}\right)=-2\theta \tag{7.94}$$

where G is the shear modules and θ is the rate of twist. The variable u is the stress function defined as

$$\tau_{13}=\frac{\partial u}{\partial X_2}, \qquad \tau_{23}=-\frac{\partial u}{\partial X_1} \tag{7.95}$$

with τ_{13} and τ_{23} being two shear stresses.

The boundary condition is

$$u=0 \quad \text{on } \Gamma \tag{7.96}$$

Consider now a homogeneous material for which Eq. (7.94) can be rewritten as

$$\frac{\partial^2 u}{\partial X_1^2}+\frac{\partial^2 u}{\partial X_2^2}=-2 \tag{7.97}$$

in which $G\theta$ is taken as unity for simplicity. In our analysis, the cross-section of the beam is assumed to be an ellipse, defined by the equation

$$\frac{X_1^2}{a^2}+\frac{X_2^2}{b^2}=1 \tag{7.98}$$

and the corresponding exact solution of the problem has been given in Ref. [21]

$$u=\frac{a^2b^2}{a^2+b^2}\left(1-\frac{X_1^2}{a^2}-\frac{X_2^2}{b^2}\right) \tag{7.99}$$

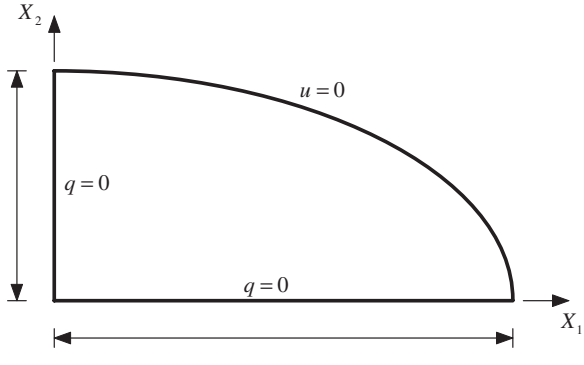


FIGURE 7.7
Mathematical model of Saint Venant torsion of an elliptic beam

Owing to the symmetry of the problem, only one quarter of the solution domain is taken into consideration (see Figure 7.7), with flux equal to zero prescribed along the symmetry axes.

In our analysis, eight quadratic elements as shown in Figure 7.8 are used to model the domain under consideration. The number of terms of T-complete functions is chosen as eight and the PS-RBF r^3 is employed for RBF interpolation. The results for stress function u presented in Figure 7.8 show that the solution for $a/b = 2/1$ from the T-element approach with radial basis functions interpolation is in good agreement with the exact solution. Additionally, we list the numerical results for shear stresses τ_{13} and τ_{23} at central points of each element in Table 7.3 and compare them with exact results. Good accuracy for derivative quantities is achieved.

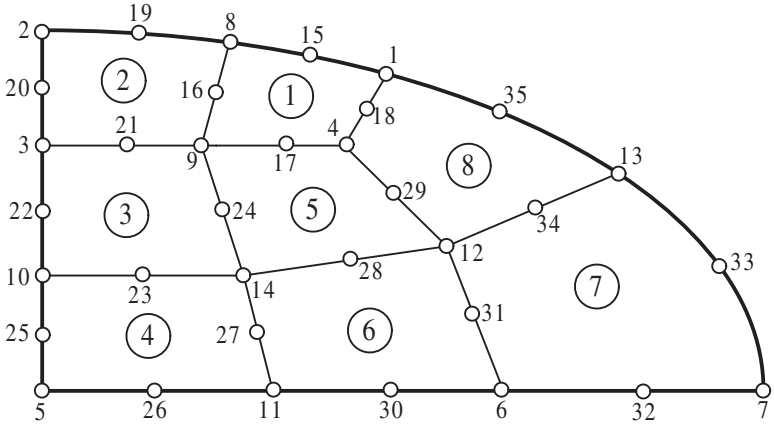


FIGURE 7.8
Configuration of quadratic element meshes

Example 7.2 Two-dimensional Poisson's problem with rectangular domain
Consider a classic Poisson's problem whose domain is a rectangle of size 1×0.4 (see Figure 7.10). The source function is $b = -X_1$. The upper and bottom boundary are assumed to be insulated, whereas Dirichlet boundary conditions apply on the remaining boundaries.

The exact solution of this problem is

$$u = \frac{7}{6} - \frac{1}{6}X_1^3 \tag{7.100}$$

In this example, the rectangular domain is divided into four elements with quadratic frame functions (see Figure 7.11). The number of terms of T-complete functions is

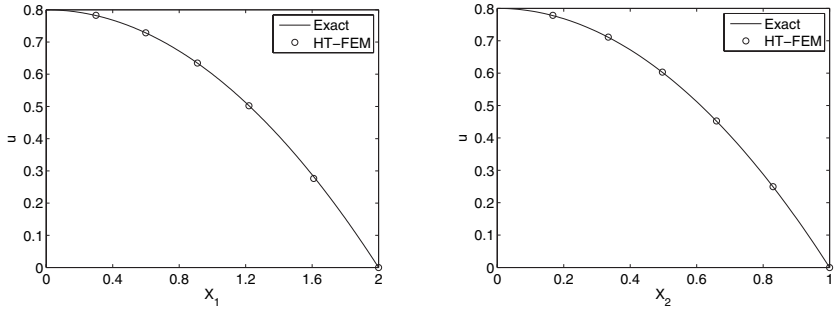


FIGURE 7.9
Comparison of HT-FEM results for stress function u with the exact results along the X_1-axis (left) and X_2-axis (right)

TABLE 7.3
Comparison of HT-FEM results of shear stresses with exact results

Coordinates	τ_{13}		τ_{23}	
	HT-FEM	Exact	HT-FEM	Exact
(0.7612, 0.7739)	−1.2383	−1.2382	0.3051	0.3045
(0.2703, 0.8181)	−1.3093	−1.3090	0.1091	0.1081
(0.2654, 0.4969)	−0.7954	−0.7950	0.1072	0.1061
(0.2881, 0.1696)	−0.2717	−0.2714	0.1167	0.1152
(0.7520, 0.5095)	−0.8156	−0.8153	0.3016	0.3008
(0.8554, 0.1880)	−0.3013	−0.3008	0.3432	0.3422
(1.4566, 0.2724)	−0.4243	−0.4358	0.5853	0.5826
(1.1331, 0.6370)	−1.0189	−1.0192	0.4517	0.4532

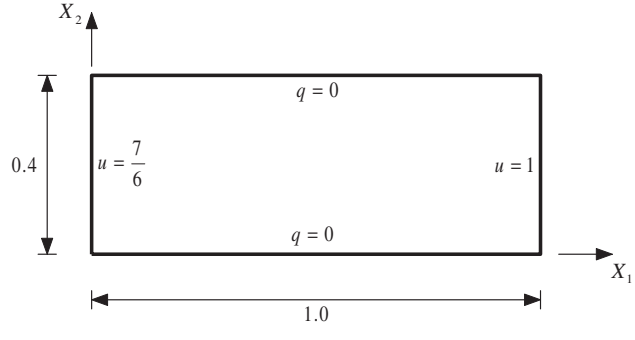


FIGURE 7.10
Geometry of the rectangular domain and specified boundary conditions

chosen as eight and the PS-RBF r^3 is again employed in our computation. The numerical results for potential distribution along $X_2 = 0.2$ are displayed in Figure 7.12. It is evident that the results obtained from HT-FEM are in good agreement with the analytical results, even though only four elements are used in the computation.

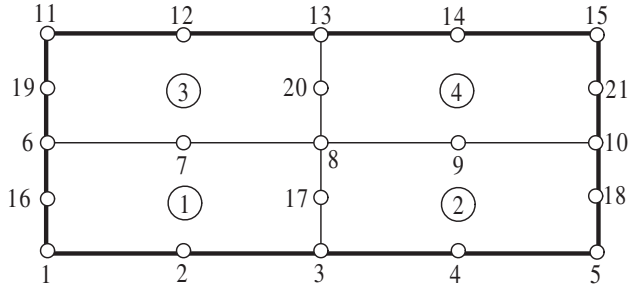


FIGURE 7.11
Configuration of regular element division

7.9.2 Plane stress/strain problems

Example 7.3 Long beam under gravity
As the third example, we consider a long beam with rectangular cross-section subjected to its self-weight. The geometry of the rectangular cross-section and the corresponding boundary conditions are shown in Figure 7.13, in which g denotes the gravity accelerator. This means that constant body forces are taken into consideration, that is, $b_1 = 0$, $b_2 = -\rho g$, ρ is the density of material.

The problem can be viewed as a plane strain problem, and the analytical solutions of displacements and stresses are given by

$$u_1 = 0, \quad u_2 = -\frac{(1+\nu)(1-2\nu)\rho g}{2E(1-\nu)}\left[a^2 - (X_2 - a)^2\right] \tag{7.101}$$

and

$$\sigma_1 = \frac{\rho g \nu}{1-\nu}(X_2 - a), \quad \sigma_2 = \rho g (X_2 - a), \quad \sigma_{12} = 0 \tag{7.102}$$

which can be used to assess the numerical accuracy of the HT-FEM results.

In the computation, let $a = 20$, $E = 1000$, $\nu = 0.25$, $\alpha = 0.001$ and $\rho g = 10$. Four 8-node quadratic elements are used to model the entire square cross-section domain (see Figure 7.14). The number of terms of T-complete functions is chosen as 15 to satisfy the requirement of minimal number of Trefftz functions defined in Eq. (1.12) ($2 \times 8 - 3 = 13$ in this example) and the TPS-RBF $r^4 \ln r$ is employed in our computation. The numerical results for the distribution of nodal displacement

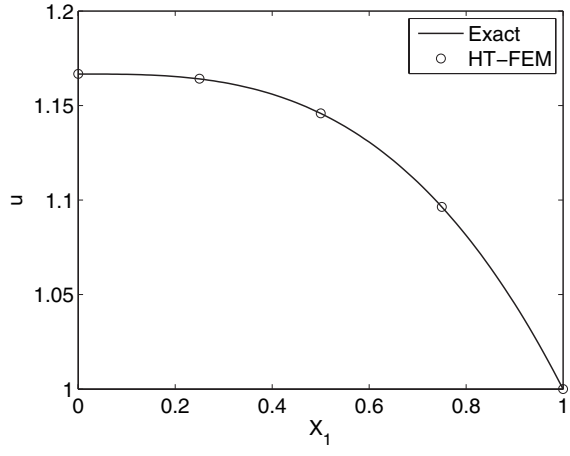


FIGURE 7.12
Distribution of potential field u along the line $X_2 = 0.2$

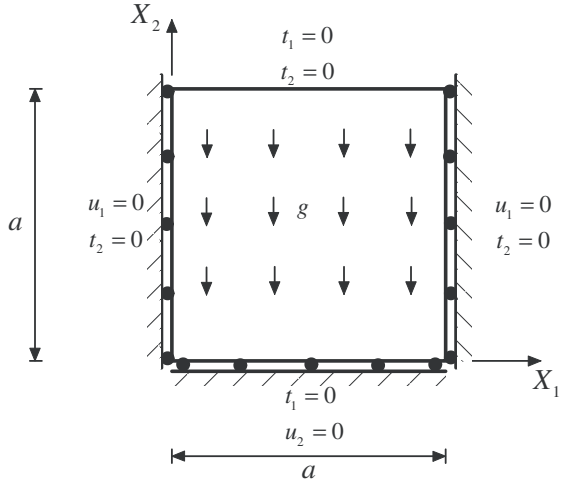


FIGURE 7.13
Long square cross-section beam under gravity

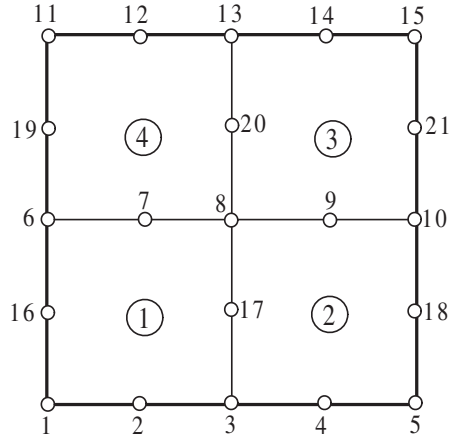


FIGURE 7.14
Mesh configuration of square cross-section beam

component u_2 along $X_1 = 10$ are displayed in Figure 7.15, from which we see that a good agreement is achieved between the analytical solutions and numerical results. Additionally, the stress results at element central points are also compared with the exact solutions in Table 7.4, from which it is clear that the numerical results obtained from HT-FEM with RBF interpolation again show the good accuracy of the results at non-nodal positions. The independence of the results for spatial variable X_1 is also evident, which is consistent with the analytical solution.

TABLE 7.4
Comparison of stress components between numerical and exact solutions

Position	σ_1		σ_2	
	HT-FEM	EXACT	HT-FEM	EXACT
(5, 5)	−49.98	−50	−149.92	−150
(15, 5)	−49.98	−50	−149.92	−150
(15, 15)	−16.64	−16.67	−49.93	−50
(5, 15)	−16.64	−16.67	−49.93	−50

Example 7.4 Circular cylinder with axisymmetric temperature change
The last example considered is a long circular cylinder with axisymmetric tempera-

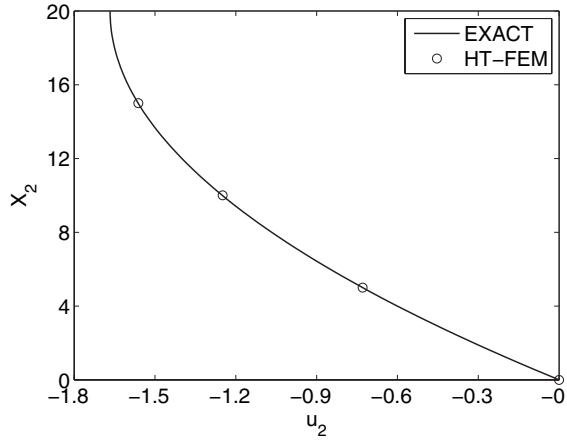


FIGURE 7.15
Distribution of nodal displacement component u_2 along $X_1 = 10$

ture change in the domain. The configurations of geometry and symmetrical conditions of the quarter part and the assumed temperature distribution ϑ with logarithmic radial variation are shown in Figure 7.16. Mechanically, both inside and outside surfaces are assumed to be free from traction. Under the assumption of plane strain, the analytical solutions for stress components without body forces are given as [21]

$$\begin{cases} \sigma_r = -\dfrac{E\alpha\vartheta_0}{2(1-\nu)}\left(\dfrac{\ln\frac{r_o}{r}}{\ln\frac{r_o}{r_i}} - \dfrac{\frac{r_o^2}{r^2}-1}{\frac{r_o^2}{r_i^2}-1}\right) \\ \sigma_\theta = -\dfrac{E\alpha\vartheta_0}{2(1-\nu)}\left(\dfrac{\ln\frac{r_o}{r}-1}{\ln\frac{r_o}{r_i}} + \dfrac{\frac{r_o^2}{r^2}+1}{\frac{r_o^2}{r_i^2}-1}\right) \end{cases} \tag{7.103}$$

In our computation, let $r_i = 5$, $r_o = 20$, $E = 1000$, $\nu = 0.3$, $\alpha = 0.001$, and $\vartheta_0 = 10$ are assumed. Here 16 8-node elements are used to model a quarter of the cylinder as shown in Figure 7.17. The radial and circumferential thermal stresses at all element centres are listed in Figure 7.18 and Figure 7.19 and compared with the theoretical values. It is found from the two figures that the radial stress displays greater error close to the outside surface, whereas the hoop stress shows excellent agreement with the theoretical solutions. Certainly, improvement in determination of the radial stress can be effected by using more elements along the radial direction. Additionally, the nodal radial displacement distribution is also provided in Figure 7.20 for reference.

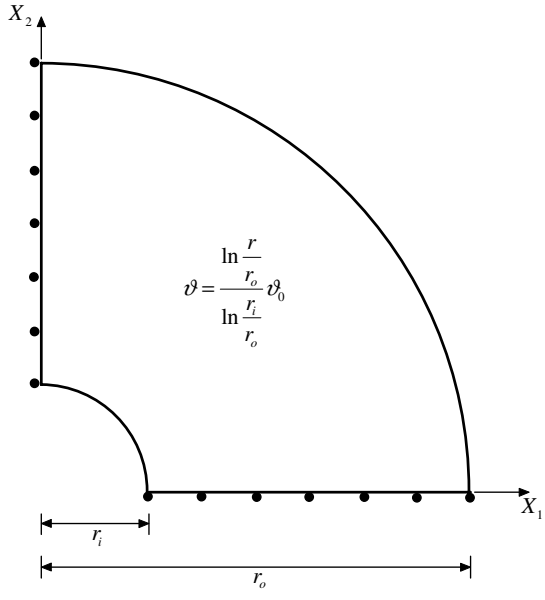


FIGURE 7.16
Configuration of long cylinder with axisymmetric temperature change

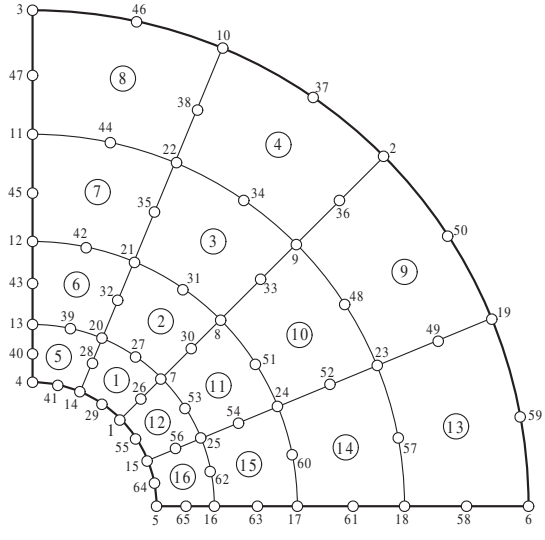


FIGURE 7.17
Mesh subdivision of a quarter of circular cylinder

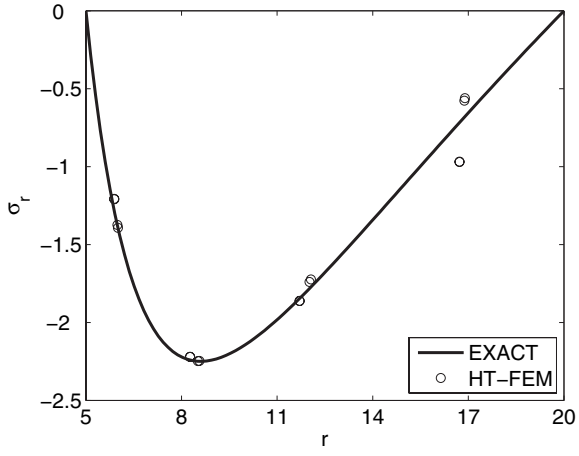


FIGURE 7.18
Radial stress distribution due to radial thermal load case

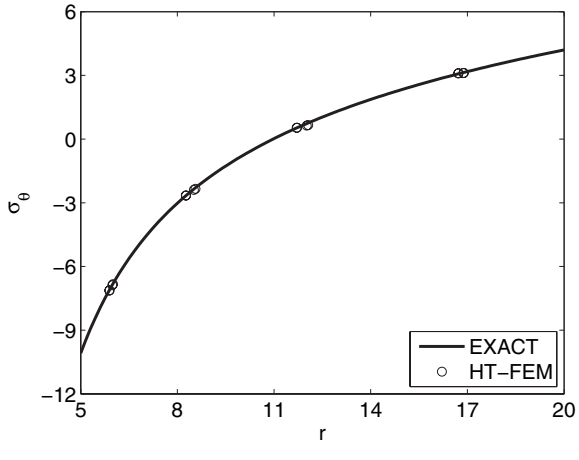


FIGURE 7.19
Hoop stress distribution due to radial thermal load case

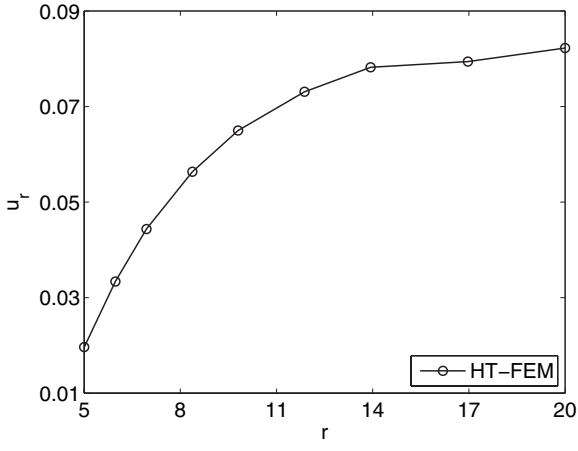


FIGURE 7.20
Radial displacement distribution due to radial temperature load case

References

[1] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*. Southampton: WIT Press.

[2] Cheng AHD, Chen CS, Golberg MA, Rashed YF (2001), BEM for thermoelasticity and elasticity with body force – a revisit. *Eng Anal Bound Elem*, **25**: 377-387.

[3] Park KH (2002), A BEM formulation for axisymmetric elasticity with arbitrary body force using particular integrals. *Comput Struct*, **80**: 2507-2514.

[4] Tran-Cong, T, Mai-Duy N, Phan-Thien N (2002), BEM-RBF approach for viscoelastic flow analysis. *Eng Anal Bound Elem*, **26**: 757-762.

[5] Bridges TR, Wrobel LC (1996), A dual reciprocity formulation for elasticity problems with body forces using augmented thin plate splines. *Commun Numer Meth En*, **12**: 209-220.

[6] Partridge PW, Brebbia CA, Wrobel LC (1992), *The Dual Reciprocity Boundary Element Method*. Southampton: Computational Mechanics Publications.

[7] Golberg MA, Chen CS, Bowman H (1999), Some recent results and proposals for the use of radial basis functions in the BEM. *Eng Anal Bound Elem*, **23**: 285-296.

[8] Golberg MA, Chen CS, Bowman H, Power H (1998), Some comments on the

use of radial basis functions in the dual reciprocity method. *Comput Mech*, **21**: 141-148.

[9] Tiago CM, Leitao VMA (2006), Application of radial basis functions to linear and nonlinear structural analysis problems. *Comput Math Appl*, **51**:1311-1334.

[10] Cho KB, Wang BH (1996), Radial basis function based on adaptive fuzzy systems and their applications to system identification and prediction. *Fuzzy Set Syst*, **83**: 325-339.

[11] Arad N, Dyn N, Reisfeld D, Yeshurun Y (1994), Image warping by radial basis functions: application to facial expressions. *CVGIP-Graphical Models and Image Processing*, **56**: 161-172.

[12] Girosi F (1992), Some extensions of radial basis functions and their applications in artificial intelligence. *Comput Math Appl*, **24**: 61-80.

[13] Fornefett M, Rohr K, Stiehl HS (2001), Radial basis functions with compact support for elastic registration of medical images. *Image Vision Comput*, **19**: 87-96.

[14] Wendland H (1998), Error estimates for interpolation by compactly supported radial basis functions of minimal degree. *J Approx Theory*, **93**: 258-272.

[15] Floater MS, Iske A (1996), Multistep scattered data interpolation using compactly supported radial basis functions. *J Comput Appl Math*, **73**: 65-78.

[16] Wendland H (1995), Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Adv Comput Math*, **4**: 389-396.

[17] Li J, Hon YC, Chen CS (2002), Numerical comparisons of two meshless methods using radial basis functions. *Eng Anal Bound Elem*, **26**: 205-225.

[18] Larsson E, Fornberg B (2003), A numerical study of some radial basis function based solution methods for elliptic PDEs. *Comput Math Appl*, **46**: 891-902.

[19] Golberg MA, Chen CS, Karur SR (1996), Improved multiquadric approximation for partial differential equations. *Eng Anal Bound Elem*, **18**: 9-17.

[20] Fung YC (1965), *Foundation of Solid Mechanics*. Englewood: Prentice-Hall.

[21] Timoshenko SP, Goodier JN (1951), *Theory of Elasticity* (2nd edition). New York: McGraw-Hill.

8

Special purpose T-elements

8.1 Introduction

It is well known that singularities induced by local defects such as singular corners, cracks, circular holes, concentrated loads, and so on, can be accurately accounted for in the conventional FE model by way of appropriate local refinement of the element mesh. However, an important advantage of HT-FEM over conventional FEM and BEM is that special T-elements based on special purpose functions can be constructed to handle such problems more efficiently [1 - 5].

In HT-FEM, elements containing local defects (see Figure 8.1) are treated by simply replacing the regular T-complete functions with appropriate special purpose "trial" functions. One common characteristic of such special trial functions is that it is not only the governing partial differential equations which are satisfied exactly, but also some prescribed boundary conditions at a particular portion of the element boundary. This enables various singularities to be specifically taken into consideration without troublesome mesh refinement. Since the whole element formulation remains unchanged (except that now the frame function is defined and the boundary integral is performed at the portion of the element boundary), all that is needed to implement the elements containing such special trial functions is to provide the element subroutine of the regular elements with a library of various optional sets of special purpose functions.

To take full advantage of the HT approach, in this chapter we discuss how special purpose functions satisfying both the governing equations and the prescribed boundary conditions can be constructed (Section 8.2), and then the element subroutines are presented (Section 8.3), with a library of optional Trefftz functions for accurate handling of circular holes in plane Laplace problems and plane elasticity problems, which are discussed in Sections 8.4 and 8.5.

8.2 Basic concept of special Trefftz functions

Consider a typical T-element containing a circular hole with radius b as shown in Figure 8.2. Let (X_1, X_2) represent global coordinates and (x_1, x_2) the local coordinates

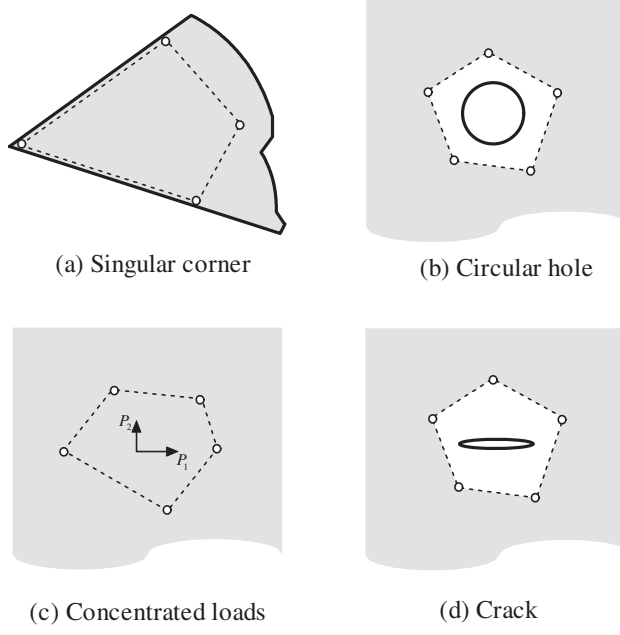


FIGURE 8.1
Special purpose T-elements

to the element containing the hole, centred at the centroid of the element. Generally, such functions which fulfill both the governing equations (plane Laplace equation or plane elasticity equations here) and the prescribed boundary conditions along the hole surface in an infinite plane may be viewed as special Trefftz functions. In the T-element shown in Figure 8.2, the hole surface Γ_{se} is regarded as a particular local portion of the entire element boundary. As a consequence, the full boundary definition of the element in HT formulation (see Chapters 5 and 6) now corresponds to [5, 6]

$$\partial\Omega_e = \Gamma_e \cup \Gamma_{se} \tag{8.1}$$

Since at circular surface Γ_{se} all requisite conditions are satisfied *a priori*, the frame field $\tilde{u}_e$ and the numerical integration involved in evaluation of the auxiliary matrices $\mathbf{H}_e$, $\mathbf{G}_e$ and $\mathbf{g}_e$ are now confined to the remaining portion of the element boundary. This means that once special Trefftz functions can be made available, a similar procedure to the regular HT element implementation is used to determine all unknowns, with minor modification.

