

Zia Javanbakht · Andreas Öchsner

# Advanced Finite Element Simulation with MSC Marc

 Springer

# Advanced Finite Element Simulation with MSC Marc

Zia Javanbakht · Andreas Öchsner

# Advanced Finite Element Simulation with MSC Marc

Application of User Subroutines

Zia Javanbakht  
Griffith School of Engineering  
Griffith University (Gold Coast Campus)  
Southport, QLD  
Australia

Andreas Öchsner  
Griffith School of Engineering  
Griffith University (Gold Coast Campus)  
Southport, QLD  
Australia

ISBN 978-3-319-47667-4      ISBN 978-3-319-47668-1 (eBook)  
DOI 10.1007/978-3-319-47668-1

Library of Congress Control Number: 2016956821

© Springer International Publishing AG 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature  
The registered company is Springer International Publishing AG  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*To our parents*

# Preface

This book relates to the general-purpose finite element program MSC Marc, which is distributed by the MSC Software Corporation. It is a specialized program for nonlinear problems (implicit solver) which is common in academia and industry. The primary goal of this book is to provide a comprehensive introduction to an advanced feature of this software: The user can write user subroutines in the programming language FORTRAN, which is the language of all classical finite element packages. This subroutine feature allows the user to replace certain modules of the core code and to implement new features such as constitutive laws or new elements. Thus, the functionality of this commercial code ('black box') can easily be extended by linking user-written code to the main core of the program. This feature allows to take advantage of a commercial software package with the flexibility of a 'semi-open' code.

Chapter 1 is a comprehensive introduction to the programming language FORTRAN. This chapter can be easily skipped by an experienced FORTRAN programmer. Nevertheless, it seems that there is a common trend at certain universities to avoid a classical programming language in engineering degrees. The engaged reader may find this chapter useful to overcome this serious deficiency. In contrast, it can play the role of a quick guide for an experienced user. Chapter 2 is a general introduction to the finite element package MSC Marc/Mentat. However, the focus is not on the handling of the graphical interface, i.e., the pre- and post-processor Mentat, but rather on topics which are important for the extension of the functionality. Chapter 3 introduces the different classes of subroutines, whereas the focus is on single-task examples which are mainly carried out by means of a single subroutine. Chapter 4 is devoted to more complicated examples which deal with several subroutines and their interaction and communication.

The instructions provided in this book relate to the MSC Marc/Mentat 2014.2.0 (64 bit) version under Microsoft Windows OS and the Intel XE 2013 FORTRAN (update 5) compiler (also known as Intel FORTRAN version 13). The application of user subroutines and common blocks might be slightly different for older versions, and the reader is in that case advised to adjust some of the given instructions and examples. The same must be expected for future versions. Finally, it should be

highlighted that the provided examples and computer codes are intended for purely educational purposes. The routines are tested and checked for specific application examples. Transferring routines to other examples may require certain adjustments and further extensive testing. Nevertheless, the provided results by these subroutines must be validated in every case.

We look forward to receive some comments and suggestions for the next edition of this textbook.

Southport, QLD, Australia  
September 2016

Zia Javanbakht  
Andreas Öchsner

# Acknowledgments

It is important to highlight the contribution of students which helped to finalize the content of this book. Their questions and comments during different lectures and their work in the scope of final-year projects helped to compile this book. Deeper insight into the software and many recommendations were provided by Dr. Andy Bell, senior technical consultant, MSC Software Corporation, UK. Furthermore, we would like to express our sincere appreciation to the Springer Publisher, especially to Dr. Christoph Baumann, for giving us the opportunity to realize this book. Finally, we would like to thank Marco Öchsner for his corrections and suggestions.



# Contents

|   |    |
|---|----|
| <b>1 Fortran – Advanced Features</b> . . . . .      | 1  |
| 1.1 Preliminary Concepts . . . . .                  | 1  |
| 1.1.1 Standard Syntax . . . . .                     | 2  |
| 1.1.2 Basic Definitions . . . . .                   | 3  |
| 1.1.3 Statement Order . . . . .                     | 7  |
| 1.1.4 Source File Format . . . . .                  | 10 |
| 1.1.5 Programming Conventions . . . . .             | 12 |
| 1.1.6 Naming Identifiers . . . . .                  | 15 |
| 1.2 Programming - Phases and Tools . . . . .        | 17 |
| 1.2.1 Planning the Logic . . . . .                  | 18 |
| 1.2.2 Pseudocode Conventions . . . . .              | 18 |
| 1.2.3 Flowchart Conventions . . . . .               | 20 |
| 1.3 Structured Programming . . . . .                | 20 |
| 1.3.1 Sequence, Selection and Repetition . . . . .  | 21 |
| 1.3.2 Combining Structured Logic . . . . .          | 24 |
| 1.4 Control Constructs in Fortran . . . . .         | 26 |
| 1.4.1 IF Construct . . . . .                        | 26 |
| 1.4.2 CASE Construct . . . . .                      | 28 |
| 1.4.3 DO Construct . . . . .                        | 31 |
| 1.4.4 REPEAT UNTIL . . . . .                        | 33 |
| 1.4.5 Altering the DO Construct . . . . .           | 33 |
| 1.4.6 Branching . . . . .                           | 33 |
| 1.5 Procedural/Modular Programming . . . . .        | 34 |
| 1.5.1 Structure of Program Units . . . . .          | 37 |
| 1.5.2 Subprograms . . . . .                         | 38 |
| 1.5.3 Procedure Referencing and Arguments . . . . . | 41 |
| 1.5.4 Modules . . . . .                             | 43 |
| 1.6 Specification Part . . . . .                    | 44 |
| 1.6.1 USE Statement . . . . .                       | 45 |
| 1.6.2 IMPLICIT Declaration . . . . .                | 45 |

- 1.6.3 Declaration Construct . . . . . 46
- 1.6.4 Association and Scope. . . . . 47
- 1.7 Data Type Declaration . . . . . 52
  - 1.7.1 Type Parameters . . . . . 54
  - 1.7.2 Data Representation. . . . . 55
  - 1.7.3 Intrinsic Data Types . . . . . 65
  - 1.7.4 Numeric Data Types . . . . . 67
  - 1.7.5 Non-Numeric Data Types . . . . . 71
  - 1.7.6 Expressions, Operators and Operands . . . . . 72
  - 1.7.7 Derived-Data Types. . . . . 73
  - 1.7.8 Arrays . . . . . 75
- 1.8 Data Attributes . . . . . 83
  - 1.8.1 PARAMETER Statement . . . . . 84
  - 1.8.2 PUBLIC Versus PRIVATE. . . . . 85
  - 1.8.3 SAVE and COMMON Attribute . . . . . 86
  - 1.8.4 DATA Statement and Explicit Initialization . . . . . 88
  - 1.8.5 INTENT and OPTIONAL Statement. . . . . 89
  - 1.8.6 ALLOCATABLE, POINTER and TARGET. . . . . 90
  - 1.8.7 CRAY Pointer. . . . . 92
  - 1.8.8 Interface Block . . . . . 94
- 1.9 Input and Output Management . . . . . 96
  - 1.9.1 Files, Records and Positions . . . . . 97
  - 1.9.2 Connection Statements . . . . . 98
  - 1.9.3 Data Transfer Statements. . . . . 102
  - 1.9.4 File Positioning Statements . . . . . 104
  - 1.9.5 INQUIRY Statement . . . . . 105
  - 1.9.6 Data Format . . . . . 105
- 1.10 Summary of Accessing Files . . . . . 111
  - 1.10.1 Sequential Formatted Access - Advancing Versus  
Non-advancing . . . . . 112
  - 1.10.2 Sequential Access - Unformatted. . . . . 116
  - 1.10.3 Direct Access - Formatted Versus Unformatted. . . . . 117
- 2 Introduction to Marc/Mentat . . . . . 121**
  - 2.1 MARC/MENTAT Interactions . . . . . 121
    - 2.1.1 Mentat Commands . . . . . 124
    - 2.1.2 MARC Solver Types. . . . . 125
    - 2.1.3 Structure of the Installation Folder . . . . . 125
  - 2.2 The Input File. . . . . 128
    - 2.2.1 Grouped Structure . . . . . 129
    - 2.2.2 Format Conventions . . . . . 131
    - 2.2.3 Extended Precision Mode . . . . . 133
    - 2.2.4 Modifying the Input File . . . . . 134
    - 2.2.5 Table-Driven Input . . . . . 135
    - 2.2.6 Items, Sets and Numbering . . . . . 141