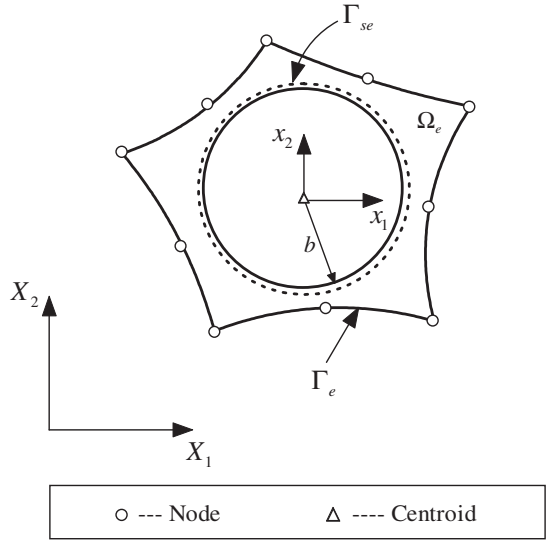


FIGURE 8.2
Generation of special purpose Trefftz function in an infinite plane with a circular hole

8.3 Special purpose elements for potential problems

8.3.1 Trefftz-complete solutions for circular hole elements

The derivation of special purpose functions is usually based on the general solution in an infinite domain. It is well known that the two-dimensional Laplace equation in polar coordinates $(r,\ \theta)$

$$\nabla^2 u = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} = 0 \tag{8.2}$$

has a general solution expressed in the form [5]

$$\begin{aligned} u(r,\theta) = a_0 &+ \sum_{n=1}^{\infty}\left(a_n r^{\lambda_n} + b_n r^{-\lambda_n}\right)\cos(\lambda_n\theta) \\ &+ \sum_{n=1}^{\infty}\left(d_n r^{\lambda_n} + e_n r^{-\lambda_n}\right)\sin(\lambda_n\theta) \end{aligned} \tag{8.3}$$

where

$$r = \sqrt{x_1^2 + x_2^2}, \qquad \theta = \arctan\frac{x_2}{x_1} \tag{8.4}$$

are defined in the coordinates $(x_1,\ x_2)$ local to the element.

Here, we simply review the derivation of special trial function for the case of an element containing a circular hole. Typical trial functions for such elements are found by considering an infinite perforated domain (see Figure 8.2) for which the free boundary condition along the circular hole boundary is assumed as

$$q_r = \frac{\partial u}{\partial r} = 0 \qquad \text{for} \quad r = b \tag{8.5}$$

Differentiating the solution (8.3) and substituting it into the condition (8.5) yields

$$\begin{aligned} \left.\frac{\partial u}{\partial r}\right|_{r=b} &= \sum_{n=1}^{\infty}\left(a_n\lambda_n b^{\lambda_n-1} - b_n\lambda_n b^{-\lambda_n-1}\right)\cos(\lambda_n\theta) \\ &+ \sum_{n=1}^{\infty}\left(d_n\lambda_n b^{\lambda_n-1} - e_n\lambda_n b^{-\lambda_n-1}\right)\sin(\lambda_n\theta) = 0 \end{aligned} \tag{8.6}$$

Noting that Eq. (8.6) holds for any θ, we have

$$\begin{aligned} a_n\lambda_n b^{\lambda_n-1} - b_n\lambda_n b^{-\lambda_n-1} &= 0 \\ d_n\lambda_n b^{\lambda_n-1} - e_n\lambda_n b^{-\lambda_n-1} &= 0 \end{aligned} \tag{8.7}$$

from which it can be observed

$$b_n = a_n b^{2\lambda_n}, \qquad e_n = d_n b^{2\lambda_n} \tag{8.8}$$

Further, from the single value requirement of the potential field in the domain

$$u(r,\theta) = u(r,\theta+2\pi) \tag{8.9}$$

we finally have

$$\lambda_n = n \qquad (n = 1,2,3,\cdots) \tag{8.10}$$

Hence, the final form of the homogeneous solution is

$$u(r,\theta) = a_0 + \sum_{n=1}^{\infty}\left(r^n + b^{2n}r^{-n}\right)\left[a_n\cos(n\theta) + d_n\sin(n\theta)\right] \tag{8.11}$$

from which the internal special Trefftz function defined in

$$u = \sum_{j=1}^{m} N_{ej}c_j \tag{8.12}$$

may be taken as

$$\begin{aligned} N_{2k-1} = &\quad \left(r^k + b^{2k}r^{-k}\right)\cos(k\theta) \\ N_{2k} = &\quad \left(r^k + b^{2k}r^{-k}\right)\sin(k\theta) \end{aligned} \tag{8.13}$$

for $k = 1,2,3,\cdots$.

Making use of the expressions (5.11) and (8.13), the derivatives of T-complete functions can be derived as

$$\begin{aligned} \mathbf{T}_{2k-1} &= \left\{\begin{array}{c} \dfrac{\partial N_{2k-1}}{\partial x_1} \\ \dfrac{\partial N_{2k-1}}{\partial x_2} \end{array}\right\} = \left\{\begin{array}{l} kr^{k-1}\cos\left[(k-1)\,\theta\right] - kb^{2k}r^{-k-1}\cos\left[(k+1)\,\theta\right] \\ -kr^{k-1}\sin\left[(k-1)\,\theta\right] - kb^{2k}r^{-k-1}\sin\left[(k+1)\,\theta\right] \end{array}\right\} \\ \mathbf{T}_{2k} &= \left\{\begin{array}{c} \dfrac{\partial N_{2k}}{\partial x_1} \\ \dfrac{\partial N_{2k}}{\partial x_2} \end{array}\right\} = \left\{\begin{array}{l} kr^{k-1}\sin\left[(k-1)\,\theta\right] - kb^{2k}r^{-k-1}\sin\left[(k+1)\,\theta\right] \\ kr^{k-1}\cos\left[(k-1)\,\theta\right] + kb^{2k}r^{-k-1}\cos\left[(k+1)\,\theta\right] \end{array}\right\} \end{aligned} \tag{8.14}$$

8.4 Special purpose elements for linear elastic problems

8.4.1 Special Trefftz solutions for circular hole elements

Considering an infinite perforated plane, the governing differential equations of linear elastic problems can be rewritten in terms of polar coordinates system as [7]

(a) Equilibrium equations:

$$\begin{aligned} &\frac{\partial \sigma_r}{\partial r}+\frac{1}{r}\frac{\partial \tau_{r\theta}}{\partial \theta}+\frac{\sigma_r-\sigma_\theta}{r}+R_r=0 \\ &\frac{1}{r}\frac{\partial \sigma_\theta}{\partial \theta}+\frac{\partial \tau_{r\theta}}{\partial r}+\frac{2}{r}\tau_{r\theta}+R_\theta=0 \end{aligned} \tag{8.15}$$

(b) Constitutive relations for plane stress:

$$\begin{aligned} \sigma_r &= \frac{E}{1-\nu^2}(\varepsilon_r+\nu\varepsilon_\theta) \\ \sigma_\theta &= \frac{E}{1-\nu^2}(\varepsilon_\theta+\nu\varepsilon_r) \\ \tau_{r\theta} &= G\gamma_{r\theta} \end{aligned} \tag{8.16}$$

(c) Strain-displacement relations:

$$\begin{aligned} \varepsilon_r &= \frac{\partial u_r}{\partial r} \\ \varepsilon_\theta &= \frac{u_r}{r}+\frac{1}{r}\frac{\partial u_\theta}{\partial \theta} \\ \gamma_{r\theta} &= \frac{1}{r}\frac{\partial u_r}{\partial \theta}+\frac{\partial u_\theta}{\partial r}-\frac{u_\theta}{r} \end{aligned} \tag{8.17}$$

where σ_r, σ_θ, $\sigma_{r\theta}$ and ε_r, ε_θ, $\gamma_{r\theta}$ denote the stress and strain components in a polar coordinates system; u_r and u_θ are displacement components along r and θ directions, respectively; R_r and R_θ represent body forces along r and θ directions, respectively. E and ν are Young's modulus and Poisson's ratio, and shear modulus $G=E/[2(1+\nu)]$.

Substitution of Eqs. (8.16) and (8.17) into Eq. (8.15), neglecting body forces, yields the following governing equations in terms of displacement components*:

$$\begin{aligned} &2\frac{\partial^2 u_r}{\partial r^2}+\frac{1-\nu}{r^2}\frac{\partial^2 u_r}{\partial \theta^2}+\frac{1+\nu}{r}\frac{\partial^2 u_\theta}{\partial r\partial \theta}+\frac{2}{r}\frac{\partial u_r}{\partial r}-\frac{3-\nu}{r^2}\frac{\partial u_\theta}{\partial \theta}-\frac{2}{r^2}u_r=0 \\ &\frac{2}{r^2}\frac{\partial^2 u_\theta}{\partial \theta^2}+(1-\nu)\frac{\partial^2 u_\theta}{\partial r^2}+\frac{1+\nu}{r}\frac{\partial^2 u_r}{\partial r\partial \theta}+\frac{1-\nu}{r}\frac{\partial u_\theta}{\partial r}+\frac{3-\nu}{r^2}\frac{\partial u_r}{\partial \theta}-\frac{1-\nu}{r^2}u_\theta=0 \end{aligned} \tag{8.18}$$

Assume that the general solutions to Eq. (8.18) have following form [2]:

$$u_r=f_k(r)\cos k\theta, \qquad u_\theta=g_k(r)\sin k\theta \qquad (k=0,1,2,\cdots) \tag{8.19}$$

*In the expression (8.18), the coefficient factor $E/[2(1-\nu^2)]$ has been omitted.

where $f_k(r)$ and $g_k(r)$ are undetermined functions of spatial variable r. Substituting Eq. (8.19) into Eq. (8.18) yields [1]

$$\begin{aligned}
&\frac{\cos(k\theta)}{r^2}\left[\begin{array}{l} 2r^2\dfrac{\mathrm{d}^2 f_k}{\mathrm{d}r^2}+2r\dfrac{\mathrm{d}f_k}{\mathrm{d}r}-\left(k^2-k^2\nu+2\right)f_k \\ +kr(1+\nu)\dfrac{\mathrm{d}g_k}{\mathrm{d}r}-k(3-\nu)g_k \end{array}\right]=0 \\
&\frac{\sin(k\theta)}{r^2}\left[\begin{array}{l} r^2(1-\nu)\dfrac{\mathrm{d}^2 g_k}{\mathrm{d}r^2}+r(1-\nu)\dfrac{\mathrm{d}g_k}{\mathrm{d}r}-\left(2k^2-\nu+1\right)g_k \\ -kr(1+\nu)\dfrac{\mathrm{d}f_k}{\mathrm{d}r}-k(3-\nu)f_k \end{array}\right]=0
\end{aligned} \tag{8.20}$$

In particular, for the case of $k=0$, we notice that the second equation in Eq. (8.20) holds for any $g(r)$, so we finally have

$$2\left(r^2\frac{\mathrm{d}^2 f_0}{\mathrm{d}r^2}+r\frac{\mathrm{d}f_0}{\mathrm{d}r}-f_0\right)=0 \tag{8.21}$$

which leads to

$$f_0=C_1^0 r+C_2^0 r^{-1} \tag{8.22}$$

Subsequently, for $k\geq 1$, considering that the two equations in Eq. (8.20) should hold for arbitrary r and θ, this fact means that

$$\begin{aligned}
&\left[\begin{array}{l} 2r^2\dfrac{\mathrm{d}^2 f_k}{\mathrm{d}r^2}+2r\dfrac{\mathrm{d}f_k}{\mathrm{d}r}-\left(k^2-k^2\nu+2\right)f_k \\ +kr(1+\nu)\dfrac{\mathrm{d}g_k}{\mathrm{d}r}-k(3-\nu)g_k \end{array}\right]=0 \\
&\left[\begin{array}{l} r^2(1-\nu)\dfrac{\mathrm{d}^2 g_k}{\mathrm{d}r^2}+r(1-\nu)\dfrac{\mathrm{d}g_k}{\mathrm{d}r}-\left(2k^2-\nu+1\right)g_k \\ -kr(1+\nu)\dfrac{\mathrm{d}f_k}{\mathrm{d}r}-k(3-\nu)f_k \end{array}\right]=0
\end{aligned} \tag{8.23}$$

In particular, for $k=1$, we obtain

$$\begin{aligned}
2r^2\frac{\mathrm{d}^2 f_1}{\mathrm{d}r^2}+2r\frac{\mathrm{d}f_1}{\mathrm{d}r}-(3-\nu)f_k+r(1+\nu)\frac{\mathrm{d}g_1}{\mathrm{d}r}-(3-\nu)g_1=0 \\
r^2(1-\nu)\frac{\mathrm{d}^2 g_1}{\mathrm{d}r^2}+r(1-\nu)\frac{\mathrm{d}g_1}{\mathrm{d}r}-(3-\nu)g_1-r(1+\nu)\frac{\mathrm{d}f_1}{\mathrm{d}r}-(3-\nu)f_1=0
\end{aligned} \tag{8.24}$$

which produces the general solutions

$$\begin{aligned}
f_1&=C_1^1 r^2+C_2^1 r^{-2}+C_3^1+C_4^1\ln r \\
g_1&=\frac{5+\nu}{1-3\nu}C_1^1 r^2+C_2^1 r^{-2}-C_3^1-\left(\frac{1+\nu}{3-\nu}+\ln r\right)C_4^1
\end{aligned} \tag{8.25}$$

and for the case $k \geq 2$, we obtain similarly

$$\begin{aligned} f_k &= C_1^k r^{1+k} + C_2^k r^{-1-k} + C_3^k r^{1-k} + C_4^k r^{-1+k} \\ g_k &= \frac{4+k(1+\nu)}{2(1-\nu)-k(1+\nu)} C_1^k r^{1+k} + C_2^k r^{-1-k} \\ &\quad + \frac{k(1+\nu)-4}{2(1-\nu)+k(1+\nu)} C_3^k r^{1-k} - C_4^k r^{-1+k} \end{aligned} \tag{8.26}$$

It should be mentioned that the functions $g_1, f_2,$ and g_2 presented here are different from those appearing on p. 1454 of Ref. [2]. Consequently, the corresponding functions derived from $g_1, f_2,$ and g_2 are also different from those on p. 1470 of Ref. [2]. In the above equations, C_1^k, C_2^k, C_3^k, C_4^k $(k = 0, 1, 2, \cdots)$ are undetermined constants, and half of them can be eliminated by fulfilling (for any k separately) the homogeneous boundary conditions at the hole boundary $r = b$

$$\sigma_r|_{r=b} = \tau_{r\theta}|_{r=b} = 0 \tag{8.27}$$

The remaining constants become the undetermined coefficients c_j of the special-purpose expansion set. For example, for the case of $k \geq 2$, the application of boundary conditions (8.27) yields the following displacement fields:

$$\begin{Bmatrix} u_r \\ u_\theta \end{Bmatrix} = \begin{bmatrix} F_2 \cos k\theta & F_3 \cos k\theta \\ G_2 \sin k\theta & G_3 \sin k\theta \end{bmatrix} \begin{Bmatrix} C_1 \\ C_2 \end{Bmatrix} \tag{8.28}$$

where F_2, F_3, G_2, G_3 are known functions of variable r only and C_1, C_2 are undetermined coefficients.

An alternative set of solutions can be found by assuming

$$u_r = f_k(r) \sin k\theta, \quad u_\theta = g_k(r) \cos k\theta \quad (k = 0, 1, 2, \cdots) \tag{8.29}$$

which can produce the following expressions by substituting Eq. (8.29) into (8.18):

$$\begin{aligned} &\begin{bmatrix} 2\dfrac{\mathrm{d}^2 f_k}{\mathrm{d}r^2} + 2r\dfrac{\mathrm{d}f_k}{\mathrm{d}r} - \left(k^2 - k^2\nu + 2\right) f_k \\ -kr(1+\nu)\dfrac{\mathrm{d}g_k}{\mathrm{d}r} + k(3-\nu) g_k \end{bmatrix} = 0 \\ &\begin{bmatrix} r^2(1-\nu)\dfrac{\mathrm{d}^2 g_k}{\mathrm{d}r^2} + r(1-\nu)\dfrac{\mathrm{d}g_k}{\mathrm{d}r} - \left(2k^2 - \nu + 1\right) g_k \\ +kr(1+\nu)\dfrac{\mathrm{d}f_k}{\mathrm{d}r} + k(3-\nu) f_k \end{bmatrix} = 0 \end{aligned} \tag{8.30}$$

A similar procedure can be performed by setting the parameter k in Eq. (8.30) to be 0, 1 and $k \geq 2$ in turn, and we can finally obtain

$$g_0 = C_3^0 r + C_4^0 r^{-1} \tag{8.31}$$

$$f_1 = C_1^1 r^2 + C_2^1 r^{-2} + C_3^1 + C_4^1 \ln r$$

$$g_1 = -\frac{5+\nu}{1-3\nu}C_1^1 r^2 - C_2^1 r^{-2} + C_3^1 + \left(\frac{1+\nu}{3-\nu} + \ln r\right) C_4^1 \tag{8.32}$$

$$\begin{aligned} f_k &= C_1^k r^{1+k} + C_2^k r^{-1-k} + C_3^k r^{1-k} + C_4^k r^{-1+k} \\ g_k &= -\frac{4+k(1+\nu)}{2(1-\nu)-k(1+\nu)} C_1^k r^{1+k} - C_2^k r^{-1-k} \\ &\quad - \frac{k(1+\nu)-4}{2(1-\nu)+k(1+\nu)} C_3^k r^{1-k} + C_4^k r^{-1+k} \end{aligned} \tag{8.33}$$

from which we can derive different sets of solutions with different k. Making use of the free boundary conditions (8.27), we can eliminate half of the unknown coefficients, and the remaining half is regarded as the undetermined coefficients of Trefftz interpolation within a typical HT-FEM element. For example, for the case of $k \geq 2$, the following displacement fields are obtained:

$$\begin{Bmatrix} u_r \\ u_\theta \end{Bmatrix} = \begin{bmatrix} F_2 \sin k\theta & F_3 \sin k\theta \\ -G_2 \cos k\theta & -G_3 \cos k\theta \end{bmatrix} \begin{Bmatrix} C_1 \\ C_2 \end{Bmatrix} \tag{8.34}$$

The complete displacement homogeneous solutions $\hat{\mathbf{N}}_{jk}^* = \left\{\hat{N}_{rjk} \;\; \hat{N}_{\theta jk}\right\}^{\mathrm{T}}$ constructed from the two sets of displacement fields (8.19) and (8.29) can be written as:

$$\hat{\mathbf{N}}_{10}^* = \begin{Bmatrix} \dfrac{(1+\nu)}{1-\nu} b^2 r^{-1} + r \\ 0 \end{Bmatrix}, \quad \hat{\mathbf{N}}_{20}^* = \begin{Bmatrix} 0 \\ r \end{Bmatrix}$$

for $k = 0$, and

$$\hat{\mathbf{N}}_{11}^* = \begin{Bmatrix} F_1 \cos\theta \\ G_1 \sin\theta \end{Bmatrix}, \quad \hat{\mathbf{N}}_{21}^* = \begin{Bmatrix} F_1 \sin\theta \\ -G_1 \cos\theta \end{Bmatrix}$$

$$\hat{\mathbf{N}}_{31}^* = \begin{Bmatrix} \cos\theta \\ -\sin\theta \end{Bmatrix}, \quad \hat{\mathbf{N}}_{41}^* = \begin{Bmatrix} \sin\theta \\ \cos\theta \end{Bmatrix}$$

for $k = 1$, and

$$\hat{\mathbf{N}}_{1k}^* = \begin{Bmatrix} F_2 \cos k\theta \\ G_2 \sin k\theta \end{Bmatrix}, \quad \hat{\mathbf{N}}_{2k}^* = \begin{Bmatrix} F_2 \sin k\theta \\ -G_2 \cos k\theta \end{Bmatrix}$$

$$\hat{\mathbf{N}}_{3k}^* = \begin{Bmatrix} F_3 \cos k\theta \\ G_3 \sin k\theta \end{Bmatrix}, \quad \hat{\mathbf{N}}_{4k}^* = \begin{Bmatrix} F_3 \sin k\theta \\ -G_3 \cos k\theta \end{Bmatrix}$$

for $k \geq 2$, where

$$\begin{aligned} F_1 &= A_1 r^{-2} + r^2 \\ G_1 &= A_1 r^{-2} + A_2 r^2 \\ F_2 &= B_1 r^{1-k} + B_2 r^{k-1} + r^{1+k} \\ G_2 &= B_5 r^{1-k} - B_2 r^{k-1} + B_6 r^{k+1} \\ F_3 &= B_3 r^{1-k} + B_4 r^{k-1} + r^{-k-1} \\ G_3 &= B_7 r^{1-k} - B_4 r^{k-1} + r^{-k-1} \end{aligned}$$

with

$$A_1 = \frac{1+\nu}{1-3\nu}b^4, \qquad A_2 = \frac{5+\nu}{1-3\nu}$$
$$B_1 = \frac{2(1-\nu)+k(1+\nu)}{(k-1)[2(1-\nu)-k(1+\nu)]}b^{2k}, \quad B_2 = \frac{k^2(1+\nu)}{(k-1)[2(1-\nu)-k(1+\nu)]}b^2$$
$$B_3 = \frac{2(1-\nu)+k(1+\nu)}{(1-k)(1+\nu)}b^{-2}, \qquad B_4 = \frac{1}{1-k}b^{-2k}$$
$$B_5 = \frac{k(1+\nu)-4}{(k-1)[2(1-\nu)-k(1+\nu)]}b^{2k}, \quad B_6 = \frac{k(1+\nu)+4}{2(1-\nu)-k(1+\nu)}$$
$$B_7 = \frac{k(1+\nu)-4}{(1-k)(1+\nu)}b^{-2}$$

Then, the corresponding stress homogeneous solutions $\hat{\mathbf{T}}^*_{jk} = \left\{\hat{T}_{rjk} \;\; \hat{T}_{\theta jk} \;\; \hat{T}_{r\theta jk}\right\}^{\mathrm{T}}$ in polar coordinates can be obtained by substituting the displacement fields into Eqs. (8.17) and (8.16):

$$\hat{\mathbf{T}}^*_{10} = \begin{Bmatrix} \dfrac{E}{1-\nu} - \dfrac{E}{1+\nu}B_0 r^{-2} \\ \dfrac{E}{1-\nu} + \dfrac{E}{1+\nu}B_0 r^{-2} \\ 0 \end{Bmatrix}, \quad \hat{\mathbf{T}}^*_{20} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}$$

for $k = 0$, and

$$\hat{\mathbf{T}}^*_{11} = \begin{Bmatrix} \tilde{F}_1 \cos\theta \\ \tilde{G}_1 \cos\theta \\ \tilde{H}_1 \sin\theta \end{Bmatrix}, \quad \hat{\mathbf{T}}^*_{21} = \begin{Bmatrix} \tilde{F}_1 \sin\theta \\ \tilde{G}_1 \sin\theta \\ -\tilde{H}_1 \cos\theta \end{Bmatrix}$$
$$\hat{\mathbf{T}}^*_{31} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}, \qquad \hat{\mathbf{T}}^*_{41} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}$$

for $k = 1$, and

$$\hat{\mathbf{T}}^*_{1k} = \begin{Bmatrix} \tilde{F}_2 \cos k\theta \\ \tilde{G}_2 \cos k\theta \\ \tilde{H}_2 \sin k\theta \end{Bmatrix}, \quad \hat{\mathbf{T}}^*_{2k} = \begin{Bmatrix} \tilde{F}_2 \sin k\theta \\ \tilde{G}_2 \sin k\theta \\ -\tilde{H}_2 \cos k\theta \end{Bmatrix},$$
$$\hat{\mathbf{T}}^*_{3k} = \begin{Bmatrix} \tilde{F}_3 \cos k\theta \\ \tilde{G}_3 \cos k\theta \\ \tilde{H}_3 \sin k\theta \end{Bmatrix}, \quad \hat{\mathbf{T}}^*_{4k} = \begin{Bmatrix} \tilde{F}_3 \sin k\theta \\ \tilde{G}_3 \sin k\theta \\ -\tilde{H}_3 \cos k\theta \end{Bmatrix},$$

for $k \geq 2$, where

$$\begin{aligned}
\tilde{F}_1 &= D_1 r^{-3} + D_2 r \\
\tilde{G}_1 &= -D_1 r^{-3} + D_3 r \\
\tilde{H}_1 &= D_1 r^{-3} + D_4 r \\
\tilde{F}_2 &= J_1 r^{-k} + J_2 r^{k-2} + J_3 r^k \\
\tilde{G}_2 &= J_4 r^{-k} - J_2 r^{k-2} + J_5 r^k \\
\tilde{H}_2 &= J_6 r^{-k} - J_2 r^{k-2} + J_7 r^k \\
\tilde{F}_3 &= J_8 r^{-k} + J_9 r^{k-2} + J_{10} r^{-2-k} \\
\tilde{G}_3 &= J_{11} r^{-k} - J_9 r^{k-2} - J_{10} r^{-2-k} \\
\tilde{H}_3 &= J_{12} r^{-k} - J_9 r^{k-2} + J_{10} r^{-2-k}
\end{aligned}$$

with

$$\begin{aligned}
D_1 &= -\frac{2A_1 E}{1+\nu}, & D_2 &= \frac{E(2+\nu+\nu A_2)}{1-\nu^2}, \\
D_3 &= \frac{E(1+2\nu+A_2)}{1-\nu^2}, & D_4 &= \frac{E(-1+A_2)}{2(1+\nu)} \\
J_1 &= \frac{E[(1+\nu-k)B_1+\nu k B_5]}{1-\nu^2}, & J_2 &= \frac{E(k-1)B_2}{1+\nu}, \\
J_3 &= \frac{E(1+\nu+k+\nu k B_6)}{1-\nu^2}, & J_4 &= \frac{E[(1+\nu-\nu k)B_1+k B_5]}{1-\nu^2}, \\
J_5 &= \frac{E(1+\nu+\nu k+k B_6)}{1-\nu^2}, & J_6 &= -\frac{Ek(B_1+B_5)}{2(1+\nu)}, \\
J_7 &= \frac{Ek(B_6-1)}{2(1+\nu)}, & J_8 &= \frac{E[(1+\nu-k)B_3+\nu k B_7]}{1-\nu^2} \\
J_9 &= \frac{E(k-1)B_4}{1+\nu}, & J_{10} &= -\frac{E(1+k)}{1+\nu}, \\
J_{11} &= \frac{E[(1+\nu-\nu k)B_3+k B_7]}{1-\nu^2}, & J_{12} &= -\frac{Ek(B_3+B_7)}{2(1+\nu)}
\end{aligned}$$

The above displacement and stress expressions are presented in terms of polar coordinates. To obtain the corresponding expressions in Cartesian coordinates, the coordinate transformations of these expressions are necessary. According to the transformation rules for displacement and stress components given in Ref. [7], we have

$$\mathbf{N}^*_{jk} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \hat{\mathbf{N}}^*_{jk} \tag{8.35}$$

$$\mathbf{T}^*_{jk} = \begin{bmatrix} \cos^2\theta & \sin^2\theta & -\sin 2\theta \\ \sin^2\theta & \cos^2\theta & \sin 2\theta \\ \frac{1}{2}\sin 2\theta & -\frac{1}{2}\sin 2\theta & \cos 2\theta \end{bmatrix} \hat{\mathbf{T}}^*_{jk} \tag{8.36}$$

which is also different from the corresponding expression given on p. 1470 of Ref. [2]. Generally, properly truncated T-complete solutions can be used in the assumed intra-element field. It is worth pointing out that the homogeneous displacement solutions $\hat{\mathbf{N}}^*_{20}$ for $k=0$ and $\hat{\mathbf{N}}^*_{31}$, $\hat{\mathbf{N}}^*_{41}$ for $k=1$ represent rigid-body motions and thus cause zero stress solutions $\hat{\mathbf{T}}^*_{20}$, $\hat{\mathbf{T}}^*_{31}$ and $\hat{\mathbf{T}}^*_{41}$. They should be discarded in selecting the T-complete displacement vector $\mathbf{N}_e = \{\mathbf{N}_{e1} \quad \mathbf{N}_{e2} \quad \cdots \quad \mathbf{N}_{em}\}$ as a set of linearly independent functions $\mathbf{N}_{ej}$ associated with non-vanishing strains and stresses. In addition to the terms $\hat{\mathbf{N}}^*_{10}$ for $k=0$ and $\hat{\mathbf{N}}^*_{11}$, $\hat{\mathbf{N}}^*_{21}$ for $k=1$, the complete system of Trefftz functions should be truncated after either $\hat{\mathbf{N}}^*_{2k}$ and $\hat{\mathbf{T}}^*_{2k}$ or $\hat{\mathbf{N}}^*_{4k}$ and $\hat{\mathbf{T}}^*_{4k}$ for $k \geq 2$ in order to preserve the desirable geometrical invariance under rotation of the coordinate axes. This choice leads to the total number m of internal functions being odd. For illustration, we take $m=7$ as an example and present the proper choice of Trefftz functions by considering the rule of geometrical invariance above:

$$\begin{aligned}
\mathbf{N}_1 &= \mathbf{N}^*_{10}, \quad & \mathbf{T}_1 &= \mathbf{T}^*_{10} \\
\mathbf{N}_2 &= \mathbf{N}^*_{11}, \quad & \mathbf{T}_2 &= \mathbf{T}^*_{11} \\
\mathbf{N}_3 &= \mathbf{N}^*_{21}, \quad & \mathbf{T}_3 &= \mathbf{T}^*_{21} \\
\mathbf{N}_4 &= \mathbf{N}^*_{12}, \quad & \mathbf{T}_4 &= \mathbf{T}^*_{12} \\
\mathbf{N}_5 &= \mathbf{N}^*_{22}, \quad & \mathbf{T}_5 &= \mathbf{T}^*_{22} \\
\mathbf{N}_6 &= \mathbf{N}^*_{32}, \quad & \mathbf{T}_6 &= \mathbf{T}^*_{32} \\
\mathbf{N}_7 &= \mathbf{N}^*_{42}, \quad & \mathbf{T}_7 &= \mathbf{T}^*_{42}
\end{aligned} \tag{8.37}$$

8.5 Programming implementation

The programming procedure for the special HT circular hole element is the same as those presented in Chapters 2, 5 and 6 except for a few minor modifications on the regular HT element. Details of these minor modifications are presented next.