- 2.3 Subroutines . . . . . 143
  - 2.3.1 Activating Subroutines . . . . . 144
  - 2.3.2 Structure of Subroutines . . . . . 145
  - 2.3.3 Predefined Common Blocks of Marc . . . . . 147
- 2.4 Debugging . . . . . 154
  - 2.4.1 Common Pitfalls . . . . . 155
  - 2.4.2 Requesting Additional Information . . . . . 158
  - 2.4.3 Activating the Debugging Mode . . . . . 161
  - 2.4.4 Compiler Directives . . . . . 162
  - 2.4.5 Controlling the Job Submission . . . . . 165
  - 2.4.6 Using the Visual Studio IDE . . . . . 169
- 2.5 Miscellaneous Tools . . . . . 174
  - 2.5.1 Procedure Files . . . . . 174
  - 2.5.2 Python and Mentat . . . . . 176
  - 2.5.3 C Programming Language . . . . . 177
- 3 Basic Examples . . . . . 181**
  - 3.1 Overview . . . . . 181
  - 3.2 Examples . . . . . 181
    - 3.2.1 FORCDT . . . . . 182
    - 3.2.2 FORCEM . . . . . 184
    - 3.2.3 WKSLP . . . . . 186
    - 3.2.4 PLOTV . . . . . 189
    - 3.2.5 HOOKLW and ORIENT2 . . . . . 192
    - 3.2.6 USDATA and UACTIVE . . . . . 196
    - 3.2.7 SEPFOR and MOTION . . . . . 198
    - 3.2.8 UINSTR . . . . . 201
    - 3.2.9 UBREAKGLUE . . . . . 202
    - 3.2.10 USHELL . . . . . 207
- 4 Advanced Examples . . . . . 211**
  - 4.1 Overview . . . . . 211
  - 4.2 Examples . . . . . 212
    - 4.2.1 USPRNG and UEDINC . . . . . 212
    - 4.2.2 UFXORD, UEDINC and UBGINC . . . . . 216
    - 4.2.3 USPLIT\_MESH . . . . . 219
    - 4.2.4 IMPD and NODVAR . . . . . 224
    - 4.2.5 ELMVAR and ELEVAR . . . . . 234
    - 4.2.6 UVSCPL . . . . . 241
    - 4.2.7 USELEM . . . . . 254
- 5 Listing of the Customized Modules . . . . . 271**
  - 5.1 Overview . . . . . 271
  - 5.2 Naming Rules and Abbreviations . . . . . 273
  - 5.3 Modules . . . . . 276