8.5.1 Data preparation

In addition to the material statement and element connectivity definition used in the regular element, two new arrays of data should be introduced in the circular hole element in order to completely define it. One array is used to store information about the element type, called `TSELE`, which is used to determine whether the element is a regular element (with value equal to 0) or a circular hole element (with value equal to 1). Another array, known as `CHELE`, stores the radius of the hole for a circular hole element and is set at zero for a regular element:

`TSELE(NELEM,1)`	Array storing types of element = 0: regular element = 1: circular hole element

`CHELE(NELEM,1)`	Array storing the radius of the hole. Value is 0 for a regular element and is the radius of the hole for a circular hole element

The corresponding section of the input data file shown in Appendix A should also be modified accordingly, for example,

```
-- Read element connect,type,radius and material number
Elem# Mat# TEle# CEle# Node#1--->#NNODE
1     1    0     0.0    1  75  17  76  19  77   4  78
2     1    1     2.0   17  79   2  18   3  80  19  76
......
```

8.5.2 Special Trefftz functions

Similar to the procedures in Chapters 5 and 6, the special Trefftz functions can be chosen from the series presented in Sections 8.3 and 8.4 and according to the specified number m of special Trefftz functions. The corresponding special subroutine is designed based on this understanding and is presented in Sections 8.6 and 8.7.

8.5.3 Output quantities

In regular T-elements, in general, the generalised nodal displacement field is computed and the generalised displacement and stress fields at the centroid of the element are then evaluated using the related formulations given in Chapters 5 and 6. However, in the special HT circular hole element, the related quantities at the element centroid have no physical meaning. In this case, only the fields at points on the circular hole boundary are considered (see Figure 8.3).

8.6 MATLAB functions for special T-elements

From the above procedure we can see that for each element we must justify whether it is a special element, as a result, all routines related to the usage of Trefftz functions must be modified to exhibit this choice. Besides the main function and data input module, four routines including `HMATRIX`, `GMATRIX`, `FIEDCEN`, `RIGIDRV` are related to Trefftz functions, so we will put our emphasis on these and ignore the unchanged ones for the sake of simplicity.

8.6.1 Potential problems

```
function MAINFUN
% Main program using HTFEM for 2D Laplace problems
```

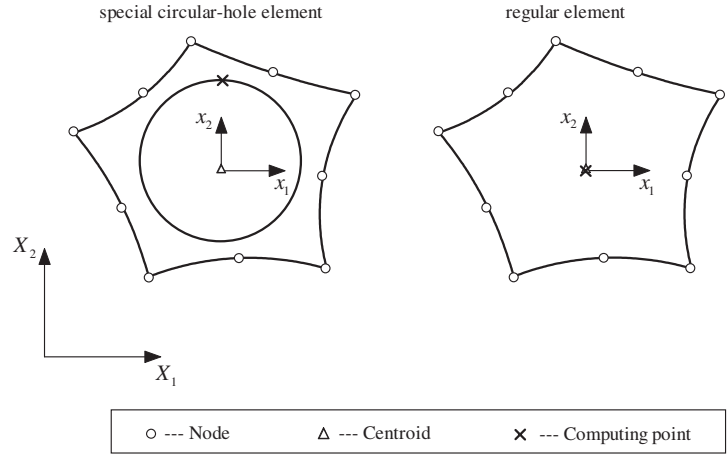


FIGURE 8.3
Selection of computing points within HT element

```
% with special element
% ******************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;

disp('*********************************************');
disp('              Hybrid Trefftz FEM');
disp('            for 2D Laplace problems');
disp('       with general and special elements');
disp('*********************************************');

% Input data from file
[NPOIN,NELEM,COORD,MATNO,TSELE,CHELE,LNODS,NVFIX,...
    NOFIX,IFPRE,PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,...
    NOPRS,PRESS,PROPS]=INPUTDT;
% Generate local relations of nodes and edges
[ELNOD]=TYPELEM;
% Element loop for stiffness matrix
NEQNS=NPOIN*NDOFN;
GSTIF=zeros(NEQNS,NEQNS);
GLOAD=zeros(NEQNS,1);
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    kTSEL=TSELE(iELEM);
    CLENG=CHELE(iELEM);
    % Compute some quantities related to each element
```

```
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS);
    %EHMTX=1/2*(EHMTX+EHMTX');
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS);
    % Compute element stiffnesss matrix
    [ESTIF]=KMATRIX(EHMTX,EGMTX);
    % Assemble stiffness matrix
    [GSTIF]=ASMSTIF(iELEM,LNODS,ESTIF,GSTIF);
end
% Compute equivalent loads
[GLOAD]=PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,...
    NEASS,NOPRS,PRESS,NPLOD,LODPT,POINT,GLOAD);
% Introduce constrained displacements and point loads
[GSTIF,GLOAD]=INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,...
    GSTIF,GLOAD);
% Solve linear system of equations and store
% displacements of each node in the array ASDIS
[ASDIS]=LSSOLVR(GSTIF,GLOAD,NEQNS);
% Output nodal displacements
[UPOIN]=FIEDNOD(NPOIN,ASDIS);
% Compute displacement and stress components at central
% point of element
[CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    TSELE,CHELE,LNODS,COORD,PROPS,ELNOD,ASDIS);
% Output results
OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,...
    NVFIX,NPLOD,NDLEG);
disp('--- All procedures are finished ---');


-------------------------------------------------------
function [NPOIN,NELEM,COORD,MATNO,TSELE,CHELE,LNODS,...
    NVFIX,NOFIX,IFPRE,PRESC,NPLOD,NDLEG,LODPT,POINT,...
    NEASS,NOPRS,PRESS,PROPS]=INPUTDT
% Input data from a file
% Input parameters: No
% Output parameters:
%   NPOIN: Number of nodes in domain
%   NELEM: Number of elements in domain
%   COORD: Coordinates of nodes
%   MATNO: Material index of each element
%   TSELE: Element types
%              =0, general element
```

```
%              =1, special circular hole element
%   LNODS: Element connectivity
%   NVFIX: Number of boundary nodes at which specified
%          DOF is restricted
%   NOFIX: Global index of nodes at which specified DOF
%          is restricted
%   IFPRE: Types of constraints of each DOF
%   PRESC: Specified values
%   NPLOD: Number of concentrated loads
%   NDLEG: Number of loaded edges
%   LODPT: Global index of nodes at which concentrated
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   PROPS: Properties of materials
% *********************************
global NTREF NTYPE NDIME NDOFN NSTRE NNODE NODEG NEDGE;
global NMATS NPROP NGAUS;

% Open the input file
FILE1=input('Input data file name: ','s');
fp=fopen(FILE1,'r');
if fp<0
    error('-- Error: Can't open data file!');
end
%fp=fopen('input.txt','r');
dummy=char(zeros(1,100));  % fill with ASCII zeros
TITLE=char(zeros(1,200));  % fill with ASCII zeros
% Description of problem
dummy=fgets(fp);
TITLE=fgets(fp);
dummy=fgets(fp);
% Number of Trefftz functions and type of problems
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NTREF,NTYPE]=deal(TMP(1),TMP(2));
% Element properties
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NNODE,NEDGE,NODEG]=deal(TMP(1),TMP(2),TMP(3));
% Number of Dimensions, DOF and stress components
dummy=fgets(fp);
```

```
TMP=str2num(fgets(fp));
[NDIME,NDOFN,NSTRE]=deal(TMP(1),TMP(2),TMP(3));
% Number of materials, material properties and Gaussian
% points
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NMATS,NPROP,NGAUS]=deal(TMP(1),TMP(2),TMP(3));
% Number of nodes, elements and boundary conditions
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NPOIN,NELEM,NVFIX,NPLOD,NDLEG]=...
    deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5));
% Read element connectivity and material numbers
MATNO=zeros(NELEM,1);
TSELE=zeros(NELEM,1);
LNODS=zeros(NELEM,NNODE);
dummy=fgets(fp);
dummy=fgets(fp);
for iELEM=1:NELEM
    TMP=str2num(fgets(fp));
    [N,MATNO(iELEM),TSELE(iELEM),CHELE(iELEM),...
        LNODS(iELEM,:)]=...
       deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5:4+NNODE));
end
% Read nodal coordinates
COORD=zeros(NPOIN,NDIME);
dummy=fgets(fp);
dummy=fgets(fp);
for iPOIN=1:NPOIN
    TMP=str2num(fgets(fp));
    [N,COORD(iPOIN,:)]=deal(TMP(1),TMP(2:1+NDIME));
end
% Read essential boundary conditions
NOFIX=zeros(NVFIX,1);
IFPRE=zeros(NVFIX,NDOFN);
PRESC=zeros(NVFIX,NDOFN);
N1=2+NDOFN;
N2=3+NDOFN;
N3=2+2*NDOFN;
dummy=fgets(fp);
dummy=fgets(fp);
for iVFIX=1:NVFIX
    TMP=str2num(fgets(fp));
    [N,NOFIX(iVFIX),IFPRE(iVFIX,:),PRESC(iVFIX,:)]=...
        deal(TMP(1),TMP(2),TMP(3:N1),TMP(N2:N3));
```

```
end
% Read natural boundary conditions
LODPT=zeros(NPLOD,1);
POINT=zeros(NPLOD,NDIME);
NEASS=zeros(NDLEG,1);
NOPRS=zeros(NDLEG,NODEG);
PRESS=zeros(NDLEG,NODEG*NDOFN);
% Read concentrated loads
dummy=fgets(fp);
if NPLOD>0
    dummy=fgets(fp);
    for iPLOD=1:NPLOD
        TMP=str2num(fgets(fp));
        [N,LODPT(iPLOD),POINT(iPLOD,:)]=...
            deal(TMP(1),TMP(2),TMP(3:2+NDIME));
    end
end
% Read distributed edge loads
dummy=fgets(fp);
if NDLEG>0
    N1=2+NODEG;
    N2=3+NODEG;
    N3=2+NODEG+NODEG*NDOFN;
    dummy=fgets(fp);
    for iDLEG=1:NDLEG
        TMP=str2num(fgets(fp));
        [N,NEASS(iDLEG),NOPRS(iDLEG,:),...
            PRESS(iDLEG,:)]=...
            deal(TMP(1),TMP(2),TMP(3:N1),TMP(N2:N3));
    end
end
% Read material properties
PROPS=zeros(NMATS,NPROP);
dummy=fgets(fp);
dummy=fgets(fp);
for iMATS=1:NMATS
    TMP=str2num(fgets(fp));
    [N,PROPS(iMATS,:)]=deal(TMP(1),TMP(2:1+NPROP));
end
fclose(fp);


------------------------------------------------------
function [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,...
    CLENG,PROPS)
```

```
% Compute element matrix H
%       H = integral( QT * N) on element boundary
% Input parameters:
%   ECOOD: Coordinates of nodes of specified element
%   ELNOD: Local relation of edge and nodes
%   kMATS: Material number of specified element
%   kTSEL: Element type
%   CLENG: Element characteristic length
%   PROPS: Properties of materials
% Output parameters:
%   EHMTX: Element H matrix
% *********************************
global NTREF NDIME NDOFN NNODE NEDGE NGAUS;

% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);
% Material properties
THICK=PROPS(kMATS,3);
% Compute H matrix of every edge of element
EHMTX=zeros(NTREF,NTREF);
for iEDGE=1:NEDGE
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
            EXISP,ECOOD,ELNOD);
        DVOLU=DVOLU*WEIGP(iGAUS);
        if THICK+1~=1
            DVOLU=DVOLU*THICK;
        end
        xp=CORGS(1);
        yp=CORGS(2);
        if kTSEL==0 % general element
            [N_SET,T_SET]=TREFFTZ(xp,yp);
        elseif kTSEL==1 % special circular hole element
            [N_SET,T_SET]=SCHTREF(xp,yp,CLENG);
        end
        Q_SET=AMTRX*T_SET;
        QTN=Q_SET'*N_SET; % m by m
        for im=1:NTREF
            for jm=1:NTREF
                EHMTX(im,jm)=EHMTX(im,jm)+...
                    QTN(im,jm)*DVOLU;
            end
        end
    end
```

```
end
clear Q_SET QTN N_SET T_SET POSGP WEIGP SHMTX;


-----------------------------------------------------
function [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,...
    CLENG,PROPS)
% Compute element matrix G
%   G = integral( QT* ShapEge) along element boundary
% Input parameters:
%   ECOOD: Coordinates of nodes of specified element
%   ELNOD: Local relation of edge and nodes
%   kMATS: Material number of specified element
%   kTSEL: Element type
%   CLENG: Element characteristic length
%   PROPS: Properties of materials
% Output parameters:
%   EGMTX: Element G matrix
% ***********************************************
global NTREF NDIME NDOFN NEDGE NNODE NGAUS;

% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);
% Material properties
THICK=PROPS(kMATS,3);

% Compute G matrix of every element
EGMTX=zeros(NTREF,NNODE);
for iEDGE=1:NEDGE
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
            EXISP,ECOOD,ELNOD);
        DVOLU=DVOLU*WEIGP(iGAUS);
        if THICK+1~=1
            DVOLU=DVOLU*THICK;
        end
        xp=CORGS(1);
        yp=CORGS(2);
        if kTSEL==0 % general element
            [N_SET,T_SET]=TREFFTZ(xp,yp);
        elseif kTSEL==1 % special circular hole element
            [N_SET,T_SET]=SCHTREF(xp,yp,CLENG);
        end
        Q_SET=AMTRX*T_SET;
```

```
        QTShapEge=Q_SET'*SHMTX;
        for im=1:NTREF
            for jn=1:NNODE
                EGMTX(im,jn)=EGMTX(im,jn)+...
                    QTShapEge(im,jn)*DVOLU;
            end
        end
    end
end
clear Q_SET QTShapEge N_SET T_SET POSGP WEIGP SHMTX;


------------------------------------------------------
function [CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    TSELE,CHELE,LNODS,COORD,PROPS,ELNOD,ASDIS)
% Compute internal fields at central point of each
% element
% Input parameters:
%   NELEM: Number of elements in domain
%   MATNO: Material index of each element
%   TSELE: Element type
%            = 0: general element
%            = 1: circular hole element
%   CHELE: Element characteristic length
%            = 0 for general element
%            = b for special element
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
%   PROPS: Properties of materials
%   ELNOD: Local relation of edge and nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
% Output parameters:
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
% ************************************************
global NDIME NDOFN NNODE NEDGE NODEG NSTRE NMATS NGAUS;

CECOD=zeros(NELEM,NDIME);
UCENP=zeros(NELEM,NDOFN);
SCENP=zeros(NELEM,NSTRE);
% Loop for all nodes
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
```

```
    kTSEL=TSELE(iELEM);
    CLENG=CHELE(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Identify nodal field of the specified element
    [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS);
    % Calculate the ce coefficients: m by 1
    [c_Ele]=CMATRIX(EHMTX,EGMTX,d_Ele);
    % Recover rigid displacement
    [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele,kTSEL,CLENG);
    % Compute internal fields at element central point
    % Note: coordinates of centroid is (0,0) in local
    % element coordinates system
    if kTSEL==0 % conventional Trefftz element
        xp=0;
        yp=0;
        [N_SET,T_SET]=TREFFTZ(xp,yp);
    elseif kTSEL==1 % special circular hole element
        xp=0;
        yp=CLENG;
        [N_SET,T_SET]=SCHTREF(xp,yp,CLENG);
    end
    GDISP=N_SET*c_Ele;
    GSTRE=T_SET*c_Ele;
    UCENP(iELEM,1)=GDISP(1)+c0;
    SCENP(iELEM,1)=GSTRE(1);
    SCENP(iELEM,2)=GSTRE(2);
    % Coordinates of commputing point
    CECOD(iELEM,:)=[CenCoord(1)+xp,CenCoord(2)+yp];
end
clear EHMTX EGMTX c_Ele d_Ele GDISP GSTRE;


-------------------------------------------------------
function [c0]=RigidRecover(ECOOD,c_Ele,d_Ele,...
    kTSEL,CLENG)
% Recovery of rigid body motion
% Input parameters:
%   ECOOD: Coordinates of element nodes
%   c_Ele: Coefficients of Trefftz interpolation
%   d_Ele: Displacement field at element nodes
```

```
%   kTSEL: Element type
%   CLENG: Element characteristic length
% Output parameters:
%   c0: Rigid body motion term
% ********************************

global NNODE;
sum=0;
for iNODE=1:NNODE
    xp=ECOOD(iNODE,1);
    yp=ECOOD(iNODE,2);
    if kTSEL==0 % general element
        [N_SET,T_SET]=TREFFTZ(xp,yp);
    elseif kTSEL==1 % circular hole element
        [N_SET,T_SET]=SCHTREF(xp,yp,CLENG);
    end
    sum=sum+(d_Ele(iNODE)-N_SET*c_Ele);
end
c0=sum/NNODE;


------------------------------------------------------
function [N_SET,T_SET]=SCHTREF(sp,tp,b)
% Compute the special circular hole Trefftz functions
% with specified terms
% Input parameters:
%   sp, tp: coordinates
%   b     : radius of circular hole
% Output parameters:
%   N_SET: Trefftz functions
%   T_SET: Derivatives of Trefftz functions
%              dN/ds and dN/dt
% ***********************************************
global NTREF NNODE NDOFN;

% Check the number of terms of Trefftz functions
temp=NNODE*NDOFN-1;
if NTREF<temp
    error('Too small terms of Trefftz functions!');
end
% Compute N_SET and T_SET
N_SET=zeros(1,NTREF);
T_SET=zeros(2,NTREF);

r=sqrt(sp^2+tp^2);
```

```
s=atan2(tp,sp);

for im=1:NTREF
    % Determine the order of Ni
    n=ceil(im/2);
    % Justify im is odd or even number
    remaind=rem(im,2);
    if 1+remaind==1 % im is even
        N_SET(1,im)=(r^n+b^(2*n)*r^(-n))*sin(n*s);
        T_SET(1,im)=r^(n-1)*n*sin(n*s-s)-...
            b^(2*n)*r^(-n-1)*n*sin(n*s+s);
        T_SET(2,im)=r^(n-1)*n*cos(n*s-s)+...
            b^(2*n)*r^(-n-1)*n*cos(n*s+s);
    else % im is odd
        N_SET(1,im)=(r^n+b^(2*n)*r^(-n))*cos(n*s);
        T_SET(1,im)= r^(n-1)*n*cos(n*s-s)-...
            b^(2*n)*r^(-n-1)*n*cos(n*s+s);
        T_SET(2,im)=-r^(n-1)*n*sin(n*s-s)-...
            b^(2*n)*r^(-n-1)*n*sin(n*s+s);
    end
end
```