|        |                                |     |
|--------|--------------------------------|-----|
| 5.4    | MarcTools Module . . . . .     | 277 |
| 5.4.1  | CalcNodVal. . . . .            | 277 |
| 5.4.2  | DelElmFreeEdge . . . . .       | 278 |
| 5.4.3  | ExtractElmEdgeLst . . . . .    | 279 |
| 5.4.4  | ExtractElmNodLst . . . . .     | 280 |
| 5.4.5  | ExtractElmNodLst2 . . . . .    | 281 |
| 5.4.6  | ExtractNodCloseIPLst . . . . . | 282 |
| 5.4.7  | ExtractSetItemLst . . . . .    | 283 |
| 5.4.8  | GetElmArea. . . . .            | 284 |
| 5.4.9  | GetElmAveVal . . . . .         | 285 |
| 5.4.10 | GetElmCenCoord . . . . .       | 286 |
| 5.4.11 | GetElmEdgeVal. . . . .         | 287 |
| 5.4.12 | GetElmExtID. . . . .           | 288 |
| 5.4.13 | GetElmIPCCount . . . . .       | 289 |
| 5.4.14 | GetElmIntID . . . . .          | 289 |
| 5.4.15 | GetIPCoord . . . . .           | 290 |
| 5.4.16 | GetIPVal . . . . .             | 291 |
| 5.4.17 | GetNodCoord . . . . .          | 292 |
| 5.4.18 | GetNodExtID . . . . .          | 293 |
| 5.4.19 | GetNodExtraVal . . . . .       | 293 |
| 5.4.20 | GetNodIPVal. . . . .           | 295 |
| 5.4.21 | GetNodIntID . . . . .          | 297 |
| 5.4.22 | IsElmIDValid . . . . .         | 297 |
| 5.4.23 | IsItemInSet . . . . .          | 298 |
| 5.4.24 | IsNodIDValid . . . . .         | 299 |
| 5.4.25 | MakeElmIDLst . . . . .         | 299 |
| 5.4.26 | MakeIPCoordLst . . . . .       | 300 |
| 5.4.27 | MakeIPValLst . . . . .         | 301 |
| 5.4.28 | MakeNodCoordLst . . . . .      | 302 |
| 5.4.29 | MakeNodIDLst . . . . .         | 303 |
| 5.4.30 | MakeNodValIPLst. . . . .       | 303 |
| 5.4.31 | MakeNodValLst . . . . .        | 305 |
| 5.4.32 | PrintElmIDGroupedLst . . . . . | 306 |
| 5.4.33 | PrintElmIDLst . . . . .        | 307 |
| 5.4.34 | PrintIPCoordLst. . . . .       | 308 |
| 5.4.35 | PrintIPValLst. . . . .         | 309 |
| 5.4.36 | PrintNodCoordLst . . . . .     | 310 |
| 5.4.37 | PrintNodIDLst. . . . .         | 311 |
| 5.4.38 | PrintNodValIPLst . . . . .     | 312 |
| 5.4.39 | PrintNodValLst . . . . .       | 314 |
| 5.4.40 | PrintSetItemLstID . . . . .    | 315 |
| 5.4.41 | PrintSetItemLstName. . . . .   | 316 |
| 5.4.42 | PrintSetLst. . . . .           | 317 |

- 5.5 FileTools Module . . . . . 318
  - 5.5.1 AutoFilename . . . . . 318
  - 5.5.2 FindFreeUnit . . . . . 318
  - 5.5.3 DeleteFile . . . . . 319
- 5.6 MiscTools Module . . . . . 320
  - 5.6.1 DelRepeated . . . . . 320
  - 5.6.2 DelRepeated2D . . . . . 321
  - 5.6.3 ExtractIntersectLst . . . . . 321
  - 5.6.4 GetDistance . . . . . 322
  - 5.6.5 GetIndex . . . . . 323
  - 5.6.6 GetRandNum. . . . . 323
  - 5.6.7 PrintElapsedTime . . . . . 324
  - 5.6.8 PutSmallFirst . . . . . 325
  - 5.6.9 SwapInt . . . . . 325
  - 5.6.10 SwapReal . . . . . 326
- References** . . . . . 327
- Index** . . . . . 329