8.6.2 Elastic problems

```
function MAINFUN
%** Main function calling other subroutines for solving
%   plane elasticity with general and special elements
%*****************************************
global NTREF NTYPE NDIME NDOFN NNODE NEDGE NODEG NSTRE;
global NMATS NPROP NGAUS;

disp('*********************************************');
disp('            Hybrid Trefftz FEM');
disp('        for plane elastic problems');
disp('     with general and special elements');
disp('*********************************************');

% Input data from file
[NPOIN,NELEM,COORD,MATNO,TSELE,CHELE,LNODS,NVFIX,...
    NOFIX,IFPRE,PRESC,NPLOD,NDLEG,LODPT,POINT,NEASS,...
    NOPRS,PRESS,PROPS]=INPUTDT;
% Generate local relations of nodes and edges
[ELNOD]=TYPELEM;
% Element loop for stiffness matrix
```

```
NEQNS=NPOIN*NDOFN;
GSTIF=zeros(NEQNS,NEQNS);
GLOAD=zeros(NEQNS,1);
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    kTSEL=TSELE(iELEM);
    CLENG=CHELE(iELEM);
    % Compute some quantities of each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,...
        CLENG,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,...
        CLENG,PROPS);
    % Compute element stiffnesss matrix
    [ESTIF]=KMATRIX(EHMTX,EGMTX);
    % Assemble stiffness matrix
    [GSTIF]=ASMSTIF(iELEM,NNODE,LNODS,ESTIF,GSTIF);
end
% Compute equivalent loads
[GLOAD]=PVECTOR(LNODS,COORD,NDLEG,NEASS,NOPRS,...
    PRESS,NPLOD,LODPT,POINT,GLOAD);
% Introduce constrained displacements and point loads
[GSTIF,GLOAD]=INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,...
    GSTIF,GLOAD);
% Solve linear system of equations and store
% displacements of each node in the array ASDIS
[ASDIS]=LSSOLVR(GSTIF,GLOAD,NEQNS);
% Output nodal potential
[UPOIN]=FIEDNOD(NPOIN,ASDIS);
% Compute potential and flux components at central
% point of element
[CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,TSELE,CHELE,...
    LNODS,COORD,PROPS,ELNOD,ASDIS);
% Output results
OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,...
    NVFIX,NPLOD,NDLEG);
disp('--- All procedures are finished ---');


------------------------------------------------------
function [NPOIN,NELEM,COORD,MATNO,TSELE,CHELE,LNODS,...
    NVFIX,NOFIX,IFPRE,PRESC,NPLOD,NDLEG,LODPT,POINT,...
    NEASS,NOPRS,PRESS,PROPS]=INPUTDT
```

```
% ** Input data from a file
% Input parameters: No
% Output parameters:
%   NPOIN: Number of nodes in domain
%   NELEM: Number of elements in domain
%   COORD: Coordinates of nodes
%   MATNO: Material index of each element
%   TSELE: Element types
%              =0, general element
%              =1, special circular hole element
%   LNODS: Element connectivity
%   NVFIX: Number of boundary nodes at which specified
%          DOF is restricted
%   NOFIX: Global index of nodes at which specified DOF
%          is restricted
%   IFPRE: Types of constraints of each DOF
%   PRESC: Specified values
%   NPLOD: Number of concentrated loads
%   NDLEG: Number of loaded edges
%   LODPT: Global index of nodes at which concentrated
%          loads are applied
%   POINT: Specified values of concentrated loads
%   NEASS: Element index with loaded edge
%   NOPRS: Global node index along loaded edge
%   PRESS: Specified values of distributed loads at
%          nodes
%   PROPS: Properties of materials
% *********************************
global NTREF NTYPE NDIME NDOFN NSTRE NNODE NODEG NEDGE;
global NMATS NPROP NGAUS;

% Open the input file
FILE1=input('Input data file name: ','s');
fp=fopen(FILE1,'r');
if fp<0
    error('-- Error: Can't open data file!');
end
%fp=fopen('input.txt','r');
dummy=char(zeros(1,100));  % fill with ASCII zeros
TITLE=char(zeros(1,200));  % fill with ASCII zeros
% Description of problem
dummy=fgets(fp);
TITLE=fgets(fp);
dummy=fgets(fp);
% Number of Trefftz functions and type of problems
```

```
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NTREF,NTYPE]=deal(TMP(1),TMP(2));
% Element properties
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NNODE,NEDGE,NODEG]=deal(TMP(1),TMP(2),TMP(3));
% Number of Dimensions, DOF and stress components
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NDIME,NDOFN,NSTRE]=deal(TMP(1),TMP(2),TMP(3));
% Number of materials, material properties and Gaussian
% points
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NMATS,NPROP,NGAUS]=deal(TMP(1),TMP(2),TMP(3));
% Number of nodes, elements and boundary conditions
dummy=fgets(fp);
TMP=str2num(fgets(fp));
[NPOIN,NELEM,NVFIX,NPLOD,NDLEG]=...
    deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5));
% Read element connectivity and material numbers
MATNO=zeros(NELEM,1);
TSELE=zeros(NELEM,1);
LNODS=zeros(NELEM,NNODE);
dummy=fgets(fp);
dummy=fgets(fp);
for iELEM=1:NELEM
    TMP=str2num(fgets(fp));
    [N,MATNO(iELEM),TSELE(iELEM),CHELE(iELEM),...
        LNODS(iELEM,:)]=...
       deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5:4+NNODE));
end
% Read nodal coordinates
COORD=zeros(NPOIN,NDIME);
dummy=fgets(fp);
dummy=fgets(fp);
for iPOIN=1:NPOIN
    TMP=str2num(fgets(fp));
    [N,COORD(iPOIN,:)]=deal(TMP(1),TMP(2:1+NDIME));
end
% Read essential boundary conditions
NOFIX=zeros(NVFIX,1);
IFPRE=zeros(NVFIX,NDOFN);
PRESC=zeros(NVFIX,NDOFN);
```

```
N1=2+NDOFN;
N2=3+NDOFN;
N3=2+2*NDOFN;
dummy=fgets(fp);
dummy=fgets(fp);
for iVFIX=1:NVFIX
    TMP=str2num(fgets(fp));
    [N,NOFIX(iVFIX),IFPRE(iVFIX,:),PRESC(iVFIX,:)]=...
        deal(TMP(1),TMP(2),TMP(3:N1),TMP(N2:N3));
end
% Read natural boundary conditions
LODPT=zeros(NPLOD,1);
POINT=zeros(NPLOD,NDIME);
NEASS=zeros(NDLEG,1);
NOPRS=zeros(NDLEG,NODEG);
PRESS=zeros(NDLEG,NODEG*NDOFN);
% Read concentrated loads
dummy=fgets(fp);
if NPLOD>0
    dummy=fgets(fp);
    for iPLOD=1:NPLOD
        TMP=str2num(fgets(fp));
        [N,LODPT(iPLOD),POINT(iPLOD,:)]=...
            deal(TMP(1),TMP(2),TMP(3:2+NDIME));
    end
end
% Read distributed edge loads
dummy=fgets(fp);
if NDLEG>0
    N1=2+NODEG;
    N2=3+NODEG;
    N3=2+NODEG+NODEG*NDOFN;
    dummy=fgets(fp);
    for iDLEG=1:NDLEG
        TMP=str2num(fgets(fp));
        [N,NEASS(iDLEG),NOPRS(iDLEG,:),...
            PRESS(iDLEG,:)]=...
            deal(TMP(1),TMP(2),TMP(3:N1),TMP(N2:N3));
    end
end
% Read material properties
PROPS=zeros(NMATS,NPROP);
dummy=fgets(fp);
dummy=fgets(fp);
for iMATS=1:NMATS
```

```
    TMP=str2num(fgets(fp));
    [N,PROPS(iMATS,:)]=deal(TMP(1),TMP(2:1+NPROP));
end
fclose(fp);


------------------------------------------------------
function [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,...
    kTSEL,CLENG,PROPS)
% Compute element matrix H
%   H = integral( QT * N) on element boundary
% Input parameters:
%   ECOOD: Coordinates of nodes of specified element
%   ELNOD: Local relation of edge and nodes
%   kMATS: Material number of specified element
%   kTSEL: Type of element
%   CLENG: Characteristic length of special element
%   PROPS: Properties of materials
% Output parameters:
%   EHMTX: Element H matrix
% ************************************************
global NTREF NDIME NDOFN NNODE NEDGE NGAUS;

% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);
% Material properties
THICK=PROPS(kMATS,3);
% Compute H matrix of every edge of element
EHMTX=zeros(NTREF,NTREF);
for iEDGE=1:NEDGE
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
            EXISP,ECOOD,ELNOD);
        DVOLU=DVOLU*WEIGP(iGAUS);
        if THICK+1~=1
            DVOLU=DVOLU*THICK;
        end
        xp=CORGS(1);
        yp=CORGS(2);
        if kTSEL==0
            [N_SET,T_SET]=TREFFTZ(xp,yp,kMATS,PROPS);
        elseif kTSEL==1
            [N_SET,T_SET]=SCHTREF(xp,yp,...
                CLENG,kMATS,PROPS);
```

```
        end
        Q_SET=AMTRX*T_SET;
        QTN=Q_SET'*N_SET;
        for im=1:NTREF
            for jm=1:NTREF
                EHMTX(im,jm)=EHMTX(im,jm)+...
                    QTN(im,jm)*DVOLU;
            end
        end
    end
end
clear Q_SET QTN N_SET T_SET POSGP WEIGP SHMTX;


------------------------------------------------------
function [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,...
    kTSEL,CLENG,PROPS)
% Compute element matrix G
%   G = integral( QT* Shape) along element boundary
% Input parameters:
%   ECOOD: Coordinates of nodes of specified element
%   ELNOD: Local relation of edge and nodes
%   kMATS: Material number of specified element
%   kTSEL: Type of element
%   CLENG: Characteristic length of special element
%   PROPS: Properties of materials
% Output parameters:
%   EGMTX: Element G matrix
% ************************************************
global NTREF NDIME NDOFN NEDGE NNODE NGAUS;

% Gaussian point and weight coefficients
[POSGP,WEIGP]=GAUSSQU(NGAUS);
% Material properties
THICK=PROPS(kMATS,3);
% Compute G matrix of every element
NEVAB=NNODE*NDOFN;
EGMTX=zeros(NTREF,NEVAB);
for iEDGE=1:NEDGE
    for iGAUS=1:NGAUS
        EXISP=POSGP(iGAUS);
        [CORGS,DVOLU,AMTRX,SHMTX]=QUANGAS(iEDGE,...
            EXISP,ECOOD,ELNOD);
        DVOLU=DVOLU*WEIGP(iGAUS);
        if THICK+1~=1
```

```
            DVOLU=DVOLU*THICK;
        end
        xp=CORGS(1);
        yp=CORGS(2);
        if kTSEL==0
            [N_SET,T_SET]=TREFFTZ(xp,yp,kMATS,PROPS);
        elseif kTSEL==1
            [N_SET,T_SET]=SCHTREF(xp,yp,CLENG,...
                kMATS,PROPS);
        end
        Q_SET=AMTRX*T_SET;
        QTS=Q_SET'*SHMTX;
        for im=1:NTREF
            for jn=1:NEVAB
                EGMTX(im,jn)=EGMTX(im,jn)+...
                    QTS(im,jn)*DVOLU;
            end
        end
    end
end
clear Q_SET QTS N_SET T_SET POSGP WEIGP SHMTX;


------------------------------------------------------
function [CECOD,UCENP,SCENP]=FIEDCEN(NELEM,MATNO,...
    TSELE,CHELE,LNODS,COORD,PROPS,ELNOD,ASDIS)
% Compute internal fields at central point of each
% element
% Input parameters:
%   NELEM: Number of elements in domain
%   MATNO: Material index of each element
%   LNODS: Element connectivity
%   COORD: Coordinates of nodes
%   PROPS: Properties of materials
%   ELNOD: Local relation of edge and nodes
%   ASDIS: Nodal generalised displacement field in DOF
%          order
% Output parameters:
%   CECOD: Coordinates of centroid of each element
%   UCENP: Displacement fields at centroid
%   SCENP: Stress fields at centroid
% ************************************************
global NDIME NDOFN NNODE NEDGE NODEG NSTRE NMATS NGAUS;

CECOD=zeros(NELEM,NDIME);
```

```
UCENP=zeros(NELEM,NDOFN);
SCENP=zeros(NELEM,NSTRE);
% Element loop for internal fields at central point
for iELEM=1:NELEM
    kMATS=MATNO(iELEM);
    kTSEL=TSELE(iELEM);
    CLENG=CHELE(iELEM);
    % Compute some quantities related to each element
    [ECOOD,CenCoord]=ELEPARS(iELEM,LNODS,COORD);
    % Identify nodal field of the specified element
    [d_Ele]=EDISNOD(iELEM,LNODS,ASDIS);
    % Compute H matrix
    [EHMTX]=HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,...
        CLENG,PROPS);
    % Compute G matrix
    [EGMTX]=GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,...
        CLENG,PROPS);
    % Calculate the ce coefficients: m by 1
    [c_Ele]=CMATRIX(EHMTX,EGMTX,d_Ele);
    % Recover rigid-body motion vector: 3 by 1
    [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,kTSEL,...
        CLENG,PROPS);
    % Compute internal fields at element central point
    % Note: coordinates of centroid is (0,0) in local
    % element coordinates system
    if kTSEL==0
        xp=0;
        yp=0;
        [N_SET,T_SET]=TREFFTZ(xp,yp,kMATS,PROPS);
    elseif kTSEL==1
        xp=0;
        yp=CLENG;
        [N_SET,T_SET]=SCHTREF(xp,yp,CLENG,kMATS,PROPS);
    end
    GDISP=N_SET*c_Ele;
    GSTRE=T_SET*c_Ele;
    UCENP(iELEM,1)=GDISP(1)+c0(1)+yp*c0(3);
    UCENP(iELEM,2)=GDISP(2)+c0(2)-xp*c0(3);
    SCENP(iELEM,1)=GSTRE(1);
    SCENP(iELEM,2)=GSTRE(2);
    SCENP(iELEM,3)=GSTRE(3);
    % Coordinates of computing points
    CECOD(iELEM,:)=[CenCoord(1)+xp,CenCoord(2)+yp];
end
clear EHMTX EGMTX c_Ele d_Ele N_SET T_SET;
```

```
------------------------------------------------------
function [c0]=RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,...
    kTSEL,CLENG,PROPS)
% Rigid body motion recovery
global NNODE;

RMATX=zeros(3,3);
rvect=zeros(3,1);
for iNODE=1:NNODE
    x1=ECOOD(iNODE,1);
    x2=ECOOD(iNODE,2);
    if kTSEL==0 % general element
        [N_SET,T_SET]=TREFFTZ(x1,x2,kMATS,PROPS);
    elseif kTSEL==1 % special circular hole element
        [N_SET,T_SET]=SCHTREF(x1,x2,kMATS,CLENG,PROPS);
    end
    u=N_SET*c_Ele;

    du1=d_Ele(iNODE*2-1)-u(1);
    du2=d_Ele(iNODE*2  )-u(2);
    rvect(1)=rvect(1)+du1;
    rvect(2)=rvect(2)+du2;
    rvect(3)=rvect(3)+x2*du1-x1*du2;

    RMATX(1,3)=RMATX(1,3)+x2;
    RMATX(2,3)=RMATX(2,3)-x1;
    RMATX(3,3)=RMATX(3,3)+(x1^2+x2^2);
end
RMATX(3,1)=RMATX(1,3);
RMATX(3,2)=RMATX(2,3);
RMATX(1,1)=NNODE;
RMATX(2,2)=NNODE;

c0=RMATX\rvect;

clear N_SET T_SET RMATX rvect;


------------------------------------------------------
function [N_SET,T_SET]=SCHTREF(sp,tp,b,kMATS,PROPS)
% Compute Trefftz functions of special circular hole
% element
% Input parameters:
```

```
%   sp,tp: Coordinates
%   CLENG: Characteristic length
%   kMATS: Material number
%   PROPS: Material properties
% Output parameters:
%   N_SET: Displacements Trefftz functions
%   T_SET: Stresses Trefftz functions
% ************************************************
global NTREF NTYPE NNODE NDOFN;

% Check the number of terms of Trefftz functions
temp=NNODE*NDOFN-3;
if NTREF<temp
    error('Too small terms of Trefftz functions!');
end
% Initializing N and T
N_SET=zeros(2,NTREF);
T_SET=zeros(3,NTREF);
% Material properties
YOUNG=PROPS(kMATS,1);
POISS=PROPS(kMATS,2);
if NTYPE==1 % plane stress
    E=YOUNG;
    nu=POISS;
elseif NTYPE==2 % plane strain
    E=YOUNG/(1-POISS^2);
    nu=POISS/(1-POISS);
end
% Evaluate r ans s
r=sqrt(sp^2+tp^2);
s=atan2(tp,sp);
% k=0
k=0;
A0=(1+nu)/(1-nu)*b^2;

DF0=A0/r+r;
DG0=0;
SF0=E/(1-nu)-E/(1+nu)*A0/r^2;
SG0=E/(1-nu)+E/(1+nu)*A0/r^2;
SH0=0;

N_SET(1,1)=DF0;
N_SET(2,1)=DG0;
T_SET(1,1)=SF0;
T_SET(2,1)=SG0;
```

```
T_SET(3,1)=SH0;
% k=1
k=1;
A1=(1+nu)/(1-3*nu)*b^4;
A2=(5+nu)/(1-3*nu);
D1=-2*E*A1/(1+nu);
D2=E*(2+nu+nu*A2)/(1-nu^2);
D3=E*(1+2*nu+A2)/(1-nu^2);
D4=E*(-1+A2)/2/(1+nu);

DF1=A1/r^2+r^2;
DG1=A1/r^2+A2*r^2;
SF1= D1*r^(-3)+D2*r;
SG1=-D1*r^(-3)+D3*r;
SH1= D1*r^(-3)+D4*r;

N_SET(1,2)=DF1*cos(s);
N_SET(2,2)=DG1*sin(s);
T_SET(1,2)=SF1*cos(s);
T_SET(2,2)=SG1*cos(s);
T_SET(3,2)=SH1*sin(s);

N_SET(1,3)= DF1*sin(s);
N_SET(2,3)=-DG1*cos(s);
T_SET(1,3)= SF1*sin(s);
T_SET(2,3)= SG1*sin(s);
T_SET(3,3)=-SH1*cos(s);

NK(1)=0;
NK(2:3)=1;

nt=4;
% k>=2
while nt<NTREF
    k=k+1;

    B1=(2*(1-nu)+k*(1+nu))/((k-1)*(2*(1-nu)-k*(1+nu)))*b^(2*k);
    B2=k^2*(1+nu)/((k-1)*(2*(1-nu)-k*(1+nu)))*b^2;
    B3=(2*(1-nu)+k*(1+nu))/(1-k)/(1+nu)*b^(-2);
    B4=1/(1-k)*b^(-2*k);
    B5=(k*(1+nu)-4)/((k-1)*(2*(1-nu)-k*(1+nu)))*b^(2*k);
    B6=(k*(1+nu)+4)/(2*(1-nu)-k*(1+nu));
    B7=(k*(1+nu)-4)/(1-k)/(1+nu)*b^(-2);
    J1=E*((1+nu-k)*B1+nu*k*B5)/(1-nu^2);
    J2=E*(k-1)*B2/(1+nu);
```

```
J3=E*((1+nu+k)+nu*k*B6)/(1-nu^2);
J4=E*((1+nu-nu*k)*B1+k*B5)/(1-nu^2);
J5=E*((1+nu+nu*k)+k*B6)/(1-nu^2);
J6=-E*k*(B1+B5)/2/(1+nu);
J7=E*k*(B6-1)/2/(1+nu);
J8=E*((1+nu-k)*B3+nu*k*B7)/(1-nu^2);
J9=E*(k-1)*B4/(1+nu);
J10=-E*(1+k)/(1+nu);
J11=E*((1+nu-nu*k)*B3+k*B7)/(1-nu^2);
J12=-E*k*(B3+B7)/2/(1+nu);

DF2=B1*r^(1-k)+B2*r^(k-1)+r^(1+k);
DG2=B5*r^(1-k)-B2*r^(k-1)+B6*r^(1+k);
DF3=B3*r^(1-k)+B4*r^(k-1)+r^(-1-k);
DG3=B7*r^(1-k)-B4*r^(k-1)+r^(-1-k);
SF2=J1*r^(-k)+J2*r^(k-2)+J3*r^(k);
SG2=J4*r^(-k)-J2*r^(k-2)+J5*r^(k);
SH2=J6*r^(-k)-J2*r^(k-2)+J7*r^(k);
SF3=J8* r^(-k)+J9*r^(k-2)+J10*r^(-2-k);
SG3=J11*r^(-k)-J9*r^(k-2)-J10*r^(-2-k);
SH3=J12*r^(-k)-J9*r^(k-2)+J10*r^(-2-k);

N_SET(1,nt)=DF2*cos(k*s);
N_SET(2,nt)=DG2*sin(k*s);
T_SET(1,nt)=SF2*cos(k*s);
T_SET(2,nt)=SG2*cos(k*s);
T_SET(3,nt)=SH2*sin(k*s);
NK(nt)=k;

nt=nt+1;
N_SET(1,nt)= DF2*sin(k*s);
N_SET(2,nt)=-DG2*cos(k*s);
T_SET(1,nt)= SF2*sin(k*s);
T_SET(2,nt)= SG2*sin(k*s);
T_SET(3,nt)=-SH2*cos(k*s);
NK(nt)=k;

nt=nt+1;
if nt>=NTREF
    break;
end
N_SET(1,nt)=DF3*cos(k*s);
N_SET(2,nt)=DG3*sin(k*s);
T_SET(1,nt)=SF3*cos(k*s);
T_SET(2,nt)=SG3*cos(k*s);
```

```
    T_SET(3,nt)=SH3*sin(k*s);
    NK(nt)=k;

    nt=nt+1;
    N_SET(1,nt)= DF3*sin(k*s);
    N_SET(2,nt)=-DG3*cos(k*s);
    T_SET(1,nt)= SF3*sin(k*s);
    T_SET(2,nt)= SG3*sin(k*s);
    T_SET(3,nt)=-SH3*cos(k*s);
    NK(nt)=k;

    nt=nt+1;
end
% Transformation of coordinates
DM=[cos(s),-sin(s);sin(s),cos(s)];
SM=[(cos(s))^2,(sin(s))^2,-sin(2*s);...
    (sin(s))^2,(cos(s))^2,sin(2*s);...
    0.5*sin(2*s),-0.5*sin(2*s),cos(2*s)];
N_SET=DM*N_SET;
T_SET=SM*T_SET;
```

8.7 C programming for special T-elements

8.7.1 Potential problems

```
/*
*********************************************************
* Mainfunction MAINFUN                                  *
* - Call other subroutines                              *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int NTREF,NTYPE,NNODE,NEDGE,NODEG,NDIME,NDOFN,NSTRE,
NMATS,NPROP,NGAUS;
void main()
{
    void DUCLEAN(); void ITCLEAN(); void INPUTDT();
    void TYPELEM(); void ELEPARS();
    void HMATRIX(); void GMATRIX(); void KMATRIX();
```

```
void ASMSTIF(); void PVECTOR(); void INDISBC();
void LSSOLVR(); void FIEDNOD(); void FIEDCEN();
void OPRESUT();
FILE *fp;
int NEQNS,NPOIN,NELEM,NVFIX,NPLOD,NDLEG,TNFEG,NEVAB;
int *MATNO,*LNODS,*NOFIX,*IFPRE,*LODPT,*NEASS,*NOPRS,
    *ELNOD,*TSELE;
double *COORD,*PRESC,*POINT,*PRESS,*PROPS,*CHELE;
double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*ESTIF,*GSTIF,
    *GLOAD,*UPOIN,*CECOD,*UCENP,*SCENP,CLENG;
char dummy[201],TITLE[201],file[81];
int i,j,k,N,n1,n2,iELEM,kMATS,kTSEL;

printf("************************************\n");
printf("          Hybrid Trefftz FEM\n");
printf("        for 2D Laplace problems\n");
printf("with general and special Trefftz functions\n");
printf("************************************\n");
/** Input data from file **/
puts("Input file name < dir:fn.txt >: ");
gets(file);
if((fp=fopen(file,"r"))==NULL)
{
    printf("Warning! Can't open input file\n");
    exit(0);
}
// basic parameters
fgets(dummy,200,fp);
fgets(TITLE,200,fp);
fgets(dummy,200,fp);

fgets(dummy,200,fp);
fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
```

```
        &NPLOD,&NDLEG);

    // element connectivity
    MATNO=(int *)calloc(NELEM,sizeof(int));
    ITCLEAN(NELEM,1,MATNO);
    TSELE=(int *)calloc(NELEM,sizeof(int));
    ITCLEAN(NELEM,1,TSELE);
    CHELE=(double *)calloc(NELEM,sizeof(double));
    DUCLEAN(NELEM,1,CHELE);
    LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
    ITCLEAN(NELEM,NNODE,LNODS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NELEM;i++)
    {
        fscanf(fp,"%d %d %d %lf",
            &N,&n1,&TSELE[i],&CHELE[i]);
        MATNO[i]=n1-1;
        for(j=0;j<NNODE;j++)
        {
            fscanf(fp,"%d",&n2);
            LNODS[i*NNODE+j]=n2-1;
        }
        fscanf(fp,"\n");
    }
    // nodal coordinates
    COORD=(double *)calloc(NPOIN*NDIME,sizeof(double));
    DUCLEAN(NPOIN,NDIME,COORD);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NPOIN;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NDIME;j++)
        {
            fscanf(fp,"%lf",&COORD[i*NDIME+j]);
        }
        fscanf(fp,"\n");
    }
    // specified nodal potential/displacement
    NOFIX=(int *)calloc(NVFIX,sizeof(int));
    ITCLEAN(NVFIX,1,NOFIX);
    IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
    ITCLEAN(NVFIX,NDOFN,IFPRE);
    PRESC=(double *)calloc(NVFIX*NDOFN,sizeof(double));
```

```
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    NOFIX[i]=n1-1;
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
    }
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
    }
    fscanf(fp,"\n");
}
// specified concentrated loads at nodes
fgets(dummy,200,fp);
if(NPLOD>0)
{
    LODPT=(int *)calloc(NPLOD*1,sizeof(int));
    ITCLEAN(NPLOD,1,LODPT);
    POINT=(double *)calloc(NPLOD*NDOFN,
        sizeof(double));
    DUCLEAN(NPLOD,NDOFN,POINT);
    fgets(dummy,200,fp);
    for(i=0;i<NPLOD;i++)
    {
        fscanf(fp,"%d %d",&N,&n1);
        LODPT[i]=n1-1;
        for(j=0;j<NDOFN;j++)
        {
            fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
        }
        fscanf(fp,"\n");
    }
}
// specified distributed edge loads
fgets(dummy,200,fp);
if(NDLEG>0)
{
    NEASS=(int *)calloc(NDLEG*1,sizeof(int));
    ITCLEAN(NDLEG,1,NEASS);
    NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
```

```
        ITCLEAN(NDLEG,NODEG,NOPRS);
        TNFEG=NODEG*NDOFN;
        PRESS=(double *)calloc(NDLEG*TNFEG,sizeof(double));
        DUCLEAN(NDLEG,TNFEG,PRESS);
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
        for(j=0;j<NPROP;j++)
        {
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }

    /** Establish local relations of nodes and edges **/
    ELNOD=(int *)calloc(NEDGE*NODEG,sizeof(int));
    ITCLEAN(NEDGE,NODEG,ELNOD);
    TYPELEM(ELNOD);

    /** Form stiffness matrix **/
    NEQNS=NPOIN*NDOFN;
    GSTIF=(double *)calloc(NEQNS*NEQNS,sizeof(double));
    DUCLEAN(NEQNS,NEQNS,GSTIF);
```

```
for(iELEM=0;iELEM<NELEM;iELEM++)
{
    kMATS=MATNO[iELEM];
    kTSEL=TSELE[iELEM];
    CLENG=CHELE[iELEM];
    // Compute some quantities of each element
    ECOOD=(double *)calloc(NNODE*NDIME,
        sizeof(double));
    DUCLEAN(NNODE,NDIME,ECOOD);
    CenCoord=(double *)calloc(1*NDIME,
        sizeof(double));
    DUCLEAN(1,NDIME,CenCoord);
    ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
    // Compute H matrix
    EHMTX=(double *)calloc(NTREF*NTREF,
        sizeof(double));
    DUCLEAN(NTREF,NTREF,EHMTX);
    HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,
        PROPS,EHMTX);
    // Compute G matrix
    NEVAB=NNODE*NDOFN;
    EGMTX=(double *)calloc(NTREF*NEVAB,
        sizeof(double));
    DUCLEAN(NTREF,NEVAB,EGMTX);
    GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,
        PROPS,EGMTX);
    // Compute element stiffness matrix
    ESTIF=(double *)calloc(NEVAB*NEVAB,
        sizeof(double));
    DUCLEAN(NEVAB,NEVAB,ESTIF);
    KMATRIX(EHMTX,EGMTX,ESTIF);
    // Assemble stiffness matrix
    ASMSTIF(iELEM,NEQNS,LNODS,ESTIF,GSTIF);
    free(EHMTX); free(EGMTX); free(ESTIF);
}
// Compute equivalent loads
GLOAD=(double *)calloc(NEQNS*1,sizeof(double));
DUCLEAN(NEQNS,1,GLOAD);
PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,NOPRS,
    PRESS,NPLOD,LODPT,POINT,GLOAD);

// Introduce constrained displacements
INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,GSTIF,GLOAD);

// Solve linear system of equations
```

```
    LSSOLVR(GSTIF,GLOAD,NEQNS);

    // Output nodal displacement
    UPOIN=(double *)calloc(NPOIN*NDOFN,sizeof(double));
    DUCLEAN(NPOIN,NDOFN,UPOIN);
    FIEDNOD(NPOIN,GLOAD,UPOIN);

    // Compute quantities at centroid of each element
    CECOD=(double *)calloc(NELEM*NDIME,sizeof(double));
    DUCLEAN(NELEM,NDIME,CECOD);
    UCENP=(double *)calloc(NELEM*NDOFN,sizeof(double));
    DUCLEAN(NELEM,NDOFN,UCENP);
    SCENP=(double *)calloc(NELEM*NSTRE,sizeof(double));
    DUCLEAN(NELEM,NSTRE,SCENP);
    FIEDCEN(NELEM,MATNO,TSELE,CHELE,LNODS,COORD,PROPS,
        ELNOD,GLOAD,CECOD,UCENP,SCENP);

    // Output results
    OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,
        NVFIX,NPLOD,NDLEG);

    free(COORD); free(LNODS); free(MATNO);
    free(NOFIX); free(IFPRE); free(PRESC);
    free(PROPS); free(ECOOD); free(CenCoord);
    free(GSTIF); free(GLOAD); free(UPOIN);
    free(CECOD); free(UCENP); free(SCENP);
    printf("------------ Finished ------------\n");
    return;
}


/*
*******************************************************
* Subroutine HMATRIX                                  *
* - Compute H matrix for each element                 *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void HMATRIX(double ECOOD[],int ELNOD[],int kMATS,
             int kTSEL,double CLENG,double PROPS[],
             double EHMTX[])
{
    void DUCLEAN();
```

```
void GAUSSQU();
void QUANGAS();
void TREFFTZ();
void SCHTREF();
void MATMULT();
void MATTRAN();
extern int NTREF,NDIME,NDOFN,NNODE,NEDGE,NGAUS,
    NPROP,NSTRE;
double *POSGP,*WEIGP,THICK,EXISP,*CORGS,DVOLU,
    *AMTRX,*SHMTX,*N_SET,*T_SET,*Q_SET,*TQ_SET,*QTN;
int iEDGE,iGAUS,im,jm,n1,NEVAB;

NEVAB=NNODE*NDOFN;
// Gaussian point and weight coefficients
POSGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,POSGP);
WEIGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,WEIGP);
GAUSSQU(POSGP,WEIGP);
// Material properties
THICK=PROPS[kMATS*NPROP+2];
// Compute H matrix
for(iEDGE=0;iEDGE<NEDGE;iEDGE++)
{
    for(iGAUS=0;iGAUS<NGAUS;iGAUS++)
    {
        EXISP=POSGP[iGAUS];
        //
        CORGS=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CORGS);
        AMTRX=(double *)calloc(NDOFN*NSTRE,
            sizeof(double));
        DUCLEAN(NDOFN,NSTRE,AMTRX);
        SHMTX=(double *)calloc(NDOFN*NEVAB,
            sizeof(double));
        DUCLEAN(NDOFN,NEVAB,SHMTX);
        QUANGAS(iEDGE,EXISP,ECOOD,ELNOD,CORGS,
            &DVOLU,AMTRX,SHMTX);
        DVOLU=DVOLU*WEIGP[iGAUS];
        if((THICK+1)!=1)
        {
            DVOLU=DVOLU*THICK;
        }
        // Trefftz functions
```

```
            N_SET=(double *)calloc(NDOFN*NTREF,
                sizeof(double));
            DUCLEAN(NDOFN,NTREF,N_SET);
            T_SET=(double *)calloc(NSTRE*NTREF,
                sizeof(double));
            DUCLEAN(NSTRE,NTREF,T_SET);
            if(kTSEL==0) // general element
            {
                TREFFTZ(CORGS[0],CORGS[1],N_SET,T_SET);
            }
            else if(kTSEL==1)
            { // special circular hole element
                SCHTREF(CORGS[0],CORGS[1],CLENG,
                    N_SET,T_SET);
            }
            //
            Q_SET=(double *)calloc(NDOFN*NTREF,
                sizeof(double));
            DUCLEAN(NDOFN,NTREF,Q_SET);
            MATMULT(AMTRX,T_SET,NDOFN,NSTRE,NTREF,Q_SET);
            //
            TQ_SET=(double *)calloc(NTREF*NDOFN,
                sizeof(double));
            DUCLEAN(NTREF,NDOFN,TQ_SET);
            MATTRAN(Q_SET,NDOFN,NTREF,TQ_SET);
            //
            QTN=(double *)calloc(NTREF*NTREF,
                sizeof(double));
            DUCLEAN(NTREF,NTREF,QTN);
            MATMULT(TQ_SET,N_SET,NTREF,NDOFN,NTREF,QTN);
            for(im=0;im<NTREF;im++)
            {
                for(jm=0;jm<NTREF;jm++)
                {
                    n1=im*NTREF+jm;
                    EHMTX[n1]=EHMTX[n1]+QTN[n1]*DVOLU;
                }
            }
            free(CORGS); free(AMTRX);
            free(SHMTX); free(N_SET);
            free(T_SET); free(Q_SET);
            free(TQ_SET); free(QTN);
        }
    }
    free(POSGP); free(WEIGP);
```

```
    return;
}


/*
*******************************************************
* Subroutine GMATRIX                                  *
* - Compute G matrix for each element                 *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void GMATRIX(double ECOOD[],int ELNOD[],int kMATS,
             int kTSEL,double CLENG,double PROPS[],
             double EGMTX[])
{
    void DUCLEAN();
    void GAUSSQU();
    void QUANGAS();
    void TREFFTZ();
    void SCHTREF();
    void MATMULT();
    void MATTRAN();
    extern int NTREF,NDIME,NDOFN,NNODE,NEDGE,NGAUS,
        NPROP,NSTRE;
    double *POSGP,*WEIGP,THICK,EXISP,*CORGS,DVOLU,
        *AMTRX,*SHMTX,*N_SET,*T_SET,*Q_SET,*TQ_SET,*QTS;
    int ii,jj,im,jn,n1,NEVAB;

    NEVAB=NNODE*NDOFN;
    // Gaussian point and weight coefficients
    POSGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,POSGP);
    WEIGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,WEIGP);
    GAUSSQU(POSGP,WEIGP);
    // Material properties
    THICK=PROPS[kMATS*NPROP+2];
    // Compute H matrix
    for(ii=0;ii<NEDGE;ii++)
    {
        for(jj=0;jj<NGAUS;jj++)
        {
            EXISP=POSGP[jj];
```

```
// Related quantities at Gaussian point
CORGS=(double *)calloc(1*NDIME,
    sizeof(double));
DUCLEAN(1,NDIME,CORGS);
AMTRX=(double *)calloc(NDOFN*NSTRE,
    sizeof(double));
DUCLEAN(NDOFN,NSTRE,AMTRX);
SHMTX=(double *)calloc(NDOFN*NEVAB,
    sizeof(double));
DUCLEAN(NDOFN,NEVAB,SHMTX);
QUANGAS(ii,EXISP,ECOOD,ELNOD,CORGS,&DVOLU,
    AMTRX,SHMTX);
DVOLU=DVOLU*WEIGP[jj];
if((THICK+1)!=1)
{
    DVOLU=DVOLU*THICK;
}
// Trefftz functions
N_SET=(double *)calloc(NDOFN*NTREF,
    sizeof(double));
DUCLEAN(NDOFN,NTREF,N_SET);
T_SET=(double *)calloc(NSTRE*NTREF,
    sizeof(double));
DUCLEAN(NSTRE,NTREF,T_SET);
if(kTSEL==0) // general element
{
    TREFFTZ(CORGS[0],CORGS[1],N_SET,T_SET);
}
else if(kTSEL==1)
{ // special circular hole element
    SCHTREF(CORGS[0],CORGS[1],CLENG,
        N_SET,T_SET);
}
// Q=A*T
Q_SET=(double *)calloc(NDOFN*NTREF,
    sizeof(double));
DUCLEAN(NDOFN,NTREF,Q_SET);
MATMULT(AMTRX,T_SET,NDOFN,NSTRE,NTREF,Q_SET);
// Q'
TQ_SET=(double *)calloc(NTREF*NDOFN,
    sizeof(double));
DUCLEAN(NTREF,NDOFN,TQ_SET);
MATTRAN(Q_SET,NDOFN,NTREF,TQ_SET);
// Q'*SHAPE
QTS=(double *)calloc(NTREF*NEVAB,
```

```
                sizeof(double));
            DUCLEAN(NTREF,NEVAB,QTS);
            MATMULT(TQ_SET,SHMTX,NTREF,NDOFN,NEVAB,QTS);
            //
            for(im=0;im<NTREF;im++)
            {
                for(jn=0;jn<NEVAB;jn++)
                {
                    n1=im*NEVAB+jn;
                    EGMTX[n1]=EGMTX[n1]+QTS[n1]*DVOLU;
                }
            }
            free(CORGS); free(AMTRX);
            free(SHMTX); free(N_SET);
            free(T_SET); free(Q_SET);
            free(TQ_SET); free(QTS);
        }
    }
    free(POSGP); free(WEIGP);
    return;
}


/*
*******************************************************
* Subroutine FIEDCEN                                  *
* -Compute related fields at centroid of each element *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDCEN(int NELEM,int MATNO[],int TSELE[],
             double CHELE[],int LNODS[],double COORD[],
             double PROPS[],int ELNOD[],double ASDIS[],
             double CECOD[],double UCENP[],
             double SCENP[])
{
    void ELEPARS(); void DUCLEAN(); void HMATRIX();
    void GMATRIX(); void EDISNOD(); void CMATRIX();
    void RIGIDRV(); void TREFFTZ(); void MATMULT();
    void SCHTREF();
    extern int NTREF,NNODE,NDIME,NDOFN,NSTRE;
    int iELEM,kMATS,kTSEL,NEVAB;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*d_Ele,
```

```
        *c_Ele,c0,*N_SET,*T_SET,*GDISP,*GSTRE,
        xp,yp,CLENG;

    NEVAB=NNODE*NDOFN;
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        kTSEL=TSELE[iELEM];
        CLENG=CHELE[iELEM];
        // Compute some quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
        HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS,EHMTX);
        // Compute G matrix
        EGMTX=(double *)calloc(NTREF*NEVAB,sizeof(double));
        DUCLEAN(NTREF,NEVAB,EGMTX);
        GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS,EGMTX);
        // Nodal displacements
        d_Ele=(double *)calloc(NEVAB,sizeof(double));
        DUCLEAN(NEVAB,1,d_Ele);
        EDISNOD(iELEM,LNODS,ASDIS,d_Ele);
        // Calculate the ce coefficients
        c_Ele=(double *)calloc(NTREF*1,sizeof(double));
        DUCLEAN(NTREF,1,c_Ele);
        CMATRIX(EHMTX,EGMTX,d_Ele,c_Ele);
        // Recover rigid displacement
        RIGIDRV(ECOOD,c_Ele,d_Ele,kTSEL,CLENG,&c0);
        // Compute Trefftz internal fields at central point
        N_SET=(double *)calloc(NDOFN*NTREF,sizeof(double));
        DUCLEAN(NDOFN,NTREF,N_SET);
        T_SET=(double *)calloc(NSTRE*NTREF,sizeof(double));
        DUCLEAN(NSTRE,NTREF,T_SET);
        if(kTSEL==0) // general element
        {
            xp=0;
            yp=0;
            TREFFTZ(xp,yp,N_SET,T_SET);
        }
        else if(kTSEL==1)// circular hole element
```

```
        {
            xp=0;
            yp=CLENG;
            SCHTREF(xp,yp,CLENG,N_SET,T_SET);
        }
        GDISP=(double *)calloc(NDOFN*1,sizeof(double));
        DUCLEAN(NDOFN,1,GDISP);
        GSTRE=(double *)calloc(NSTRE*1,sizeof(double));
        DUCLEAN(NSTRE,1,GSTRE);
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,GDISP);
        MATMULT(T_SET,c_Ele,NSTRE,NTREF,1,GSTRE);
        UCENP[iELEM*NDOFN+0]=GDISP[0]+c0;
        SCENP[iELEM*NSTRE+0]=GSTRE[0];
        SCENP[iELEM*NSTRE+1]=GSTRE[1];
        // Coordinates of commputing point
        CECOD[iELEM*NDIME+0]=CenCoord[0]+xp;
        CECOD[iELEM*NDIME+1]=CenCoord[1]+yp;
    }
    free(ECOOD); free(CenCoord); free(EHMTX);
    free(EGMTX); free(d_Ele); free(c_Ele);
    free(N_SET); free(T_SET); free(GDISP);
    free(GSTRE);
}


/*
*********************************************************
* Subroutine RIGIDRV                                    *
* - Recovery of rigid body motion                       *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void RIGIDRV(double ECOOD[],double c_Ele[],
             double d_Ele[],int kTSEL,double CLENG,
             double *c0)
{
    void DUCLEAN();
    void MATMULT();
    void TREFFTZ();
    void SCHTREF();
    extern int NTREF,NDIME,NDOFN,NNODE;
    double sum,xp,yp,*N_SET,*T_SET,*TEMP;
    int iNODE;
```

```
    sum=0.0;
    for(iNODE=0;iNODE<NNODE;iNODE++)
    {
        xp=ECOOD[iNODE*NDIME+0];
        yp=ECOOD[iNODE*NDIME+1];
        N_SET=(double *)calloc(1*NTREF,sizeof(double));
        DUCLEAN(1,NTREF,N_SET);
        T_SET=(double *)calloc(2*NTREF,sizeof(double));
        DUCLEAN(2,NTREF,T_SET);
        if(kTSEL==0) // general element
        {
            TREFFTZ(xp,yp,N_SET,T_SET);
        }
        else if(kTSEL==1)// circular hole element
        {
            SCHTREF(xp,yp,CLENG,N_SET,T_SET);
        }
        TEMP=(double *)calloc(1*1,sizeof(double));
        DUCLEAN(1,1,TEMP);
        MATMULT(N_SET,c_Ele,1,NTREF,1,TEMP);
        sum=sum+(d_Ele[iNODE]-TEMP[0]);
    }
    *c0=sum/(NNODE*1.0);
    free(N_SET); free(T_SET);
    return;
}


/*
*********************************************************
* Subroutine SCHTREF                                    *
* - Evaluate circular hole Trefftz functions            *
*   truncated with specified terms number               *
*********************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void SCHTREF(double sp,double tp,double b,
             double N_SET[],double T_SET[])
{
        extern int NTREF,NNODE,NDOFN;
    int temp,im,n;
    double remaind,r,s;
    // Check the number of terms of Trefftz functions
```

```
    temp=NNODE*NDOFN-1;
    if(NTREF<temp)
    {
        printf("Small number of Trefftz functions!");
        exit(0);
    }
    //
    r=sqrt(pow(sp,2.0)+pow(tp,2.0));
    s=atan2(tp,sp);
    for(im=0;im<NTREF;im++)
    {
        // Determine the order of Ni
        n=(int)(ceil((im+1)/2.0));
        // Justify im is odd or even number
        remaind=fmod(im,2.0);
        if((1+remaind)!=1) // im is even
        {
            N_SET[im]=
                (pow(r,n)+pow(b,(2*n))*pow(r,(-n)))
                *sin(n*s);
            T_SET[0*NTREF+im]=pow(r,(n-1))*n*sin(n*s-s)
                -pow(b,(2*n))*pow(r,(-n-1))*n*sin(n*s+s);
            T_SET[1*NTREF+im]=pow(r,(n-1))*n*cos(n*s-s)
                +pow(b,(2*n))*pow(r,(-n-1))*n*cos(n*s+s);
        }
        else // im is odd
        {
            N_SET[im]=
                (pow(r,n)+pow(b,(2*n))*pow(r,(-n)))
                *cos(n*s);
            T_SET[0*NTREF+im]= pow(r,(n-1))*n*cos(n*s-s)
                -pow(b,(2*n))*pow(r,(-n-1))*n*cos(n*s+s);
            T_SET[1*NTREF+im]=-pow(r,(n-1))*n*sin(n*s-s)
                -pow(b,(2*n))*pow(r,(-n-1))*n*sin(n*s+s);
        }
    }
    return;
}
```

8.7.2 Elastic problems

```
/*
*******************************************************
* Mainfunction MAINFUN                                 *
```

```
* - Call other subroutines                                    *
*************************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int NTREF,NTYPE,NNODE,NEDGE,NODEG,NDIME,NDOFN,NSTRE,
NMATS,NPROP,NGAUS;
void main()
{
    void DUCLEAN(); void ITCLEAN(); void INPUTDT();
    void TYPELEM(); void ELEPARS();
    void HMATRIX(); void GMATRIX(); void KMATRIX();
    void ASMSTIF(); void PVECTOR(); void INDISBC();
    void LSSOLVR(); void FIEDNOD(); void FIEDCEN();
    void OPRESUT();
    FILE *fp;
    int NEQNS,NPOIN,NELEM,NVFIX,NPLOD,NDLEG,TNFEG,NEVAB;
    int *MATNO,*LNODS,*NOFIX,*IFPRE,*LODPT,*NEASS,*NOPRS,
        *ELNOD,*TSELE;
    double *COORD,*PRESC,*POINT,*PRESS,*PROPS;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*ESTIF,*GSTIF,
        *GLOAD,*UPOIN,*CECOD,*UCENP,*SCENP,*CHELE,CLENG;
    char dummy[201],TITLE[201],file[81];
    int i,j,k,N,n1,n2,iELEM,kMATS,kTSEL;

    printf("************************************\n");
    printf("        Hybrid Trefftz FEM\n");
    printf("      for 2D elastic problems\n");
    printf("  with general and special elements\n");
    printf("************************************\n");
    /** Input data from file **/
    puts("Input file name < dir:fn.txt >: ");
    gets(file);
    if((fp=fopen(file,"r"))==NULL)
    {
        printf("Warning! Can't open input file\n");
        exit(0);
    }
    // basic parameters
    fgets(dummy,200,fp);
    fgets(TITLE,200,fp);
    fgets(dummy,200,fp);

    fgets(dummy,200,fp);
```

```
fscanf(fp,"%d %d\n",&NTREF,&NTYPE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NNODE,&NEDGE,&NODEG);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NDIME,&NDOFN,&NSTRE);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d\n",&NMATS,&NPROP,&NGAUS);

fgets(dummy,200,fp);
fscanf(fp,"%d %d %d %d %d\n",&NPOIN,&NELEM,&NVFIX,
    &NPLOD,&NDLEG);

// element connectivity
MATNO=(int *)calloc(NELEM,sizeof(int));
ITCLEAN(NELEM,1,MATNO);
TSELE=(int *)calloc(NELEM,sizeof(int));
ITCLEAN(NELEM,1,TSELE);
CHELE=(double *)calloc(NELEM,sizeof(double));
DUCLEAN(NELEM,1,CHELE);
LNODS=(int *)calloc(NELEM*NNODE,sizeof(int));
ITCLEAN(NELEM,NNODE,LNODS);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NELEM;i++)
{
    fscanf(fp,"%d %d %d %lf",
        &N,&n1,&TSELE[i],&CHELE[i]);
    MATNO[i]=n1-1;
    for(j=0;j<NNODE;j++)
    {
        fscanf(fp,"%d",&n2);
        LNODS[i*NNODE+j]=n2-1;
    }
    fscanf(fp,"\n");
}
// nodal coordinates
COORD=(double *)calloc(NPOIN*NDIME,
    sizeof(double));
DUCLEAN(NPOIN,NDIME,COORD);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NPOIN;i++)
```

```
{
    fscanf(fp,"%d",&N);
    for(j=0;j<NDIME;j++)
    {
        fscanf(fp,"%lf",&COORD[i*NDIME+j]);
    }
    fscanf(fp,"\n");
}
// specified nodal potential/displacement
NOFIX=(int *)calloc(NVFIX,sizeof(int));
ITCLEAN(NVFIX,1,NOFIX);
IFPRE=(int *)calloc(NVFIX*NDOFN,sizeof(int));
ITCLEAN(NVFIX,NDOFN,IFPRE);
PRESC=(double *)calloc(NVFIX*NDOFN,
    sizeof(double));
DUCLEAN(NVFIX,NDOFN,PRESC);
fgets(dummy,200,fp);
fgets(dummy,200,fp);
for(i=0;i<NVFIX;i++)
{
    fscanf(fp,"%d %d",&N,&n1);
    NOFIX[i]=n1-1;
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%d",&IFPRE[i*NDOFN+j]);
    }
    for(j=0;j<NDOFN;j++)
    {
        fscanf(fp,"%lf",&PRESC[i*NDOFN+j]);
    }
    fscanf(fp,"\n");
}
// specified concentrated loads at nodes
fgets(dummy,200,fp);
if(NPLOD>0)
{
    LODPT=(int *)calloc(NPLOD*1,sizeof(int));
    ITCLEAN(NPLOD,1,LODPT);
    POINT=(double *)calloc(NPLOD*NDOFN,
        sizeof(double));
    DUCLEAN(NPLOD,NDOFN,POINT);
    fgets(dummy,200,fp);
    for(i=0;i<NPLOD;i++)
    {
        fscanf(fp,"%d %d",&N,&n1);
```

```
            LODPT[i]=n1-1;
            for(j=0;j<NDOFN;j++)
            {
                fscanf(fp,"%lf",&POINT[i*NDOFN+j]);
            }
            fscanf(fp,"\n");
        }
    }
    // specified distributed edge loads
    fgets(dummy,200,fp);
    if(NDLEG>0)
    {
        NEASS=(int *)calloc(NDLEG*1,sizeof(int));
        ITCLEAN(NDLEG,1,NEASS);
        NOPRS=(int *)calloc(NDLEG*NODEG,sizeof(int));
        ITCLEAN(NDLEG,NODEG,NOPRS);
        TNFEG=NODEG*NDOFN;
        PRESS=(double *)calloc(NDLEG*TNFEG,
            sizeof(double));
        DUCLEAN(NDLEG,TNFEG,PRESS);
        fgets(dummy,200,fp);
        for(i=0;i<NDLEG;i++)
        {
            fscanf(fp,"%d %d",&N,&n1);
            NEASS[i]=n1-1;
            for(j=0;j<NODEG;j++)
            {
                fscanf(fp,"%d",&n2);
                NOPRS[i*NODEG+j]=n2-1;
            }
            for(k=0;k<TNFEG;k++)
            {
                fscanf(fp,"%lf",&PRESS[i*TNFEG+k]);
            }
            fscanf(fp,"\n");
        }
    }
    // material properties
    PROPS=(double *)calloc(NMATS*NPROP,sizeof(double));
    DUCLEAN(NMATS,NPROP,PROPS);
    fgets(dummy,200,fp);
    fgets(dummy,200,fp);
    for(i=0;i<NMATS;i++)
    {
        fscanf(fp,"%d",&N);
```

```
        for(j=0;j<NPROP;j++)
        {
            fscanf(fp,"%lf",&PROPS[i*NPROP+j]);
        }
        fscanf(fp,"\n");
    }

    /** Establish local relations of nodes and edges **/
    ELNOD=(int *)calloc(NEDGE*NODEG,sizeof(int));
    ITCLEAN(NEDGE,NODEG,ELNOD);
    TYPELEM(ELNOD);

    /** Form stiffness matrix **/
    NEQNS=NPOIN*NDOFN;
    GSTIF=(double *)calloc(NEQNS*NEQNS,sizeof(double));
    DUCLEAN(NEQNS,NEQNS,GSTIF);
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        kTSEL=TSELE[iELEM];
        CLENG=CHELE[iELEM];
        // Compute some quantities related to each element
        ECOOD=(double *)calloc(NNODE*NDIME,sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,
            sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
        HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,
            PROPS,EHMTX);
        // Compute G matrix
        NEVAB=NNODE*NDOFN;
        EGMTX=(double *)calloc(NTREF*NEVAB,
            sizeof(double));
        DUCLEAN(NTREF,NEVAB,EGMTX);
        GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,
            PROPS,EGMTX);
        // Compute element stiffness matrix
        ESTIF=(double *)calloc(NEVAB*NEVAB,
            sizeof(double));
        DUCLEAN(NEVAB,NEVAB,ESTIF);
```

```
        KMATRIX(EHMTX,EGMTX,ESTIF);
        // Assemble stiffness matrix
        ASMSTIF(iELEM,NEQNS,LNODS,ESTIF,GSTIF);
        free(EHMTX); free(EGMTX); free(ESTIF);
    }
    // Compute equivalent loads
    GLOAD=(double *)calloc(NEQNS*1,sizeof(double));
    DUCLEAN(NEQNS,1,GLOAD);
    PVECTOR(MATNO,PROPS,LNODS,COORD,NDLEG,NEASS,NOPRS,
        PRESS,NPLOD,LODPT,POINT,GLOAD);

    // Introduce constrained displacements
    INDISBC(NEQNS,NVFIX,NOFIX,IFPRE,PRESC,GSTIF,GLOAD);

    // Solve linear system of equations
    LSSOLVR(GSTIF,GLOAD,NEQNS);

    // Output nodal displacement
    UPOIN=(double *)calloc(NPOIN*NDOFN,sizeof(double));
    DUCLEAN(NPOIN,NDOFN,UPOIN);
    FIEDNOD(NPOIN,GLOAD,UPOIN);

    // Compute quantities at centroid of each element
    CECOD=(double *)calloc(NELEM*NDIME,sizeof(double));
    DUCLEAN(NELEM,NDIME,CECOD);
    UCENP=(double *)calloc(NELEM*NDOFN,sizeof(double));
    DUCLEAN(NELEM,NDOFN,UCENP);
    SCENP=(double *)calloc(NELEM*NSTRE,sizeof(double));
    DUCLEAN(NELEM,NSTRE,SCENP);
    FIEDCEN(NELEM,MATNO,TSELE,CHELE,LNODS,COORD,PROPS,
        ELNOD,GLOAD,CECOD,UCENP,SCENP);

    // Output results
    OPRESUT(NPOIN,COORD,UPOIN,NELEM,CECOD,UCENP,SCENP,
        NVFIX,NPLOD,NDLEG);

    free(COORD); free(LNODS); free(MATNO);
    free(NOFIX); free(IFPRE); free(PRESC);
    free(PROPS); free(ECOOD); free(CenCoord);
    free(GSTIF); free(GLOAD); free(UPOIN);
    free(CECOD); free(UCENP); free(SCENP);
    printf("------------ Finished ------------\n");
    return;
}
```

```
/*
*******************************************************
* Subroutine HMATRIX                                  *
* - Compute H matrix for each element                 *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void HMATRIX(double ECOOD[],int ELNOD[],int kMATS,
             int kTSEL,double CLENG,
             double PROPS[],double EHMTX[])
{
    void DUCLEAN();
    void GAUSSQU();
    void QUANGAS();
    void TREFFTZ();
    void SCHTREF();
    void MATMULT();
    void MATTRAN();
    extern int NTREF,NDIME,NDOFN,NNODE,NEDGE,NGAUS,
        NPROP,NSTRE;
    double *POSGP,*WEIGP,THICK,EXISP,*CORGS,DVOLU,
        *AMTRX,*SHMTX,*N_SET,*T_SET,*Q_SET,*TQ_SET,
        *QTN;
    int iEDGE,iGAUS,im,jm,n1,NEVAB;

    NEVAB=NNODE*NDOFN;
    // Gaussian point and weight coefficients
    POSGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,POSGP);
    WEIGP=(double *)calloc(NGAUS,sizeof(double));
    DUCLEAN(NGAUS,1,WEIGP);
    GAUSSQU(POSGP,WEIGP);
    // Material properties
    THICK=PROPS[kMATS*NPROP+2];
    // Compute H matrix
    for(iEDGE=0;iEDGE<NEDGE;iEDGE++)
    {
        for(iGAUS=0;iGAUS<NGAUS;iGAUS++)
        {
            EXISP=POSGP[iGAUS];
            //
            CORGS=(double *)calloc(1*NDIME,
```

```
    sizeof(double));
DUCLEAN(1,NDIME,CORGS);
AMTRX=(double *)calloc(NDOFN*NSTRE,
    sizeof(double));
DUCLEAN(NDOFN,NSTRE,AMTRX);
SHMTX=(double *)calloc(NDOFN*NEVAB,
    sizeof(double));
DUCLEAN(NDOFN,NEVAB,SHMTX);
QUANGAS(iEDGE,EXISP,ECOOD,ELNOD,CORGS,
    &DVOLU,AMTRX,SHMTX);
DVOLU=DVOLU*WEIGP[iGAUS];
if((THICK+1)!=1)
{
    DVOLU=DVOLU*THICK;
}
// Trefftz functions
N_SET=(double *)calloc(NDOFN*NTREF,
    sizeof(double));
DUCLEAN(NDOFN,NTREF,N_SET);
T_SET=(double *)calloc(NSTRE*NTREF,
    sizeof(double));
DUCLEAN(NSTRE,NTREF,T_SET);
if(kTSEL==0) // general element
{
    TREFFTZ(CORGS[0],CORGS[1],kMATS,
        PROPS,N_SET,T_SET);
}
else if(kTSEL==1) // special element
{
    SCHTREF(CORGS[0],CORGS[1],CLENG,kMATS,
        PROPS,N_SET,T_SET);
}
//
Q_SET=(double *)calloc(NDOFN*NTREF,
    sizeof(double));
DUCLEAN(NDOFN,NTREF,Q_SET);
MATMULT(AMTRX,T_SET,NDOFN,NSTRE,NTREF,
    Q_SET);
//
TQ_SET=(double *)calloc(NTREF*NDOFN,
    sizeof(double));
DUCLEAN(NTREF,NDOFN,TQ_SET);
MATTRAN(Q_SET,NDOFN,NTREF,TQ_SET);
//
QTN=(double *)calloc(NTREF*NTREF,
```

```
                sizeof(double));
            DUCLEAN(NTREF,NTREF,QTN);
            MATMULT(TQ_SET,N_SET,NTREF,NDOFN,NTREF,
                QTN);
            for(im=0;im<NTREF;im++)
            {
                for(jm=0;jm<NTREF;jm++)
                {
                    n1=im*NTREF+jm;
                    EHMTX[n1]=EHMTX[n1]+QTN[n1]*DVOLU;
                }
            }
            free(CORGS); free(AMTRX);
            free(SHMTX); free(N_SET);
            free(T_SET); free(Q_SET);
            free(TQ_SET); free(QTN);
        }
    }
    free(POSGP); free(WEIGP);
    return;
}


/*
*******************************************************
* Subroutine GMATRIX                                  *
* - Compute G matrix for each element                 *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void GMATRIX(double ECOOD[],int ELNOD[],int kMATS,
             int kTSEL,double CLENG,double PROPS[],
             double EGMTX[])
{
    void DUCLEAN();
    void GAUSSQU();
    void QUANGAS();
    void TREFFTZ();
    void SCHTREF();
    void MATMULT();
    void MATTRAN();
    extern int NTREF,NDIME,NDOFN,NNODE,NEDGE,NGAUS,
        NPROP,NSTRE;
```

```
double *POSGP,*WEIGP,THICK,EXISP,*CORGS,DVOLU,
    *AMTRX,*SHMTX,*N_SET,*T_SET,*Q_SET,*TQ_SET,
    *QTS;
int ii,jj,im,jn,n1,NEVAB;

NEVAB=NNODE*NDOFN;
// Gaussian point and weight coefficients
POSGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,POSGP);
WEIGP=(double *)calloc(NGAUS,sizeof(double));
DUCLEAN(NGAUS,1,WEIGP);
GAUSSQU(POSGP,WEIGP);
// Material properties
THICK=PROPS[kMATS*NPROP+2];
// Compute H matrix
for(ii=0;ii<NEDGE;ii++)
{
    for(jj=0;jj<NGAUS;jj++)
    {
        EXISP=POSGP[jj];
        // Related quantities at Gaussian point
        CORGS=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CORGS);
        AMTRX=(double *)calloc(NDOFN*NSTRE,
            sizeof(double));
        DUCLEAN(NDOFN,NSTRE,AMTRX);
        SHMTX=(double *)calloc(NDOFN*NEVAB,
            sizeof(double));
        DUCLEAN(NDOFN,NEVAB,SHMTX);
        QUANGAS(ii,EXISP,ECOOD,ELNOD,CORGS,&DVOLU,
            AMTRX,SHMTX);
        DVOLU=DVOLU*WEIGP[jj];
        if((THICK+1)!=1)
        {
            DVOLU=DVOLU*THICK;
        }
        // Trefftz functions
        N_SET=(double *)calloc(NDOFN*NTREF,
            sizeof(double));
        DUCLEAN(NDOFN,NTREF,N_SET);
        T_SET=(double *)calloc(NSTRE*NTREF,
            sizeof(double));
        DUCLEAN(NSTRE,NTREF,T_SET);
        if(kTSEL==0) // general element
```

```
            {
                TREFFTZ(CORGS[0],CORGS[1],kMATS,
                    PROPS,N_SET,T_SET);
            }
            else if(kTSEL==1) // special element
            {
                SCHTREF(CORGS[0],CORGS[1],CLENG,kMATS,
                    PROPS,N_SET,T_SET);
            }
            // Q=A*T
            Q_SET=(double *)calloc(NDOFN*NTREF,
                sizeof(double));
            DUCLEAN(NDOFN,NTREF,Q_SET);
            MATMULT(AMTRX,T_SET,NDOFN,NSTRE,NTREF,
                Q_SET);
            // Q'
            TQ_SET=(double *)calloc(NTREF*NDOFN,
                sizeof(double));
            DUCLEAN(NTREF,NDOFN,TQ_SET);
            MATTRAN(Q_SET,NDOFN,NTREF,TQ_SET);
            // Q'*SHAPE
            QTS=(double *)calloc(NTREF*NEVAB,
                sizeof(double));
            DUCLEAN(NTREF,NEVAB,QTS);
            MATMULT(TQ_SET,SHMTX,NTREF,NDOFN,NEVAB,
                QTS);
            //
            for(im=0;im<NTREF;im++)
            {
                for(jn=0;jn<NEVAB;jn++)
                {
                    n1=im*NEVAB+jn;
                    EGMTX[n1]=EGMTX[n1]+QTS[n1]*DVOLU;
                }
            }
            free(CORGS); free(AMTRX);
            free(SHMTX); free(N_SET);
            free(T_SET); free(Q_SET);
            free(TQ_SET); free(QTS);
        }
    }
    free(POSGP); free(WEIGP);
    return;
}
```

```
/*
*******************************************************
* Subroutine FIEDCEN                                  *
* -Compute related fields at centroid of each element *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void FIEDCEN(int NELEM,int MATNO[],int TSELE[],
             double CHELE[],int LNODS[],double COORD[],
             double PROPS[],int ELNOD[],double ASDIS[],
             double CECOD[],double UCENP[],
             double SCENP[])
{
    void ELEPARS(); void DUCLEAN(); void HMATRIX();
    void GMATRIX(); void EDISNOD(); void CMATRIX();
    void RIGIDRV(); void TREFFTZ(); void MATMULT();
    void SCHTREF();
    extern int NTREF,NNODE,NDIME,NDOFN,NSTRE;
    int iELEM,kMATS,NEVAB,kTSEL;
    double *ECOOD,*CenCoord,*EHMTX,*EGMTX,*d_Ele,
        *c_Ele,*c0,*N_SET,*T_SET,*GDISP,*GSTRE,
        xp,yp,CLENG;

    NEVAB=NNODE*NDOFN;
    for(iELEM=0;iELEM<NELEM;iELEM++)
    {
        kMATS=MATNO[iELEM];
        kTSEL=TSELE[iELEM];
        CLENG=CHELE[iELEM];
        // Compute some quantities of each element
        ECOOD=(double *)calloc(NNODE*NDIME,
            sizeof(double));
        DUCLEAN(NNODE,NDIME,ECOOD);
        CenCoord=(double *)calloc(1*NDIME,
            sizeof(double));
        DUCLEAN(1,NDIME,CenCoord);
        ELEPARS(iELEM,LNODS,COORD,ECOOD,CenCoord);
        // Compute H matrix
        EHMTX=(double *)calloc(NTREF*NTREF,
            sizeof(double));
        DUCLEAN(NTREF,NTREF,EHMTX);
        HMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS,
```

```
    EHMTX);
// Compute G matrix
EGMTX=(double *)calloc(NTREF*NEVAB,
    sizeof(double));
DUCLEAN(NTREF,NEVAB,EGMTX);
GMATRIX(ECOOD,ELNOD,kMATS,kTSEL,CLENG,PROPS,
    EGMTX);
// Nodal displacements
d_Ele=(double *)calloc(NEVAB,sizeof(double));
DUCLEAN(NEVAB,1,d_Ele);
EDISNOD(iELEM,LNODS,ASDIS,d_Ele);
// Calculate the ce coefficients
c_Ele=(double *)calloc(NTREF*1,sizeof(double));
DUCLEAN(NTREF,1,c_Ele);
CMATRIX(EHMTX,EGMTX,d_Ele,c_Ele);
// Recover rigid displacement
c0=(double *)calloc(3*1,sizeof(double));
DUCLEAN(3,1,c0);
RIGIDRV(ECOOD,c_Ele,d_Ele,kMATS,kTSEL,CLENG,
    PROPS,c0);
// Compute Trefftz internal fields
// at central point
N_SET=(double *)calloc(NDOFN*NTREF,
    sizeof(double));
DUCLEAN(NDOFN,NTREF,N_SET);
T_SET=(double *)calloc(NSTRE*NTREF,
    sizeof(double));
DUCLEAN(NSTRE,NTREF,T_SET);
if(kTSEL==0) // general element
{
    xp=0;
    yp=0;
    TREFFTZ(xp,yp,kMATS,PROPS,N_SET,T_SET);
}
else if(kTSEL==1) // special element
{
    xp=0;
    yp=CLENG;
    SCHTREF(xp,yp,CLENG,kMATS,PROPS,
        N_SET,T_SET);
}
GDISP=(double *)calloc(NDOFN*1,sizeof(double));
DUCLEAN(NDOFN,1,GDISP);
GSTRE=(double *)calloc(NSTRE*1,sizeof(double));
DUCLEAN(NSTRE,1,GSTRE);
```

```
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,GDISP);
        MATMULT(T_SET,c_Ele,NSTRE,NTREF,1,GSTRE);
        UCENP[iELEM*NDOFN+0]=GDISP[0]+c0[0]+yp*c0[2];
        UCENP[iELEM*NDOFN+1]=GDISP[1]+c0[1]-xp*c0[2];
        SCENP[iELEM*NSTRE+0]=GSTRE[0];
        SCENP[iELEM*NSTRE+1]=GSTRE[1];
        SCENP[iELEM*NSTRE+2]=GSTRE[2];
        // Coordinates of commputing point
        CECOD[iELEM*NDIME+0]=CenCoord[0]+xp;
        CECOD[iELEM*NDIME+1]=CenCoord[1]+yp;
    }
    free(ECOOD); free(CenCoord); free(EHMTX);
    free(EGMTX); free(d_Ele); free(c_Ele);
    free(N_SET); free(T_SET); free(GDISP);
    free(GSTRE);
}


/*
*******************************************************
* Subroutine RIGIDRV                                  *
* - Recovery of rigid body motion                     *
*******************************************************
*/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
void RIGIDRV(double ECOOD[],double c_Ele[],
             double d_Ele[],int kMATS,int kTSEL,
             double CLENG,double PROPS[],double rvect[])
{
    void DUCLEAN();
    void MATMULT();
    void TREFFTZ();
    void SCHTREF();
    void LSSOLVR();
    extern int NTREF,NDIME,NDOFN,NNODE,NSTRE;
    double x1,x2,*N_SET,*T_SET,*RMATX,du1,du2,*u;
    int iNODE;

    RMATX=(double *)calloc(3*3,sizeof(double));
    DUCLEAN(3,3,RMATX);
    N_SET=(double *)calloc(NDOFN*NTREF,sizeof(double));
    T_SET=(double *)calloc(NSTRE*NTREF,sizeof(double));
    for(iNODE=0;iNODE<NNODE;iNODE++)
```

```
    {
        x1=ECOOD[iNODE*NDIME+0];
        x2=ECOOD[iNODE*NDIME+1];
        DUCLEAN(NDOFN,NTREF,N_SET);
        DUCLEAN(NSTRE,NTREF,T_SET);
        if(kTSEL==0) // general element
        {
            TREFFTZ(x1,x2,kMATS,PROPS,N_SET,T_SET);
        }
        else if(kTSEL==1) // special element
        {
            SCHTREF(x1,x2,CLENG,kMATS,PROPS,
                N_SET,T_SET);
        }
        u=(double *)calloc(NDOFN*1,sizeof(double));
        DUCLEAN(NDOFN,1,u);
        MATMULT(N_SET,c_Ele,NDOFN,NTREF,1,u);

        du1=d_Ele[iNODE*2  ]-u[0];
        du2=d_Ele[iNODE*2+1]-u[1];
        rvect[0]=rvect[0]+du1;
        rvect[1]=rvect[1]+du2;
        rvect[2]=rvect[2]+x2*du1-x1*du2;

        RMATX[0*3+2]=RMATX[0*3+2]+x2;
        RMATX[1*3+2]=RMATX[1*3+2]-x1;
        RMATX[2*3+2]=RMATX[2*3+2]+(pow(x1,2)+pow(x2,2));
    }
    RMATX[2*3+0]=RMATX[0*3+2];
    RMATX[2*3+1]=RMATX[1*3+2];
    RMATX[0*3+0]=NNODE;
    RMATX[1*3+1]=NNODE;
    // R*c0=r
    LSSOLVR(RMATX,rvect,3);
    // after this the rvect stores c0
    free(N_SET); free(T_SET), free(RMATX);
    return;
}
```