# Symbols and Abbreviations

## Latin Symbols (Capital Letters)

|       |  |
|-------|--|
| $A$   | Area, cross-sectional area                                       |
| $B$   | Matrix which contains the derivatives of interpolation functions |
| $E$   | Elastic modulus  |
| $F$   | Force  |
| $G$   | Shear modulus  |
| $I$   | Second moment of inertia   |
| $I_1$ | First invariant of stress  |
| $I_2$ | Second invariant of stress                                       |
| $J$   | Jacobian matrix  |
| $K$   | Spring constant  |
| $K$   | Stiffness matrix   |
| $L$   | Length   |
| $N$   | Column matrix of interpolation functions                         |
| $P$   | Point load   |
| $S_n$ | Breaking normal stress   |
| $S_t$ | Breaking tangential stress                                       |
| $T$   | Transformation matrix  |
| $X$   | Global Cartesian coordinate                                      |
| $Y$   | Global Cartesian coordinate                                      |
| $Z$   | Global Cartesian coordinate                                      |

## Latin Symbols (Small Letters)

|       |   |
|-------|---|
| $b$   | Base, radix                                   |
| $e$   | Integer value                                 |
| $e_i$ | Array extent of $i$ th dimension              |
| $f_k$ | $k$ th digit in floating-point representation |
| $f$   | Force vector                                  |
| $i$   | Iteration, integer values                     |

|              |  |
|--------------|--|
| $k$          | Encoding length, flow function         |
| $l_i$        | Lower array bound of $i$ th dimension  |
| $m$          | Residual function                      |
| $p$          | Number of mantissa digits              |
| $q$          | Distributed load, number of digits     |
| $r$          | Radius, radix                          |
| $s$          | Sign                                   |
| $t$          | Time, thickness, breaking index        |
| $u$          | Displacement                           |
| $u_i$        | Upper array bound of $i$ th dimension  |
| $\mathbf{v}$ | Solution vector                        |
| $w_k$        | $k$ th digit in integer representation |
| $x$          | Real value, Cartesian coordinate       |
| $y$          | Cartesian coordinate                   |
| $z$          | Cartesian coordinate                   |

### Greek Symbols (Small Letters)

|                       |   |
|-----------------------|---|
| $\delta$              | Displacement, contact interface thickness |
| $\epsilon$            | Strain, machine epsilon                   |
| $\xi$                 | Natural coordinate                        |
| $\kappa$              | Hardening parameter                       |
| $\lambda$             | Plastic multiplier                        |
| $\nu$                 | Poisson's ratio                           |
| $\sigma$              | Stress, normal stress                     |
| $\sigma_{\text{oct}}$ | Octahedral normal stress                  |
| $\tau_{\text{oct}}$   | Octahedral shear stress                   |
| $\tau$                | Shear stress                              |

### Mathematical Symbols

|                     |                           |
|---------------------|---------------------------|
| $\times$            | Multiplication sign       |
| $[\dots]$           | Matrix                    |
| $\text{sgn}(\dots)$ | Signum (sign) function    |
| $\partial$          | Partial derivative symbol |

### Indices, Superscripted

|                      |                |
|----------------------|----------------|
| $\dots^{\text{epl}}$ | Elasto-plastic |
| $\dots^{\text{e}}$   | Element        |
| $\dots^{\text{pl}}$  | Plastic        |
| $\dots^i$            | Iteration      |

## Indices, Subscripted

|                    |            |
|--------------------|------------|
| ... <sub>eff</sub> | Effective  |
| ... <sub>n</sub>   | Normal     |
| ... <sub>oct</sub> | Octahedral |
| ... <sub>s</sub>   | Shear      |
| ... <sub>tr</sub>  | True       |
| ... <sub>t</sub>   | Tangential |

## Abbreviations

|      |  |
|------|--|
| 1D   | One-dimensional  |
| 2D   | Two-dimensional  |
| 3D   | Three-dimensional  |
| CPU  | Central processing unit  |
| DOF  | Degree(s) of freedom   |
| FE   | Finite element   |
| GUI  | Graphical user interface   |
| ID   | Identification   |
| IP   | Integration point  |
| IDE  | Integrated development environment   |
| IEEE | Institute of electrical and electronics engineers                            |
| I/O  | Input/output   |
| LSB  | Least significant bit  |
| MPI  | Message passing interface  |
| MSB  | Most significant bit   |
| SI   | International system of units (from French ‘Système international d’unités’) |
| Sym. | Symmetric  |
| VS   | Visual Studio  |

## Some Standard Abbreviations

|      |  |
|------|--|
| e.g. | For example (from Latin ‘exempli gratia’)  |
| etc. | And others (from Latin ‘et cetera’)        |
| i.e. | That is (from Latin ‘id est’)              |
| viz. | Namely, precisely (from Latin ‘videlicet’) |



# Chapter 1

## Fortran – Advanced Features

**Abstract** Considering the fact that MARC is based on the FORTRAN programming language, not only is the basic knowledge of the language indispensable, but becoming familiar with advanced features will definitely improve the structure of the code. In this chapter, a comprehensive review of the advanced capabilities of the FORTRAN language will be presented.

### 1.1 Preliminary Concepts

A basic knowledge of the FORTRAN programming language will be required in order to gain the most from the subroutine capability of MARC. Thus in this chapter, some basic concepts of FORTRAN will be reviewed in conjunction with some general programming concepts. However, the keen reader can continue further reading through the references given in this book.

The best source as a comprehensive reference for FORTRAN is the ‘ISO/IEC 1539-1:2010 Fortran 2008 standard’ [19]. This standard describes the capabilities of FORTRAN 2008. The 2008 version is a slightly improved version of FORTRAN 2003 which is equipped with many new features in comparison with FORTRAN 77. It is the final result of a 50-year endeavor of standardizing FORTRAN, with the main objective of improving the portability of the developed programs among different operating systems and compilers. The standard is documented in a very organized fashion and it is definitely worth reading.

All the standard features of FORTRAN are descriptively expressed in the mentioned standard. However, the programming syntaxes are simplified in this book to improve the learning curve in favor of the reader. This simplification focuses on the fundamental skills to interact with MARC/MENTAT. The extensive capabilities of the language can be studied in more details as one progresses.

The main concern of the authors was to avoid the modern features of FORTRAN as much as possible and to limit themselves to the basic standard features which are almost available in every version. It is due to the fact that the development of MARC/MENTAT goes back to the older versions of FORTRAN and thus, some old commands still appear in the listings. Inevitably, these obsolescent features must be discussed.

In other words, one may incorporate a recent compiler into MARC/MENTAT but its FORTRAN 77-base will impose a lot of restrictions to the practice. For instance, it is not possible to use the *free format* for the FORTRAN source files despite the fact that the compiler supports it. The main source of restriction is within MARC itself (more on formats in Chap. 2).

In the following sections, important excerpts of FORTRAN programming will be provided as a concise reference in the framework of the fixed format.

### 1.1.1 *Standard Syntax*

The correct form of every FORTRAN entity is manifested by a specific *syntax*. In other words, a syntax is the correct grammatical presentation of a programming language. As mentioned earlier, simplified notations are adapted from the FORTRAN standard for consistent demonstrations of entities. However, the same notation of the standard will be used in some cases.

All FORTRAN listings, MARC/MENTAT codes, syntaxes and any other programming elements in the current book are displayed in San Serif. Additionally, numbered lines are used for FORTRAN and MARC/MENTAT listings to distinguish them from a syntax.

*Italic* fonts will be used for the first encounter of the words with specific meaning in the terminology of programming. For instance, a *statement* has a specific meaning in the nomenclature of the FORTRAN programming language which is different from its meaning in the normal English vocabulary.