8.8 Test examples

To illustrate the application of the special element model described in Sections 8.3 and 8.4, two examples are presented, a Laplace problem and a plane elastic problem.

8.8.1 Potential problems

Consider a Laplace problem with a square domain of size 3×3. The domain contains a circular hole of diameter $2b$ at the centre of the square. The corresponding boundary conditions are listed in Figure 8.4.

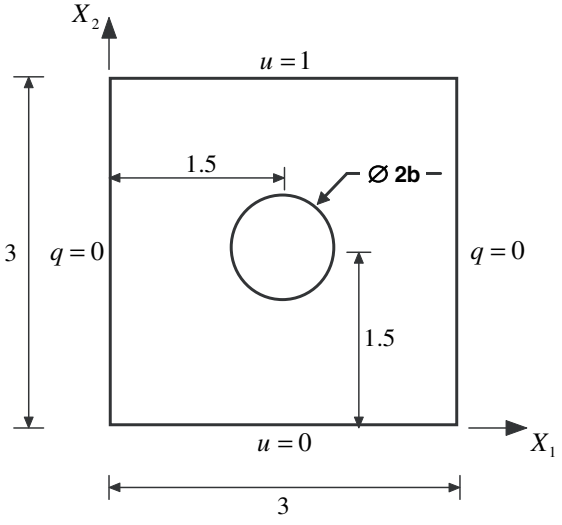


FIGURE 8.4
Square plate with a circular hole

In our analysis, the entire domain is discretised by nine 8-node elements with nonlinear frame functions, in which the element numbered 5 is a special circular hole element (see Figure 8.5). The number of truncated terms of T-complete functions is chosen as 8 to guarantee the rank requirement (5.46). The results obtained are compared with those from ABAQUS obtained using the mesh shown in Figure 8.6 (an 8-node isotropic quadrilateral element is used in the ABAQUS calculation). Because of the symmetry of the problem under consideration, only the numerical results along $X_1 = 1.5$ are considered and listed in Figure 8.7, from which it can be seen that the HT-FEM results are in good agreement with those obtained from ABAQUS, but a relatively coarse mesh is employed in HT-FEM.

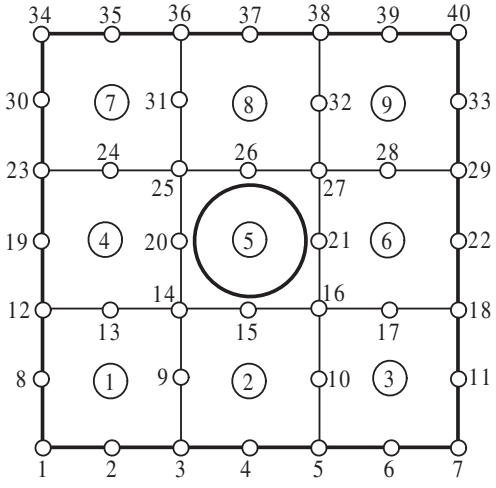


FIGURE 8.5
Configuration of HT-FEM mesh including special circular hole element

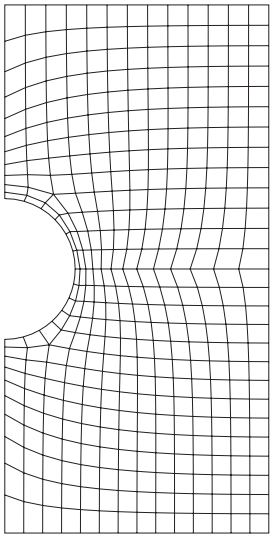


FIGURE 8.6
Configuration of ABAQUS mesh with 8-node isotropic quadrilateral element

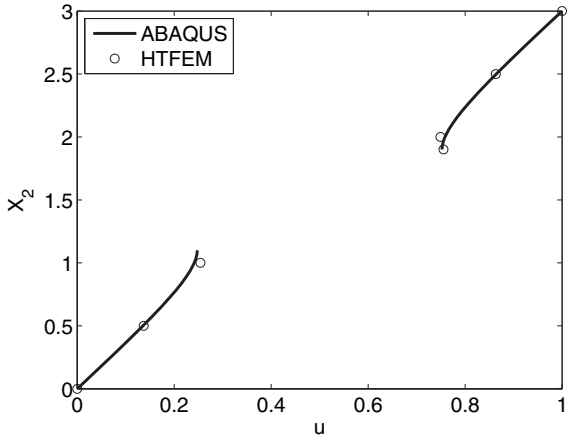


FIGURE 8.7
Comparison of numerical results from ABAQUS and HT-FEM along $X_1 = 1.5$ with $b = 0.4$

To investigate the effect of the radius of the circular hole, numerical results for various radii b (ranging from 0.1 to 0.4) under the same element mesh (see Figure 8.5) are evaluated and the results at point (1.5, 1.5+b) are listed in Table 8.2 and compared to those from ABAQUS. It is evident from Table 8.2 that good agreement is again achieved for the results from the two methods mentioned above. In addition, the property of matrix **H** corresponding to element 5 is investigated using the concept of matrix condition number. Figure 8.8 indicates that the condition number increases along with a decrease in the size of the circular hole.

TABLE 8.2
Comparison of potential u from ABAQUS and HT-FEM at point A with various radii b

Radius b	HT-FEM	ABAQUS
0.4	0.7558	0.7523
0.3	0.6950	0.6937
0.2	0.6316	0.6313
0.1	0.5664	0.5661

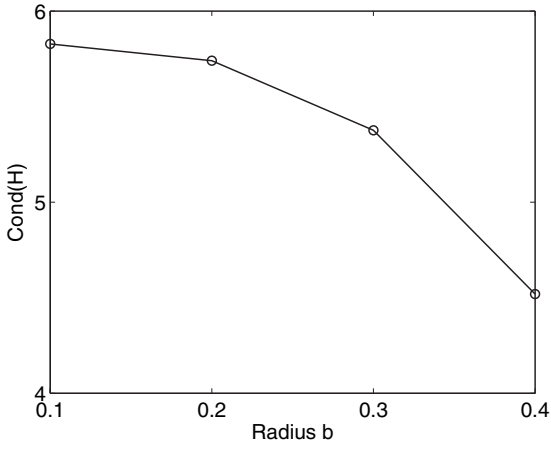


FIGURE 8.8
Variation of condition number of **H** matrix with various radii b

8.8.2 Plane elastic problems

In this example, a square plate with a circular hole subjected to a uniform tension of magnitude $\bar{p}$ in the X_1- direction (which is taken from Ref. [7]) is considered using a special purpose hole element. The geometry and loading of this problem are illustrated in Figure 8.9.

For a small circular hole located in the centre of the plate, let $\theta = \arctan(X_2/X_1)$ be the angle from the X_1-axis and $r = \sqrt{X_1^2 + X_2^2} \geq b$ the radius from the centre point. The theoretical displacement and stress distributions in the neighborhood of the hole are given as [8]

$$
\begin{aligned}
u_1 &= \frac{\bar{p}b(1-2\nu)}{8E}\left[\frac{r}{b}(1+\kappa)\cos\theta + 2\frac{b}{r}(1+\kappa)\cos\theta + 2\frac{b}{r}\cos 3\theta - 2\frac{b^3}{r^3}\cos 3\theta\right] \\
u_2 &= \frac{\bar{p}b(1-2\nu)}{8E}\left[\frac{r}{b}(\kappa-3)\sin\theta + 2\frac{b}{r}(1-\kappa)\sin\theta + 2\frac{b}{r}\sin 3\theta - 2\frac{b^3}{r^3}\sin 3\theta\right]
\end{aligned}
\tag{8.38}
$$

and

$$
\begin{aligned}
\sigma_{11} &= \bar{p}\left[1 - \frac{b^2}{r^2}\left(\frac{3}{2}\cos 2\theta + \cos 4\theta\right) + \frac{3}{2}\frac{b^4}{r^4}\cos 4\theta\right] \\
\sigma_{22} &= \bar{p}\left[-\frac{b^2}{r^2}\left(\frac{1}{2}\cos 2\theta - \cos 4\theta\right) - \frac{3}{2}\frac{b^4}{r^4}\cos 4\theta\right] \\
\sigma_{12} &= \bar{p}\left[-\frac{b^2}{r^2}\left(\frac{1}{2}\sin 2\theta + \sin 4\theta\right) + \frac{3}{2}\frac{b^4}{r^4}\sin 4\theta\right]
\end{aligned}
\tag{8.39}
$$

where E is the elastic modulus, ν is Poisson's ratio, and $\kappa = 3 - 4\nu$ for plane strain and $\kappa = (3-\nu)/(1+\nu)$ for plane stress. Note that the displacements depend on

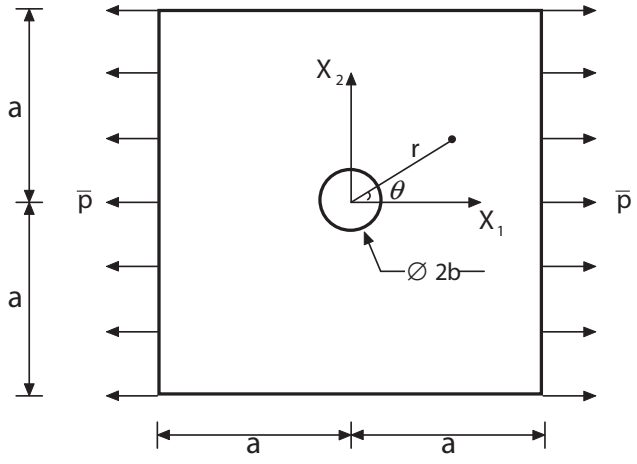


FIGURE 8.9
Square plate with a circular hole subjected to uniform tension

the material properties, but the stress components are dependent on the radius of the hole only. Making use of the coordinate transformation presented in Appendix D, the stress components in polar coordinates can be expressed as

$$\begin{aligned}
\sigma_r &= \frac{\bar{p}}{2}\left(1-\frac{b^2}{r^2}\right)+\frac{\bar{p}}{2}\left(1+\frac{3b^4}{r^4}-\frac{4b^2}{r^2}\right)\cos 2\theta \\
\sigma_\theta &= \frac{\bar{p}}{2}\left(1+\frac{b^2}{r^2}\right)-\frac{\bar{p}}{2}\left(1+\frac{3b^4}{r^4}\right)\cos 2\theta \\
\sigma_{r\theta} &= -\frac{\bar{p}}{2}\left(1-\frac{3b^4}{r^4}+\frac{2b^2}{r^2}\right)\sin 2\theta
\end{aligned} \tag{8.40}$$

It can be seen from Eq. (8.40) that both the radial stress σ_r and shear stress $\tau_{r\theta}$ are zero along the hole surface, which are the so-called free surface conditions (8.28), while the hoop stress is a function of θ: $\sigma_\theta = \bar{p}(1-2\cos 2\theta)$. Thus it has a maximum value of $3\bar{p}$ at $\theta = \pm\pi/2$ and a minimum value of $-\bar{p}$ at $\theta = 0,\ \pi$.

To simulate the stress distributions more accurately and lessen the impact of the boundary effect, $a = 3$ and the radius b varying in the range $[0.4,\ 1]$ are assumed. For the case of $b = 1$, only 25 HT-FEM elements with one special circular hole element are employed to discretise the entire square plate domain (see Figure 8.10), while in the ABAQUS calculation, a quarter of the solution domain is analysed with conventional 8-node isotropic quadrilateral elements due to symmetry (see Figure 8.11). It is evident from Figure 8.12 that the stress results from HT-FEM are in good agreement with those from ABAQUS, although fewer elements are used in HT-FEM. In addition, analytical results are also shown in Figure 8.12 for comparison. It is also evident from Figure 8.12 that when $a/b \leq 3$ the results from both HT-FEM and ABAQUS show an obvious discrepancy from the analytical solution. This discrep-

ancy is attributed to the Saint Venant effect. To determine the value of a/b at which the discrepency between the stress results from HT-FEM (or ABAQUS) and the analytical solution is acceptable (say 5%), the variation of stress component σ_{11} at point $(0, b)$ with circular radius b is graphed in Figure 8.13 when $a = 3$ and the element shown in Figure 8.10 is used. As b decreases, the stress σ_{11} converges to the analytical value 30 when b decreases to zero. It is found from Figure 8.13 that the discrepancy between the stress results from HT-FEM and the analytical solution is less than 5% when $b = 0.4$. Meanwhile, the condition number of matrix $\mathbf{H}$ for the circular hole element is also depicted in Figure 8.14 with various values of b, where it can be seen that the condition number of matrix $\mathbf{H}$ increases with a decrease in the value of b. This is particularly true when $b \leq 0.8$, and this phenomenon may increase the difficulty of evaluating the inverse of matrix $\mathbf{H}$. To bypass this problem, it is suggested to use dimensionless coordinates as detailed in Chapter 9.

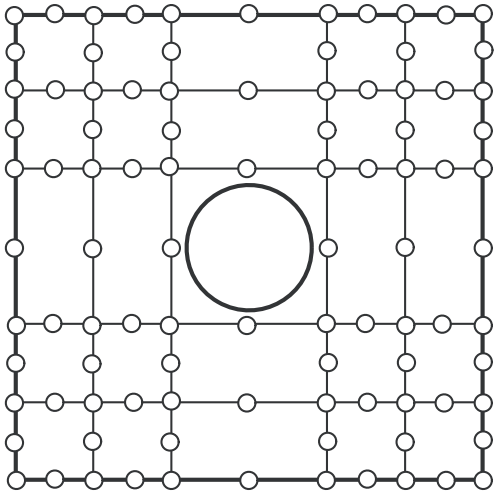


FIGURE 8.10
Configuration of HT-FEM mesh division

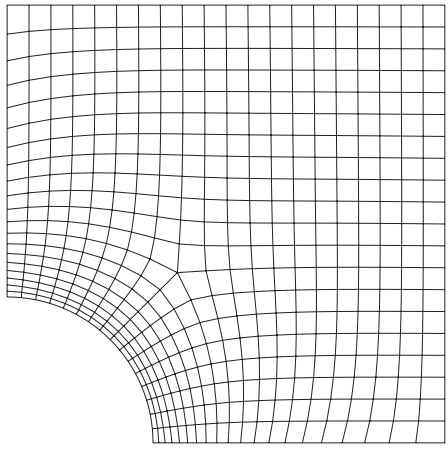


FIGURE 8.11
Configuration of ABAQUS mesh with 8-node isotropic element

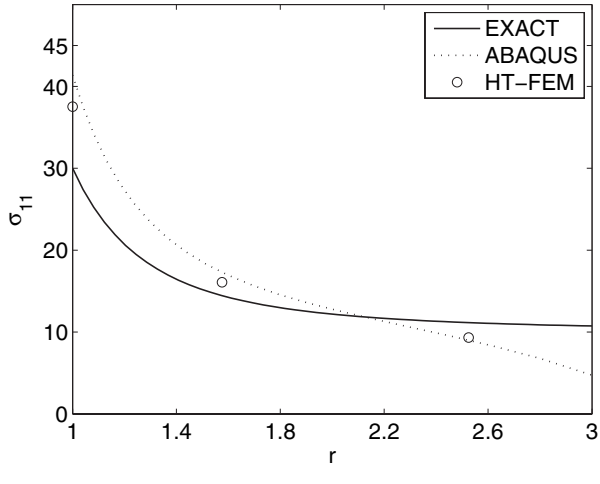


FIGURE 8.12
Stress distribution of σ_{11} along the X_2-axis

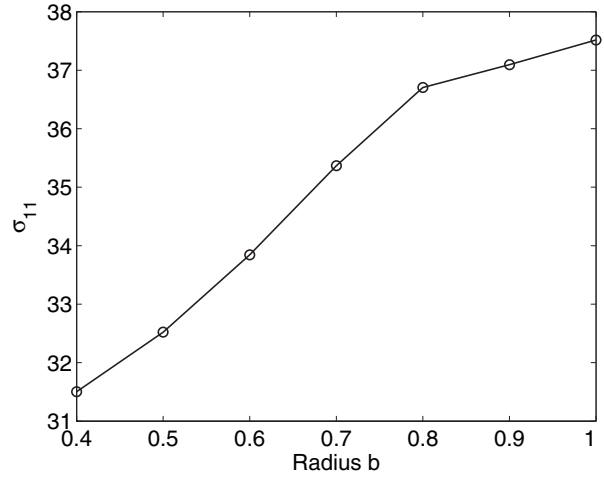


FIGURE 8.13
Variation of stress component σ_{11} at point $(0, b)$ with circular radius b

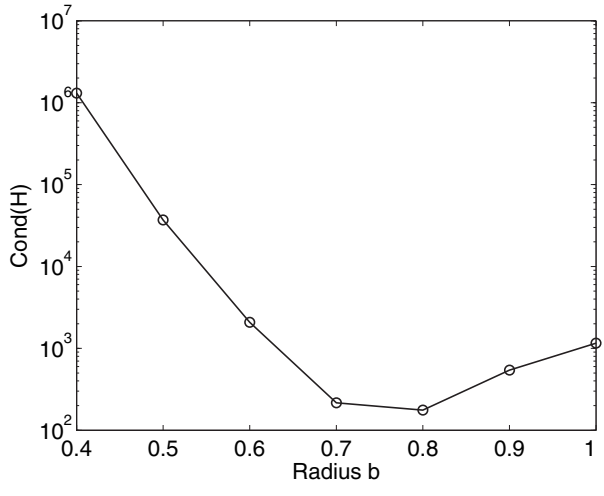


FIGURE 8.14
Variation of condition number of matrix **H** induced from the hole element with circular radius b

References

[1] Piltner R (1985), Special finite elements with holes and internal cracks. *Int J Numer Meth Eng*, **21**: 1471-1485.

[2] Jirousek J, Venkatesh A (1992), Hybrid trefftz plane elasticity elements with p-method capabilities. *Int J Numer Meth Eng*, **35**: 1443-1472.

[3] Venkatesh A, Jirousek J (1995), Accurate representation of local effects due to concentrated and discontinuous loads in hybrid-Trefftz plate bending elements. *Comput Struct*, **57**: 863-870.

[4] Dhanasekar M, Han J, Qin Q (2006), A hybrid-Trefftz element containing an elliptic hole. *Finite Elem Anal Des*, **42**: 1314-1323.

[5] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*. Southampton: WIT Press.

[6] Jirousek J, Guex L (1986), The hybrid-Trefftz finite element model and its application to plate bending. *Int J Numer Meth Eng*, **23**: 651-693.

[7] Timoshenko SP, Goodier JN (1951), *Theory of Elasticity* (2nd edition). New York: McGraw-Hill.

[8] Akin JE (2005), *Finite Element Analysis with Error Estimators*. Elsevier Butterworth-Heinemann.

9

Advanced topics for further programming development

9.1 Introduction

In the preceding chapters we have attempted to show how HT-FE formulations are formed and programmed for potential analysis and linear elastic applications. In particular, we have discussed the theoretical and practical implementations of HT-FEM for solving potential and linear elastic problems with or without non-homogeneous right-hand sources (or body forces) in two-dimensional space. Further, circular hole elements were developed using special purpose Trefftz functions in Chapter 8. The aim of this chapter is to present some advanced topics with which the computer programming given in the previous chapters can be further improved. These topics include construction of Trefftz elements, dimensionless transformation, stress evaluation and smooth treatment, sparse matrix generation and a new idea for developing hybrid elements using fundamental solutions as intra-element trial functions.

9.2 Construction of Trefftz elements

In this book, only conventional four-node linear quadrilateral elements and eight-node quadratic quadrilateral elements are involved in our numerical computation. It should, however, be mentioned that one of the advantages of T-elements is the possibility of constructing arbitrary shaped elements to fulfill the specific purposes of users. For example, we can easily construct triangular elements, pentagonal elements, and so on, by adjusting the relevant frame fields. On the other hand, an increase in elemental sides will create more element nodes and thus more nodal DOF, which requires more terms of Trefftz functions to satisfy the rank requirements (5.44) and (6.61); thus, a higher order of $\mathbf{H}$ matrix may be formed at element level and its inverse operation becomes difficult. In this case, dimensionless transformation can be introduced to overcome this obstacle. The dimensionless procedure is described in the next section.