Some abbreviations will be used in the syntactic language which are listed in Table 1.1. An executable program, for example, has the following syntax:

```
program-unit
[ program-unit ] ...
```

The interpretation of this syntax is that an executable program consists of at least one program unit (program-unit) plus one or more optional program units. As another example consider the syntax of the main program:

```
[ program-stmt ]
  [ specification-part ]
  [ execution-part ]
  [ internal-subprogram-part ]
end-program-stmt
```

This syntax indicates that a main program (main-program) consists of the following optional parts: a specification part (specification-part), an execution part (execution-part) and an internal subprogram part (internal-subprogram-part). However, a program must end with an end program statement (end-program-stmt) in all cases. A typical FORTRAN listing in the fixed format looks like the following:

**Table 1.1** Common syntactic abbreviations and symbols used in FORTRAN standards. Adapted from [19]

| Abbreviation/Symbol | Description                          |
|---------------------|--------------------------------------|
| stmt                | Statement                            |
| arg                 | Argument                             |
| attr                | Attribute                            |
| decl                | Declaration                          |
| def                 | Definition                           |
| desc                | Descriptor                           |
| expr                | Expression                           |
| op                  | Operator                             |
| spec                | Specifier                            |
| var                 | Variable                             |
| int                 | Integer                              |
| char                | Character                            |
| /                   | Or                                   |
| [ ]                 | Encloses an optional item            |
| [ ] ...             | Encloses an optionally repeated item |

1  
2  
3

```
PROGRAM Sample
...
END PROGRAM Sample
```

Note that ‘...’ in a listing indicates that at least one line of unimportant or irrelevant code may be intentionally omitted whereas in a syntax, it indicates repetitive patterns.

As mentioned earlier, the syntaxes introduced in the current book are either the same as those of the standard or a simplified versions of them. In either one, the abbreviations and symbols are kept intact in order to facilitate any possible references to the standard for more details.

### 1.1.2 Basic Definitions

There are several interrelated terms which are used in the context of programming, especially in the FORTRAN programming language. Understanding these terms and fully encompassing the meaning will be required for a deeper understanding of the language. It also improves the efficiency of using the standard. The most fundamental terms will be described briefly here and in more detail in the designated sections.

An *entity* is the most general term in the FORTRAN language and it can refer to any existing component of the language. Several combinations of the FORTRAN *character set* are used to produce various entities. An interesting analogy between the components of a natural language and a programming language has been mentioned in

**Table 1.2** Analogy between natural language and programming language

| Natural language element  | Programming language element |
|---------------------------|------------------------------|
| Letters of alphabet       | Character                    |
| Word and punctuation mark | Keyword and lexical token    |
| Sentence                  | Statement                    |
| Paragraph                 | Program unit                 |
| Text                      | Program                      |

**Table 1.3** Special characters. Adapted from [19]

| Character | Name                    | Character | Name                    |
|-----------|-------------------------|-----------|-------------------------|
|           | Blank                   | ;         | Semicolon               |
| =         | Equals                  | !         | Exclamation point       |
| +         | Plus                    | "         | Quotation mark or quote |
| -         | Minus                   | %         | Percent                 |
| *         | Asterisk                | &         | Ampersand               |
| /         | Slash                   | ~         | Tilde                   |
| \         | Backslash               | <         | Less than               |
| (         | Left parenthesis        | >         | Greater than            |
| )         | Right parenthesis       | ?         | Question mark           |
| [         | Left square bracket     | '         | Apostrophe              |
| ]         | Right square bracket    | `         | Grave accent            |
| {         | Left curly bracket      | ^         | Circumflex accent       |
| }         | Right curly bracket     |           | Vertical line           |
| ,         | Comma                   | \$        | Currency symbol         |
| .         | Decimal point or period | #         | Number sign             |
| :         | Colon                   | @         | Commercial at           |

[21] which is summarized in Table 1.2. In this table, the elements are sorted from the smallest to the largest one. Namely, similar to the fact that words are made up of the alphabet, in a programming language lexical tokens are made up of characters. Similarly, sentences are made by joining words, as statements are made by joining lexical tokens. This table is of a great help to understand the structure of a programming language; FORTRAN in our case.