Besides the regular and special purpose elements described in the previous chap-

ters, the HT p-element model based on the p-extension concept in conventional FEM [1] can also apply to HT elements. Unlike in the *h*-version of FEM,* the purpose in using the *p*-version FEM is to increase the degree of the interpolation functions while keeping elements and subdomains unchanged. The idea of providing the HT elements with p-method capabilities goes back to 1982 [2], but the first practical implementation for thin plate bending was reported only in 1987 [3]. Since that time, applications of HT *p*-element methods have extended to orthotropic plate bending [4], thick plate bending [5], plane elasticity [6], and so on. The results obtained have been so convincing that this kind of new HT *p*-element has been widely used for engineering analysis, due to its high accuracy in modelling field distribution and ease of use [7, 8]. For illustration, we consider a typical triangular *p*-version element in potential problems. It is constructed by adding an optional number of hierarchical side mode DOF, say M, associated with mid-side nodes (see Figure 9.1) to the frame field of a regular element, to achieve higher-order variations and the desired level of precision. It should be noted that the p-element model differs from the standard one described earlier only in the definition of the frame field of nodes. For example, consider the side 1-C-2 of a particular element shown in Figure 9.1. The frame function is now defined in the form [6, 9]

$$\tilde{u}_{12} = \tilde{N}_1 u_1 + \tilde{N}_2 u_2 + \sum_{i=1}^{M} \xi^{i-1} \left(1 - \xi^2\right) u_{Ci} \tag{9.1}$$

In contrast, the conventional interpolation expression is

$$\tilde{u}_{12} = \tilde{N}_1 u_1 + \tilde{N}_2 u_2 \tag{9.2}$$

which produces linear variation along the side 1-C-2. $\tilde{N}_1$ and $\tilde{N}_2$ are linear shape functions defined in Figure 9.1, ξ represents the non-dimensional variable, and u_{Ci} denotes the hierarchic DOF associated with the mid-side node C.

9.3 Dimensionless transformation

From the discussion in the previous chapters, we find that homogeneous Trefftz solutions are usually related to the different order quantities of the distance variable r. The order of r increases along with an increase in the term number m of Trefftz functions, which may affect the properties of the element flexibility matrix $\mathbf{H}_e$ defined as

$$\mathbf{H}_e = \int_{\Gamma_e} \mathbf{Q}_e^{\mathrm{T}} \mathbf{N}_e \mathrm{d}\Gamma \tag{9.3}$$

*Because the maximum dimension of finite elements is usually denoted by h, researchers refer to this conventional mesh refinement as the *h*-version of FEM.

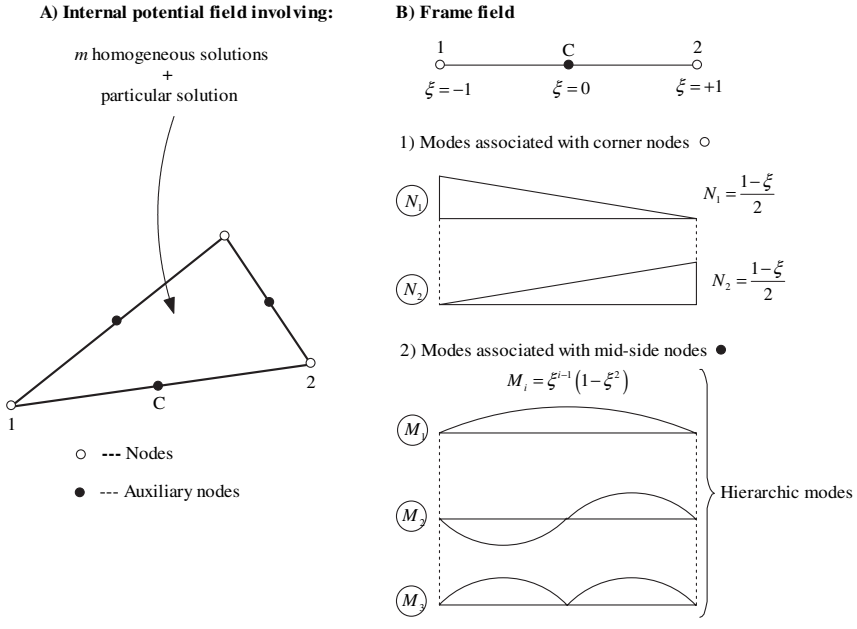


FIGURE 9.1
Triangular *p*-version element

and increase the numerical difficulty of the inverse computation of $\mathbf{H}_e$. Additionally, the geometrical size of the HT element also influences the numerical accuracy of the value of $\mathbf{H}_e$. For a large HT element in terms of geometry, the large distance between the centre and the element boundary may induce greater quantitative difference among the components of $\mathbf{N}_e$ due to the different orders of r. This phenomenon makes it difficult to evaluate the inverse of the matrix $\mathbf{H}_e$. Moreover, for special circular hole elements, the radius of the hole is another important factor affecting numerical accuracy. Too small a size of circular hole also causes numerical instability involving the computation of $\mathbf{H}_e$ and its inverse.

To ensure good numerical conditioning of the matrix $\mathbf{H}_e$ and to prevent overflow or underflow in evaluating the inverse of $\mathbf{H}_e$, the introduction of a local non-dimensional coordinates system is usually suggested [6, 9].

9.3.1 Dimensionless transformation in regular HT element for plane potential problems

To show the generalisation of the procedure of dimensionless transformation, a typical T-element as shown in Figure 9.2 is considered. In Figure 9.2, (X_1, X_2) and (x_1, x_2)represent the global coordinates system and the local coordinates system whose origin is at the centre of the element.

The dimensionless transformation can be achieved by introducing a local coordi-

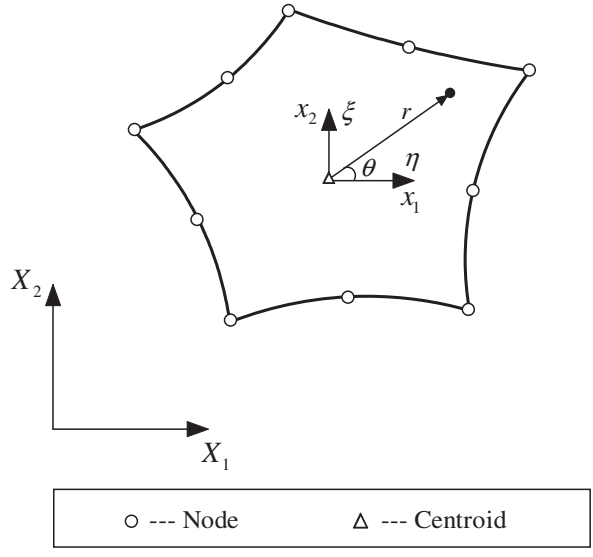


FIGURE 9.2
Dimensionless transformation in regular HT element

nates system $(x_1,\ x_2)$ centred at the centroid $(X_{1c},\ X_{2c})$ of the element. The relationship between $(X_1,\ X_2)$ and $(x_1,\ x_2)$ is defined as

$$\begin{cases} x_1 = X_1 - X_{1c} \\ x_2 = X_2 - X_{2c} \end{cases} \tag{9.4}$$

where the central coordinates $(X_{1c},\ X_{2c})$ measured in the global coordinates system $(X_1,\ X_2)$ can be determined by

$$X_{1c} = \frac{1}{n}\sum_{i=1}^{n} X_{1i}, \qquad X_{2c} = \frac{1}{n}\sum_{i=1}^{n} X_{2i} \tag{9.5}$$

in which n is the number of nodes of the element under consideration. Eq. (9.5) was used in previous chapters to determine the origin of the local coordinate system to an element.

Then, a dimensionless coordinate system $(\xi,\ \eta)$ is employed:

$$\xi = \frac{x_1}{a_e}, \qquad \eta = \frac{x_2}{a_e} \tag{9.6}$$

where a_e is usually taken as the element average distance between the element centroid and its nodes measured in the local coordinates system $(x_1,\ x_2)$

$$a_e = \frac{\sum_{i=1}^{n} \sqrt{x_{1i}^2 + x_{2i}^2}}{n} \tag{9.7}$$

which is used to guarantee the distance between the element boundary points and the centroid to be close to 1.

It should be noted that the previously defined matrices and vectors such as $\mathbf{H}_e$, $\mathbf{G}_e$ and $\mathbf{K}_e$ associated with the element are constructed in the local Cartesian coordinate system $(x_1,\ x_2)$. The process of dimensionless transformation from $(x_1,\ x_2)$ to $(\xi,\ \eta)$ will cause some changes in the expressions of these matrices. For illustration, consider the Trefftz functions used in Chapter 5:

$$N_{2k-1} = r^k \cos(k\theta), \qquad N_{2k} = r^k \sin(k\theta) \qquad (k = 1,2,3,\ldots) \tag{9.8}$$

where

$$r = \sqrt{x_1^2 + x_2^2} \quad \text{and} \quad \theta = \arctan\frac{x_2}{x_1} \tag{9.9}$$

When the dimensionless coordinates $(\xi,\ \eta)$ are employed, we have

$$r|_{(x_1,x_2)} = a_e\, \tilde{r}|_{(\xi,\eta)}, \qquad \theta|_{(x_1,x_2)} = \tilde{\theta}|_{(\xi,\eta)} \tag{9.10}$$

where

$$\tilde{r} = \sqrt{\xi^2 + \eta^2} \quad \text{and} \quad \tilde{\theta} = \arctan\frac{\eta}{\xi} \tag{9.11}$$

Substituting Eq. (9.10) into Eq. (9.8) leads to the following transformation relation on Trefftz functions:

$$\begin{aligned} N_{2k-1}|_{(x_1,x_2)} &= a_e^k\, N_{2k-1}|_{(\xi,\eta)} \\ N_{2k}|_{(x_1,x_2)} &= a_e^k\, N_{2k}|_{(\xi,\eta)} \end{aligned} \tag{9.12}$$

As a result, the interpolation vector $\mathbf{N}_e(\mathbf{x})$ defined in Eq. (5.8) becomes

$$\mathbf{N}_e|_{(x_1,x_2)} = \mathbf{N}_e|_{(\xi,\eta)}\,\mathbf{a} \tag{9.13}$$

where the diagonal matrix $\mathbf{a}$ denotes the dimensionless transformation matrix:

$$\mathbf{a} = \begin{bmatrix} a_e & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_e & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_e^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_e^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a_e^k & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_e^k \end{bmatrix}_{m\times m} \tag{9.14}$$

with $m = 2k$.

Furthermore, $\mathbf{T}_e$ is modified as

$$\mathbf{T}_e|_{(x_1,x_2)} = \left[\begin{array}{c} \dfrac{\partial \mathbf{N}_e}{\partial \hat{x}_1} \\ \dfrac{\partial \mathbf{N}_e}{\partial \hat{x}_2} \end{array}\right]\Bigg|_{(x_1,x_2)} = \left[\begin{array}{c} \dfrac{\partial \mathbf{N}_e(\xi,\eta)\,\mathbf{a}}{\partial \xi}\dfrac{\partial \xi}{\partial \hat{x}_1} \\ \dfrac{\partial \mathbf{N}_e(\xi,\eta)\,\mathbf{a}}{\partial \eta}\dfrac{\partial \eta}{\partial \hat{x}_2} \end{array}\right] = \frac{1}{a_e}\,\mathbf{T}_e|_{(\xi,\eta)}\,\mathbf{a} \tag{9.15}$$

Similarly, we have

$$\mathbf{Q}_e|_{(x_1,x_2)} = \frac{1}{a_e}\,\mathbf{Q}_e|_{(\xi,\eta)}\,\mathbf{a} \tag{9.16}$$

Finally, the matrices $\mathbf{H}_e$ and $\mathbf{G}_e$ are written as[†]

$$\begin{aligned}\mathbf{H}_e|_{(x_1,x_2)} &= \left(\int_{\Gamma_e}\mathbf{Q}_e^{\mathrm{T}}\mathbf{N}_e\mathrm{d}\Gamma\right)\Big|_{(x_1,x_2)}\\ &= \int_{\Gamma_e}\frac{1}{a_e}\mathbf{a}^{\mathrm{T}}\,\mathbf{Q}_e^{\mathrm{T}}\big|_{(\xi,\eta)}\,\mathbf{N}_e|_{(\xi,\eta)}\,\mathbf{a}a_e\,\mathrm{d}\Gamma|_{(\xi,\eta)}\\ &= \mathbf{a}\,\mathbf{H}_e|_{(\xi,\eta)}\,\mathbf{a}\end{aligned} \tag{9.17}$$

$$\begin{aligned}\mathbf{G}_e|_{(x_1,x_2)} &= \int_{\Gamma_e}\mathbf{Q}_e^{\mathrm{T}}\tilde{\mathbf{N}}_e\mathrm{d}\Gamma\Big|_{(x_1,x_2)}\\ &= \int_{\Gamma_e}\frac{1}{a_e}\mathbf{a}^{\mathrm{T}}\,\mathbf{Q}_e^{\mathrm{T}}\big|_{(\xi,\eta)}\,\tilde{\mathbf{N}}_e a_e\,\mathrm{d}\Gamma|_{(\xi,\eta)}\\ &= \mathbf{a}\,\mathbf{G}_e|_{(\xi,\eta)}\end{aligned} \tag{9.18}$$

and consequently the inverse of the matrix $\mathbf{H}_e$ can be derived

$$\mathbf{H}_e^{-1}\big|_{(x_1,x_2)} = \left[\mathbf{a}\,\mathbf{H}_e|_{(\xi,\eta)}\,\mathbf{a}\right]^{-1} = \mathbf{a}^{-1}\,\mathbf{H}_e^{-1}\big|_{(\xi,\eta)}\,\mathbf{a}^{-1} \tag{9.19}$$

with

$$\mathbf{a}^{-1} = \begin{bmatrix} a_e^{-1} & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & a_e^{-1} & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & a_e^{-2} & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & a_e^{-2} & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & \ddots & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & a_e^{-k} & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & a_e^{-k} \end{bmatrix}_{m\times m} \tag{9.20}$$

Substitution of the matrices $\mathbf{H}_e$ and $\mathbf{G}_e$ into the stiffness matrix $\mathbf{K}_e$ produces

$$\mathbf{K}_e|_{(x_1,x_2)} = \left(\mathbf{G}_e^{\mathrm{T}}\mathbf{H}_e^{-1}\mathbf{G}_e\right)\big|_{(x_1,x_2)} = \left(\mathbf{G}_e^{\mathrm{T}}\mathbf{H}_e^{-1}\mathbf{G}_e\right)\big|_{(\xi,\eta)} = \mathbf{K}_e|_{(\xi,\eta)} \tag{9.21}$$

from which we find that the stiffness matrix does not change in value after dimensionless transformation.

In addition, the coefficient vector appearing in the intra-element potential field can be changed as

$$\mathbf{c}_e|_{(x_1,x_2)} = \left(\mathbf{H}_e^{-1}\mathbf{G}_e\right)\big|_{(x_1,x_2)}\,\mathbf{d}_e = \mathbf{a}^{-1}\,\left(\mathbf{H}_e^{-1}\mathbf{G}_e\right)\big|_{(\xi,\eta)}\,\mathbf{d}_e \tag{9.22}$$

[†]Note: the relation

$$\mathrm{d}\Gamma|_{(x_1,x_2)} = \sqrt{\mathrm{d}x_1^2+\mathrm{d}x_2^2} = a_e\sqrt{\mathrm{d}\xi^2+\mathrm{d}\eta^2} = a_e\,\mathrm{d}\Gamma|_{(\xi,\eta)}$$

is used for the transformation of matrices $\mathbf{H}_e$ and $\mathbf{G}_e$.

9.3.2 Dimensionless transformation in special HT element for plane potential problems

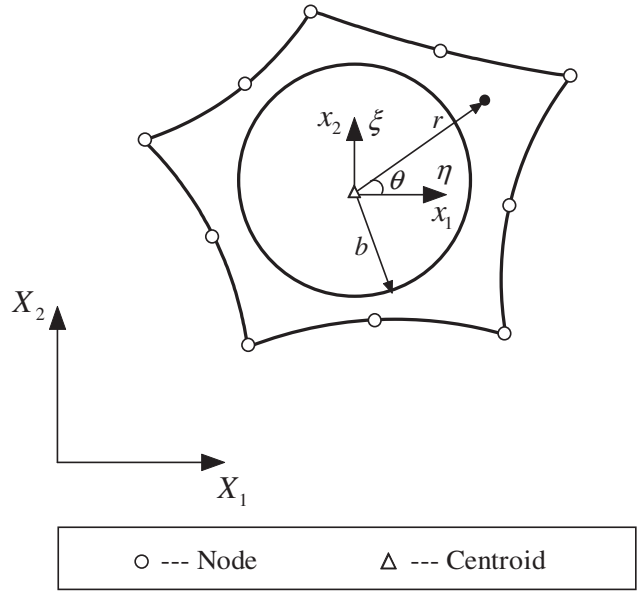


FIGURE 9.3
Dimensionless transformation in a circular hole element

For a special T-element including a circular hole with radius b as displayed in Figure 9.3, dimensionless transformation of Eq. (9.6) leads to

$$r|_{(x_1,x_2)} = a_e\, \tilde{r}|_{(\xi,\eta)}\,, \qquad \theta|_{(x_1,x_2)} = \tilde{\theta}|_{(\xi,\eta)}\,, \qquad b|_{(x_1,x_2)} = a_e\, \tilde{b}|_{(\xi,\eta)} \tag{9.23}$$

Substituting Eq. (9.23) into the potential Trefftz functions defined in Eq. (8.13) leads to the same form as that of Eq. (9.12), but different content. The corresponding matrices $\mathbf{T}_e$, $\mathbf{H}_e$ and $\mathbf{G}_e$ also have the same forms as those of Eqs. (9.12) - (9.19).

9.3.3 Dimensionless transformation in regular element for plane elastic problems

Consider again the regular HT element shown in Figure 9.2. Introduction of the dimensionless transformation (9.6) to the homogeneous displacement solutions in Eqs. (6.52) - (6.55) and stress solutions in Eqs. (6.57) - (6.60) yields

$$\begin{aligned} \mathbf{N}^*_{ik}|_{(x_1,x_2)} &= a_e^k\, \mathbf{N}^*_{ik}|_{(\xi,\eta)} \\ \mathbf{T}^*_{ik}|_{(x_1,x_2)} &= a_e^{k-1}\, \mathbf{T}^*_{ik}|_{(\xi,\eta)} \end{aligned} \tag{9.24}$$

where $i = 1,2,3,4$, $k = 1,2,3,\cdots$. In the above derivation, the following relations

$$z|_{(x_1,x_2)} = a_e\, z|_{(\xi,\eta)}\,, \qquad \bar{z}|_{(x_1,x_2)} = a_e\, \bar{z}|_{(\xi,\eta)} \tag{9.25}$$

have been used. Using the relations (9.24), the interpolation matrix defined in Eq. (6.12) yields the same form as that of Eq. (9.13) except that the matrix **a** is now defined by

$$\mathbf{a} = \begin{bmatrix} a_e & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_e & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_e & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_e^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_e^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a_e^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_e^2 \end{bmatrix} \tag{9.26}$$

for the case of $m = 7$.

The corresponding $\mathbf{H}_e$, $\mathbf{G}_e$, $\mathbf{K}_e$ and $\mathbf{c}_e$ in terms of the dimensionless coordinate system $(\xi,\ \eta)$ can be obtained in a way similar to that described in Section 9.3.1. It is found that these matrices have the same form as those of Eqs. (9.17), (9.18), (9.21) and (9.22).

9.3.4 Dimensionless transformation in hole element for plane elastic problems

Consider again the circular hole element shown in Figure 9.3. When the dimensionless coordinates (ξ,η) defined in Eq. (9.6) are used, the displacement homogeneous solutions $\hat{\mathbf{N}}^*_{ik}$ and stress solutions $\hat{\mathbf{T}}^*_{ik}$ in Section 8.4 become

$$\begin{aligned}
\hat{\mathbf{N}}^*_{10}\big|_{(x_1,x_2)} &= a_e\, \hat{\mathbf{N}}^*_{10}\big|_{(\xi,\eta)} \\
\hat{\mathbf{N}}^*_{11}\big|_{(x_1,x_2)} &= a_e^2\, \hat{\mathbf{N}}^*_{11}\big|_{(\xi,\eta)} \\
\hat{\mathbf{N}}^*_{21}\big|_{(x_1,x_2)} &= a_e^2\, \hat{\mathbf{N}}^*_{11}\big|_{(\xi,\eta)} \\
\hat{\mathbf{N}}^*_{1k}\big|_{(x_1,x_2)} &= a_e^{1+k}\, \hat{\mathbf{N}}^*_{1k}\big|_{(\xi,\eta)} \quad (k \geq 2) \\
\hat{\mathbf{N}}^*_{2k}\big|_{(x_1,x_2)} &= a_e^{1+k}\, \hat{\mathbf{N}}^*_{2k}\big|_{(\xi,\eta)} \quad (k \geq 2) \\
\hat{\mathbf{N}}^*_{3k}\big|_{(x_1,x_2)} &= a_e^{-1-k}\, \hat{\mathbf{N}}^*_{3k}\big|_{(\xi,\eta)} \quad (k \geq 2) \\
\hat{\mathbf{N}}^*_{4k}\big|_{(x_1,x_2)} &= a_e^{-1-k}\, \hat{\mathbf{N}}^*_{4k}\big|_{(\xi,\eta)} \quad (k \geq 2)
\end{aligned} \tag{9.27}$$

and

$$
\begin{aligned}
\hat{\mathbf{T}}^*_{10}\big|_{(x_1,x_2)} &= \hat{\mathbf{T}}^*_{10}\big|_{(\xi,\eta)} \\
\hat{\mathbf{T}}^*_{11}\big|_{(x_1,x_2)} &= a_e\,\hat{\mathbf{T}}^*_{11}\big|_{(\xi,\eta)} \\
\hat{\mathbf{T}}^*_{21}\big|_{(x_1,x_2)} &= a_e\,\hat{\mathbf{T}}^*_{11}\big|_{(\xi,\eta)} \\
\hat{\mathbf{T}}^*_{1k}\big|_{(x_1,x_2)} &= a_e^k\,\hat{\mathbf{T}}^*_{1k}\big|_{(\xi,\eta)} \quad (k \geq 2) \\
\hat{\mathbf{T}}^*_{2k}\big|_{(x_1,x_2)} &= a_e^k\,\hat{\mathbf{T}}^*_{2k}\big|_{(\xi,\eta)} \quad (k \geq 2) \\
\hat{\mathbf{T}}^*_{3k}\big|_{(x_1,x_2)} &= a_e^{-2-k}\,\hat{\mathbf{T}}^*_{3k}\big|_{(\xi,\eta)} \quad (k \geq 2) \\
\hat{\mathbf{T}}^*_{4k}\big|_{(x_1,x_2)} &= a_e^{-2-k}\,\hat{\mathbf{T}}^*_{4k}\big|_{(\xi,\eta)} \quad (k \geq 2)
\end{aligned}
\tag{9.28}
$$

Making use of the dimensionless coordinate system (ξ,η), the interpolation matrix defined in Eq. (6.12) yields the same form as that of Eq. (9.13) except that the matrix **a** is now defined by

$$
\mathbf{a} = \begin{bmatrix}
a_e & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & a_e^2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & a_e^2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & a_e^3 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & a_e^3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & a_e^{-3} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & a_e^{-3}
\end{bmatrix}
\tag{9.29}
$$

for the case of $m = 7$.

The corresponding $\mathbf{H}_e$, $\mathbf{G}_e$, $\mathbf{K}_e$ and $\mathbf{c}_e$ for the case of hole element can again be obtained similarly and it is found that these matrices also have the same form as those of Eqs. (9.17), (9.18), (9.21) and (9.22).

9.4 Nodal stress evaluation-smooth techniques

Like conventional FEM, HT-FE analysis generally involves the minimisation of some functional defined in terms of piecewise functions including internal Trefftz interpolation and element boundary frame functions. These functions are generally required to have a certain degree of inter-element continuity depending on terms in the functional. In many engineering problems, the quantities of primary engineering interest involve the function derivatives and in many instances, these derivatives do not possess continuity between elements. For example, in elastic analysis, continuity of displacement fields obtained by HT-FEM is guaranteed within and between elements. However, the stress components evaluated from displacement derivatives are usually discontinuous between elements (see Figure 9.4); they are continuous within each element only. As a result, histogram-type distributions of these discontinuous

fields are usually encountered in practical computation and the analyst is therefore faced with the problem of interpreting quantities with such distributions. In fact, this discontinuity phenomenon is usually not rational from the theoretical point of view in practical engineering, since in the so-called displacement-based methods, for example, FEM, BEM and HT-FEM, the discontinuous stresses between elements are caused by the different measurements of the assumed displacement variation in each element.

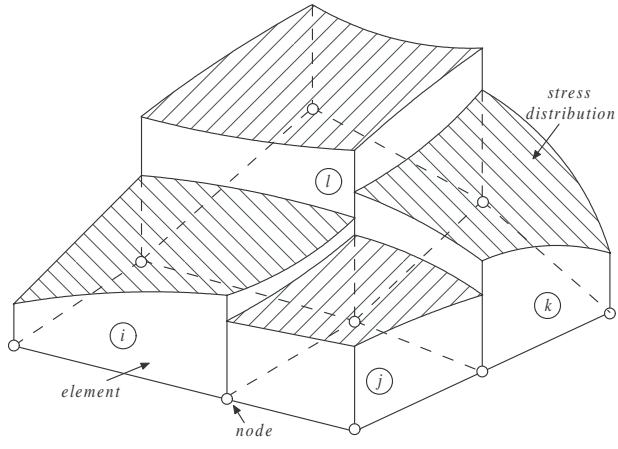


FIGURE 9.4
Illustration of discontinuous stress field between elements in HT-FEM

Although in quite a few commercial FE software programs such as ABAQUS and ANSYS some rational and consistent post-process procedures are adopted for the interpretation of discontinuous functions and can produce good smooth results, it is necessary to review some treatments to obtain smooth distribution of certain quantities at element nodes in the domain under consideration for further improvement of HT-FEM.

Consider again the elastic problems discussed in Chapter 6. Unlike in the conventional FEM, the stress distributions (6.14) within each element

$$\boldsymbol{\sigma}_e = \mathbf{T}_e \mathbf{c}_e \tag{9.30}$$

exactly satisfy the governing equations according to HT-FE theory and the domain integration is removed in the calculation of stiffness matrix in the HT-FEM. The extrapolation treatment employed widely in conventional FEM [10], using the stress values at Gauss sampling points is not suitable for the HT-FEM. Here we introduce a simpler way to obtain nodal values of stress. It is achieved by averaging the values of stress at the corner nodes. For instance, for the four elements shown in Figure 9.4,

the value of stress at the common node p is calculated using the following expression

$$\tilde{\sigma}_p = \frac{\sigma_p^{e_i} + \sigma_p^{e_j} + \sigma_p^{e_k} + \sigma_p^{e_l}}{4} \tag{9.31}$$

or

$$\tilde{\sigma}_p = \frac{\sigma_p^{e_i} A_i + \sigma_p^{e_j} A_j + \sigma_p^{e_k} A_k + \sigma_p^{e_l} A_l}{A_i + A_j + A_k + A_l} \tag{9.32}$$

where $\tilde{\sigma}_p$ represents the smoothed nodal values of stress. $\sigma_p^{e_i}$, $\sigma_p^{e_j}$, $\sigma_p^{e_k}$ and $\sigma_p^{e_l}$ are the unsmoothed stresses evaluated from Eq. (9.30) in different elements, and A_i, A_j, A_k, and A_l denote the areas of elements i, j, k and l, respectively.

9.5 Generating intra-element points for outputting field results

Unlike in the conventional FEM, all integrals involved in the calculation of a stiffness matrix are defined on the element boundary in the HT-FEM. Therefore, the output of field results at Gaussian sample points used in the conventional FEM is not suitable for the HT-FEM. In the previous chapters, only the internal fields at the centroid of an element are calculated and outputted. To obtain field distributions at more intra-element points for further reference, the following expression can be used to generate internal points:

$$x_{m1} = \frac{x_{i1}}{2}, \qquad x_{m2} = \frac{x_{i2}}{2} \tag{9.33}$$

where (x_{m1}, x_{m2}) represents the midpoint coordinate of the line with ends at the centroid and a particular element node (see Figure 9.5a) and (x_{i1}, x_{i2}) are the nodal coordinates of the node.

For the special circular hole element discussed in Chapter 8, let the origin of the local coordinates be the centre of the circular hole. We can generate extra computing points (x_{m1}, x_{m2}) on the circular boundary through the formulation (see Figure 9.5b):

$$x_{m1} = \frac{x_{i1} b}{\sqrt{x_{i1}^2 + x_{i2}^2}}, \qquad x_{m2} = \frac{x_{i2} b}{\sqrt{x_{i1}^2 + x_{i2}^2}} \tag{9.34}$$

where b represents the radius of the circular hole.

9.6 Sparse matrix generation and solving procedure

In Chapters 5 and 6, the final global stiffness equation is solved using a Gaussian elimination algorithm which is based on the full stiffness matrix of the finite element equation. However, after a node-by-node assembling procedure the resulting

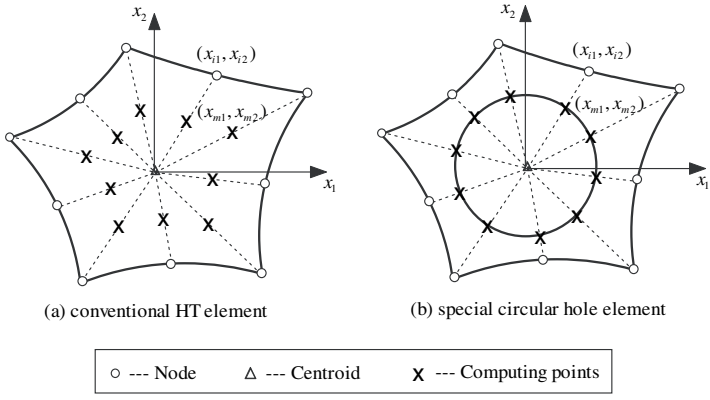


FIGURE 9.5
Generation of more internal points within a regular T-element and a special circular hole element

linear system is characterised by a system matrix which is usually large and sparse. "Sparse" here indicates that many elements in the global stiffness matrix are zero and some elements are located near the main diagonal line. To increase computational efficiency and reduce the space requirement for storing the stiffness matrix, the modified banded storage strategy is generally employed (see Figure 9.6). For the sake of convenience, we present here a brief review of this strategy.

As usual, the bandwidth can be evaluated by [11]

$$\text{bandwidth} = (\max_e D_e + 1) \times DOF \tag{9.35}$$

where D_e is the maximum difference between any node numbers occurring in a specific element in the HT-FEM, and DOF denotes the number of degrees of freedom per node.

From Eq. (9.35), we can see that to reduce bandwidth we should number systematically and try to ensure a minimum number difference between adjacent nodes. The narrow bandwidth means a small storage requirement, especially for large-scale computation.

Subsequently, the corresponding banded-symmetric solver based on modified Gaussian elimination can be designed to obtain the final nodal displacements. The detailed algorithm can be found in Ref. [11].

Alternatively, the wavefront or frontal method may be used to optimise equation solution time. In the wavefront method, elements rather than nodes are automatically renumbered, and assembly of the stiffness equations alternates with their solution by Gaussian elimination. Thus, it is somewhat more difficult to understand and to program than the classic banded-symmetric method, but it has greater computational efficiency than the latter and is therefore becoming popular in large-scale programs [12, 13].

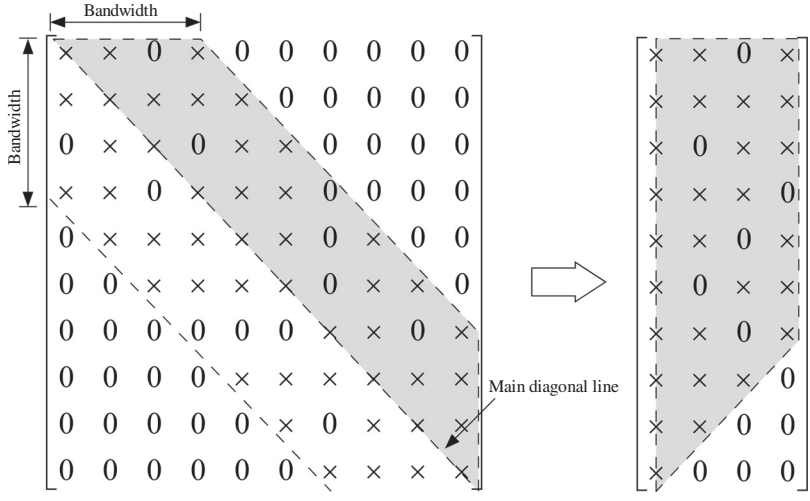


FIGURE 9.6
Sparse and symmetric stiffness matrix and banded storage, where the symbol × denotes nonzero coefficients

9.7 An alternative formulation to HT-FEM

It is well known that HT-FEM is based on a hybrid-type method which includes the use of an independent auxiliary inter-element frame field defined on each element boundary and an independent internal field chosen so as to *a priori* satisfy the homogeneous governing differential equations by means of a suitable truncated T-complete functions set of homogeneous solutions. Inter-element continuity is enforced by using a modified variational principle, which is used to construct the standard force-displacement relationship, that is, the stiffness equation, and to establish linkage of frame field and internal fields of the element. The property of non-singular element boundary integrals in HT-FEM enables us to construct arbitrary shaped elements conveniently. Moreover, special T-element can be designed to perform special-purpose analyses. However, the terms of truncated T-complete functions must be carefully selected to achieve the desired results. Further, T-complete functions are difficult to generate for some physical problems. Moreover, due to the inherent properties of T-complete functions used, that is, higher orders of the Euclidean distance variable, a relatively complex coordinate transformation is usually required in HT-FEM to keep the inverse of matrix **H** stable.

To circumvent this drawback while maintaining the advantages of HT-FEM, a novel hybrid finite formulation based on the fundamental solutions, called HFS-FEM, has been developed by Wang and Qin [14]. In the HFS-FEM, the fundamental solution is used to replace the Trefftz functions in the HT-FEM. The proposed HFS-

FEM can be viewed as a fourth type of FEM, which is significantly different from the other three types including conventional FEM [11, 15], natural FEM [16, 17], and HT-FEM [9]. In the analysis, a linear combination of the fundamental solutions at different source points is used to approximate the field variable within the element. The independent frame field is used to guarantee inter-element continuity and defined in the same way as in HT-FEM along the element boundary. The modified variational principle employed is similar to that in HT-FEM and is used to generate the final stiffness equation and establish linkage between the boundary frame field and internal field in an element. The proposed HFS-FEM inherits all the advantages of the HT-FEM and removes the difficulties encountered in constructing and selecting T-functions.

It can be seen from the discussion above that the major difference between HT-FEM and HFS-FEM lies in the different intra-element trial functions used. T-complete functions are used in HT-FEM, whereas fundamental solutions are used as internal trial functions in HFS-FEM to construct the approximated interpolation field within an element. We take the Laplace equation as an example to demonstrate the basic concept of the proposed HFS-FEM.

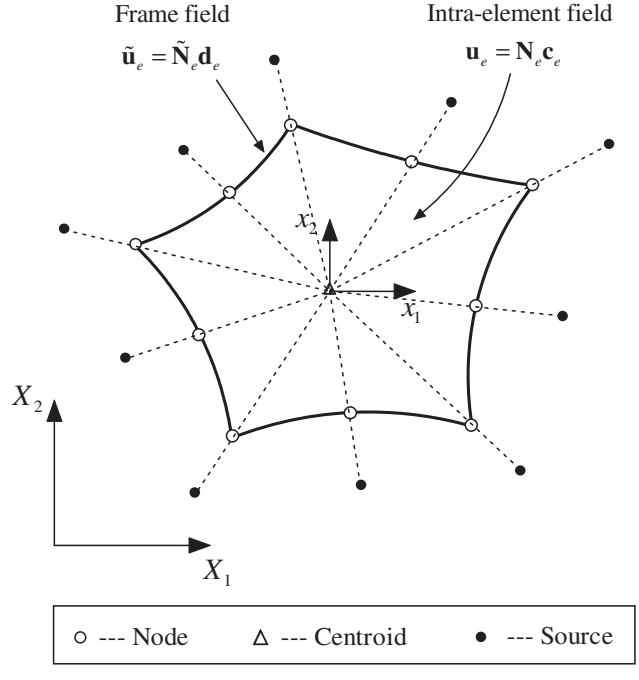


FIGURE 9.7
Intra-element field, frame field in a particular element in HFS-FEM, and the generation of source points

For a typical polygonal element as shown in Figure 9.7, the following two groups of potential fields are assumed:

- The intra-element field is defined within the element

$$u_e(\mathbf{x}) \approx \sum_{j=1}^{n_s} N_e(\mathbf{x},\mathbf{y}_j)\, c_{ej} = \mathbf{N}_e(\mathbf{x})\,\mathbf{c}_e \qquad \forall \mathbf{x} \in \Omega_e,\ \ \mathbf{y}_j \notin \Omega_e \tag{9.36}$$

where $\mathbf{c}_e$ is a vector of undetermined coefficients (or virtual source value) and n_s is the number of virtual sources outside the element under consideration. $N_e^j(\mathbf{x}) = N_e(\mathbf{x},\mathbf{y}_j)$ is the fundamental solution at point $\mathbf{y}_j$ satisfying the Laplace operator equation in an infinite domain:

$$\nabla^2 N_e(\mathbf{x},\mathbf{y}) = -\delta(\mathbf{x},\mathbf{y}) \qquad \forall \mathbf{x},\mathbf{y} \in \mathbb{R}^2 \tag{9.37}$$

where

$$\delta(\mathbf{x},\mathbf{y}) = \begin{cases} 0 & \mathbf{x} \neq \mathbf{y} \\ 1 & \mathbf{x} = \mathbf{y} \end{cases} \tag{9.38}$$

For the two-dimensional case, we have

$$N_e(\mathbf{x},\mathbf{y}) = -\frac{1}{2\pi}\ln r \tag{9.39}$$

Note that coordinates $\mathbf{x}$, $\mathbf{y}_j$ are defined in the local coordinates system (x_1, x_2) and

$$r = \|\mathbf{x} - \mathbf{y}\| \tag{9.40}$$

Substituting Eq. (9.36) into the standard Laplace equation (5.1), we have

$$\nabla^2 u(\mathbf{x}) = \sum_{j=1}^{n_s} \nabla^2 N_e(\mathbf{x},\mathbf{y}_j)\, c_{ej} = 0 \qquad \forall \mathbf{x} \in \Omega_e,\ \ \mathbf{y}_j \notin \Omega_e \tag{9.41}$$

in which the solution property of $N_e(\mathbf{x},\mathbf{y}_j)$ and the point $\mathbf{y}_j$ located outside the element are used.

The virtual source point $\mathbf{y}_j$ can be generated in the same manner as in the method of fundamental solutions (MFS) [18 - 21]

$$\mathbf{y} = \mathbf{x}_b + \gamma(\mathbf{x}_b - \mathbf{x}_c) \tag{9.42}$$

where γ is a dimensionless coefficient, $\mathbf{x}_b$ and $\mathbf{x}_c$ are the boundary point and the geometrical centroid of an element, respectively. For the element shown in Figure 9.7, we can utilise element nodes as $\mathbf{x}_b$ to generate corresponding sources outside the element. It can be proved that generating source points using all element nodes naturally satisfies the rank requirement of minimal terms (5.45).

- Frame field defined along the element boundary

$$\tilde{u}_e(\mathbf{x}) = \tilde{\mathbf{N}}_e(\mathbf{x})\,\mathbf{d}_e \qquad \mathbf{x} \in \Gamma_e \tag{9.43}$$

is designed to enforce conformity on the field variable u between neighbouring elements. $\mathbf{d}_e$ denotes nodal DOF vector of all element nodes and $\tilde{\mathbf{N}}_e$ represents the conventional finite element interpolating functions. This procedure is the same as that used in HT-FEM.

- A modified functional in the same form as Eq. (5.21)

$$\Psi_{me} = \frac{1}{2}\int_{\Omega_e}\left(q_1^2+q_2^2\right)t_e\mathrm{d}\Omega - \int_{\Gamma_e} q\tilde{u}t_e\mathrm{d}\Gamma + \int_{\Gamma_{eq}} \bar{q}\tilde{u}t_e\mathrm{d}\Gamma \tag{9.44}$$

is used to establish the linkage of unknown $\mathbf{c}_e$ and $\mathbf{d}_e$ and to obtain the final stiffness equation.

In summary, in this section we have presented the basic concept and procedure of the newly developed HFS-FEM. Further studies on this area are under way.

References

[1] Babuska I, Szabo BA, Katz IN (1981), The p-version of the finite element method. *SIAM J Numer Anal*, **18**: 515-545.

[2] Jirousek J, Teodorescu P (1982), Large finite elements method for the solution of problems in the theory of elasticity. *Comput Struct*, **15**: 575-587.

[3] Jirousek J (1987), Hybrid-Trefftz plate bending elements with *p*-method capabilities. *Int J Numer Meth Eng*, **24**: 1367-1393.

[4] Jirousek J, N'Diaye M (1990), Solution of orthotropic plates based on *p*-extension of the hybrid-Trefftz finite element model, *Comput Struct*, **34**: 51-62.

[5] Jirousek J, Wroblewski A, Qin QH, *et al*. (1995), A family of quadrilateral hybrid-Trefftz *p*-elements for thick plate analysis. *Comput Method Appl M*, **127**: 315-344.

[6] Jirousek J, Venkatesh A (1992), Hybrid trefftz plane elasticity elements with p-method capabilities. *Int J Numer Meth Eng*, **35**: 1443-1472.

[7] Jirousek J, Wroblewski A (1996), T-elements: state of the art and future trends, *Arch Comput Method E*, **3**: 323-434.

[8] Jirousek J, Szybinski B, Wroblewski A (1996), Mesh design and reliability assurance in hybrid-Trefftz p-element approach. *Finite Elem Anal Des*, **22**: 225-247.

[9] Qin QH (2000), *The Trefftz Finite and Boundary Element Method*. Southampton: WIT Press.

[10] Hinton E, Owen DRJ (1977), *Finite Element Programming*. London: Academic Press.

[11] Chandrupatla TR, Belegundu AD (2002), *Introduction to Finite Elements in Engineering* (3rd edition). New Jersey: Prentice Hall.

[12] Logan DL (2007), *A First Course in the Finite Element Method* (4th edition). Toronto: Thomson.

[13] Bruce MI (1970), A frontal solution program for finite element analysis. *Int J Numer Meth Eng*, **2**: 5-32.

[14] Wang H, Qin QH (2008), Hybrid FEM with fundamental solution as trial function for 2D Laplace problems, *in preparation*.

[15] Zienkiewicz OC, Taylor RL (2000), *The Finite Element Method* (Vol I, II). Oxford: Elsevier Butterworth-Heinemann.

[16] Argyris JH, *et al* (1974), Large natural strains and some special difficulties due to non-linearity and incompressibility in finite elements. *Comput Method Appl M*, **4**: 219-278.

[17] Tenek L, Argyris JH (1997), Computational aspects of the natural-mode finite element method. *Commun Numer Meth En*, **13**: 705-713.

[18] Young DL, Jane SJ, Fan CM, *et al*. (2006), The method of fundamental solutions for 2D and 3D Stokes problems. *J Comput Phys*, **211**: 1-8.

[19] Young DL, *et al* (2006), Method of fundamental solutions for multidimensional Stokes equations by the dual-potential formulation. *European J Mech - B/Fluids*, **25**: 877-893.

[20] Wang H, Qin QH, Kang YL (2005), A new meshless method for steady-state heat conduction problems in anisotropic and inhomogeneous media. *Arch Appl Mech*, **74**: 563-579.

[21] Wang H, Qin QH, Kang YL (2006), A meshless model for transient heat conduction in functionally graded materials. *Comput Mech*, **38**: 51-60.

A

Format of input data

```
**********************************************
Hybrid Trefftz FEM
**********************************************
NTREF NTYPE                                                  ┐
8     0                                                      │
NNODE NEDGE NODEG                                            │
8     4     3                                                │
NDIME NDOFN NSTRE                                            ├ Basic parameters
2     1     2                                                │
NMATS NPROP NGAUS                                            │
1     3     4                                                │
NPOIN NELEM NVFIX NPLOD NDLEG                                │
9     4     6     0     4                                    ┘
-- Element connectivity and material numbers                 ┐
Elem# Mat# Node#1-->#NNODE                                   │ NELEM
1     1    1   2   3   9 ...                                  │ MATNO
2     2    3   4   5  10 ...                                  │ LNODS
.........                                                    ┘
-- Nodal coordinates                                         ┐
Node# Coord#1-->#NDIME                                       │ NPOIN
1     0.000   0.000 ...                                      │ COORD
2     6.667   0.000 ...                                      │
.........                                                    ┘
-- Constrained boundary conditions                           ┐
Num# Node# DOF#1-->#NDOFN Val#1-->#NDOFN                     │ NVFIX  NOFIX
1    10    1    0 ...     3.0   0.0 ...                      │ IFPRE  PRESC
2     8    0    1 ...     0.0  10.0 ...                      │
.........                                                    ┘
-- Concentrated loads at nodes                               ┐
Num# Node# Val#1-->#NDOFN                                    │ NPLOD  LODPT
1    1     10.0   0.0 ...                                     │ POINT
2    7      0.0  -8.0 ...                                     │
.........                                                    ┘
-- Distributed edge loads                                    ┐
Num# Ele# Node#1-->#NODEG Val#1-->#NODEG*NDOFN               │ NDLEG  NEASS
1    2    4   1 ...       0.0   2.0 ...                      │ NOPRS  PRESS
2    9    5   6 ...       1.0   1.0 ...                      │
.........                                                    ┘
-- Read material properties                                  ┐
Mat# Pro#1-->#NPROP                                          │ NMATS  NPROP
1    1.0   0.30   1.0 ...                                    │ PROPS
2    8.0   0.25   2.0 ...                                    │
.........                                                    ┘
```

B

Glossary of variables

`AMTRX(NDOFN,NSTRE)`	Array storing unit normal at Gaussian points to a particular edge
`ASDIS(NEQNS,1)`	Vector of generalized nodal displacements
`CECOD(NELEM,NDIME)`	Vector of coordinates of centroids for all elements
`COORD(NPOIN,NDIME)`	Coordinates of nodal points
`CRBFI(NINTP*NDOFN,1)`	Coefficients of RBF interpolation
`DSHAP(1,NODEG)`	Array of shape function derivatives associated with a particular edge of the element under consideration
`ECOOD(NNODE,NDIME)`	Local array of nodal coordinates for the element under consideration
`EHMTX(NTREF,NTREF)`	Matrix **H** associated with a particular element
`EGMTX(NTREF,NEVAB)`	Matrix **G** associated with a particular element
`ELNOD(NEDGE,NODEG)`	Local array of nodal numbers associated with element edges in each element
`ELOAD(NEVAB,1)`	Element equivalent nodal forces for each element
`ESTIF(NEVAB,NEVAB)`	Element stiffness matrix
`GLOAD(NEQNS,1)`	Array of global equivalent nodal forces
`GSTIF(NEQNS,NEQNS)`	Global stiffness matrix

IFPRE(NVFIX,NDOFN)	Integer code to specify which degrees of freedom at a node are to be restrained or prescribed with specified field values = 1 a fixed degree of freedom = 0 no restraint is imposed on that degree of freedom
IPCOD(NINTP,NDIME)	Coordinates of RBF interpolation points
LNODS(NELEM,NNODE)	Element node numbers listed for each element
LODPT(NPLOD,1)	Number of a node where a concentrated load is applied
MATNO(NELEM,1)	Material set number for each element
NDIME	Number of dimensions = 2 for two-dimensional problems = 3 for three-dimensional problems
NDOFN	Number of degrees of freedom = 1 for potential problems = 2 for plane elastic problems
NDLEG	Total number of edges along which generalized distributed loads are applied
NEASS(NDLEG,1)	Number of an element where the loaded edge locates
NEDGE	Number of edges per element = 3 for triangular element = 4 for quadrilateral element
NELEM	Total number of elements in the solution domain
NEQNS	Total number of equations (= total number of degrees of freedom = NPOIN*NDOFN)
NEVAB	Number of degrees of freedom per element (= NNODE*NDOFN)
NGAUS	Number of Gaussian points
NINTP	Number of RBF interpolation points
NMATS	Number of material sets

`NNODE`	Number of nodes per element = 4 for 4-nodes quadrilateral element = 8 for 8-nodes quadrilateral element
`NODEG`	Number of nodes per element edge = 2 for 4-nodes quadrilateral element = 3 for 8-nodes quadrilateral element
`NOFIX(NVFIX,1)`	Number of the nodes at which displacements are specified
`NOPRS(NDLEG,NODEG)`	Array of nodal numbers associated with the loaded edges
`NPLOD`	Total number of point loads, which are applied at nodes
`NPOIN`	Total number of nodes in the solution domain
`NPROP`	Number of material parameters required to define the characteristics of a material completely = 3 for heat conduction problems = 5 for plane elastic problems
`NSTRE`	Number of generalized stress components = 2 for 2D potential problems $q_1,\ q_2$ = 3 for 2D elastic problems $\sigma_{11},\ \sigma_{22},\ \sigma_{12}$
`NTREF`	Number of Trefftz functions
`NTYPE`	Types of problems under consideration = 0 for potential cases = 1 for plane stress cases = 2 for plane strain cases
`NVFIX`	Total number of nodes at which one or more degrees of freedom are specified
`POINT(NPLOD,NDOFN)`	Array of specified values of concentrated load along different degrees of freedom
`POSGP(1,NGAUS)`	Array of coordinates of Gaussian points
`PRESC(NVFIX,NDOFN)`	Array of specified values at restrained node along different degrees of freedom

`PRESS(NDLEG,NODEG*NDOFN)`	Array of values of the normal and tangential load intensities at nodes of an edge
`PROPS(NMATS,NPROP)`	Array of material properties for each material set Laplace problems, $k(=1), a, t_e$ Plane elastic problems, E, ν, t_e, a, ρ
`SCENP(NELEM,NSTRE)`	Array of generalized stresses at centres of each element
`SHAPE(1,NODEG)`	Array of shape functions along each element edge
`SHMTX(NDOFN,NEVAB)`	Array of shape functions of frame filed in each element
`UCENP(NELEM,NDOFN)`	Array of generalized displacements at centres of each element
`UPOIN(NPOIN,NDOFN)`	Array of nodal generalized displacements
`WEIGP(1,NGAUS)`	Array of weighting factors for Gaussian points

C

Glossary of subroutines

ASMSTIF	Assemble element stiffness matrix into global stiffness matrix
CMATRIX	Compute coefficient vector **c** in intra-element fields using generalised nodal displacements
EDISNOD	Collect nodal generalised displacements for a particular element
ELEPARS	Compute coordinates of nodes and centroids for each element
FIEDCEN	Compute generalised displacements and stresses at centroid of each element
FIEDNOD	Generate generalised displacement fields at nodes
GAUSSQU	Provide local coordinates of Gauss points and related weighting coefficient
GMATRIX	Compute **G** matrix for each element
HMATRIX	Compute **H** matrix for each element
INDISBC	Modify stiffness matrix and equivalent nodal forces by introducing specified displacement boundary conditions at nodes
INPUTDT	Input data from a file
KMATRIX	Compute stiffness matrix for each element
LSSOLVR	Solver of linear equation system
MAINFUN	Main functions for calling other subroutines in HT-FEM analysis

OPRESUT	Output of numerical results
PARSOLU	Evaluate approximated particular solutions at given point
PVECTOR	Compute equivalent nodal forces
QUANGAS	Compute quantities related to Gaussian integral, such as coordinates of Gaussian points, unit normal to boundary at Gaussian points, shape function matrix at Gauss points, and so on
RBFINTP	Compute coefficients of RBF interpolation
RIGIDRV	Recover discarded rigid motion
SCHTREF	Compute Trefftz functions corresponding to special circular hole element
SHAPFUN	Compute shape functions and its derivatives of line element
TELE442	Form local node-edge relations for 4-node quadrilateral element
TELE843	Form local node-edge relations for 8-node quadrilateral element
TREFFTZ	Compute complete Trefftz function and its derivatives
TYPELEM	Provide characteristics of nodes and edges of selected element

D

Plane displacement and stress transformations

In the practical analysis, the displacement and stress components might be expressed in terms of a local coordinates system which is different from the global one of a problem. It is sometimes convenient to introduce a rule which can transform field variables from one coordinates system to another. To this end, consider Figure D.1 showing the displacements and stress components in two coordinates systems (x_1, x_2) and $(\tilde{x}_1, \tilde{x}_2)$. The variable θ in Figure D.1 is the rotation angle between the x_1 and $\tilde{x}_1$ (positive in the counterclockwise direction).

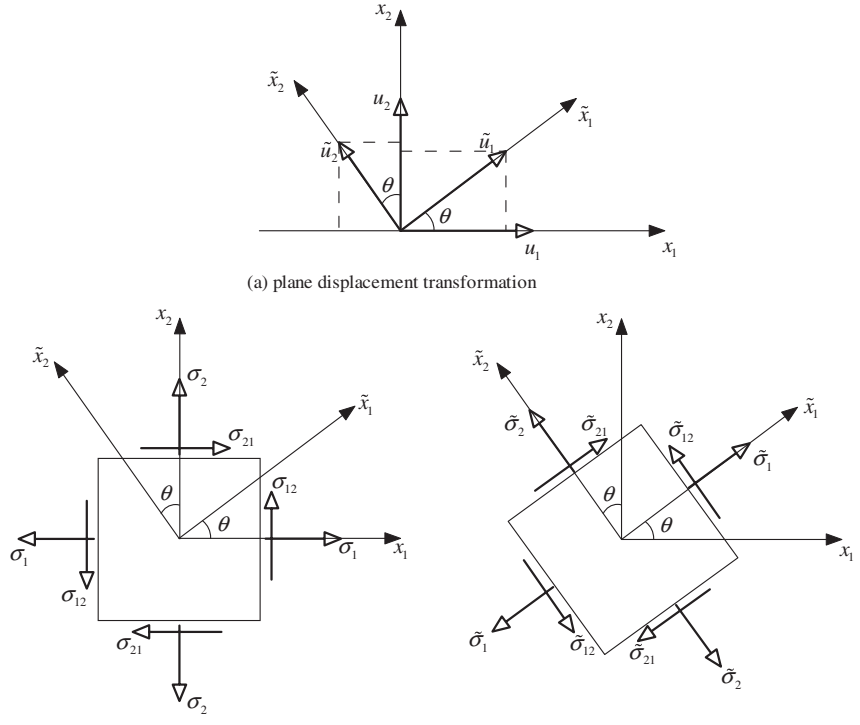


FIGURE D.1
Plane displacement and stress transformations

Noting that displacement field is a tensor of one order and stress field is a tensor of two-order, the transformation of displacement and stress fields from (x_1, x_2) to $(\tilde{x}_1, \tilde{x}_2)$ coordinates system can be expressed as

$$\begin{bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{D.1}$$

$$\begin{bmatrix} \tilde{\sigma}_1 \\ \tilde{\sigma}_2 \\ \tilde{\sigma}_{12} \end{bmatrix} = \begin{bmatrix} \cos^2\theta & \sin^2\theta & \sin 2\theta \\ \sin^2\theta & \cos^2\theta & -\sin 2\theta \\ -\frac{1}{2}\sin 2\theta & \frac{1}{2}\sin 2\theta & \cos 2\theta \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_{12} \end{bmatrix} \tag{D.2}$$

The transformation from $(\tilde{x}_1, \tilde{x}_2)$ to (x_1, x_2) system is obtained by the inverse of the two equations above

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \end{bmatrix} \tag{D.3}$$

$$\begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \cos^2\theta & \sin^2\theta & -\sin 2\theta \\ \sin^2\theta & \cos^2\theta & \sin 2\theta \\ \frac{1}{2}\sin 2\theta & -\frac{1}{2}\sin 2\theta & \cos 2\theta \end{bmatrix} \begin{bmatrix} \tilde{\sigma}_1 \\ \tilde{\sigma}_2 \\ \tilde{\sigma}_{12} \end{bmatrix} \tag{D.4}$$

It should be mentioned that the polar coordinates system (r, θ) can be viewed as a special case of general orthogonal coordinates system $(\tilde{x}_1, \tilde{x}_2)$. Therefore, if we replace $\tilde{u}_1$ and $\tilde{u}_2$ with u_r and u_θ in Eqs. (D.1) and (D.3), $\tilde{\sigma}_1$, $\tilde{\sigma}_2$ and $\tilde{\sigma}_{12}$ with σ_r, σ_θ and $\sigma_{r\theta}$ in Eqs. (D.2) and (D.4), a transformation between (r, θ) and (x_1, x_2) is obtained.