An *alphanumeric character* is either a letter (upper- or lowercase one), a digit or an underscore. Alphanumeric characters plus *special characters* (see Table 1.3) make up the FORTRAN character set. Other graphical characters normally just appear within a *comment*. Special characters are used as *separators*, *delimiters*, *operator symbols* etc.

**Table 1.4** Keywords with and without optional blanks. Adapted from [17]

| Possible form   | Suggested form   |
|-----------------|------------------|
| ENDMODULE       | END MODULE       |
| ENDSUBROUTINE   | END SUBROUTINE   |
| ENDFUNCTION     | END FUNCTION     |
| ENDDO           | END DO           |
| ENDIF           | END IF           |
| ENDTYPE         | END TYPE         |
| ENDMODULE       | END MODULE       |
| DOUBLEPRECISION | DOUBLE PRECISION |
| ELSEIF          | ELSE IF          |
| GOTO            | GO TO            |
| SELECTCASE      | SELECT CASE      |

A *token* or a *lexical token* is the smallest meaningful unit of a statement which consists of one or more characters and can be either one of the followings:

- a name,
- a label,
- an operator symbol,
- a keyword,
- a *Literal constant* except for complex literal constants,
- a delimiter or a separator ( , ) , / , [ , ] , ( / , / ) , or
- a comma , = , => , ; , :: , ; , % .

For example,  $z=x*y$  consists of five tokens.

*Blanks* or *white spaces* can be used to improve the readability of the code and in this case, multiple blanks are considered as one. However, it is not permitted to use blanks within a token, and of course they are significant within the strings. Additional, there are some optional blanks which improve the readability. Therefore, it is advised to use them as they have been used in the listings of the current book. A short list is provided in Table 1.4.

A *name* or an *identifier* consists of alphanumeric characters; a maximum of 31 characters in FORTRAN 90 and 63 characters in FORTRAN 2008. It is used to reference various entities such as any of the followings:

- a *variable*,
- a *named constant*,
- a program unit,
- a *common block*,
- a *procedure*,
- an *argument*,
- a *construct*,

- a *derived type*,
- a *namelist group*,
- a *structure component*,
- a *dummy argument*, or
- a *function result*.

All alphanumeric characters can be used to form a name, with the following exceptions:

1. no blanks can be used, e.g. instead of `my variable` which is invalid, `myVariable` shall be used.
2. the first character cannot be a number, e.g. `1variable` is invalid whereas `variable1` is valid, and
3. of course non-alphanumeric characters are not allowed, e.g. `#variable` is not valid.

In some cases, a *label* must be defined for a statement. A statement label can be one to five digits long and it starts from the first column of the file. Leading zeros are insignificant in labels and thus, at least one out of five digits must be non-zero. Therefore, a label such as 00050 is equal to 50. Although using labels is an obsolescent feature of FORTRAN, there are cases in which using labels is inevitable. For instance, labels can be used for frequently referred formats defined by the `FORMAT` statement. The common way of labeling a point within a piece of code is using the keyword `CONTINUE` in front of a label. Such lines do nothing but merely act as place-holders of the labels, for instance:

```
1 100 CONTINUE
```

Note that `CONTINUE` has a different effect if used in a loop (more details in Sect. 1.4).

There are no *reserved words* in FORTRAN because every entity is being interpreted by the compiler based on its context. However, to improve the readability of the code, meaningful *identifiers* should be used in the context of the program (see Sects. 1.1.5 and 1.1.6). For instance, consider the following code:

```
1  INTEGER :: if , then , end
2
3     if = 4
4     then = 4
5     end = 2
6  IF (if.eq.then) then = end*if+then
```

This piece of code is perfectly valid. However, the readability of the code is low and hence, it is not advised to use such ambiguous lines. This listing can be improved to the following by using proper identifier names and a good combination of indentations, capitalization, spaces and delimiters:

```
1  INTEGER :: currentRate , finalRate , tolerance
2
3     currentRate = 4
4     finalRate = 4
5     tolerance = 2
6
7     IF (currentRate .EQ. finalRate) THEN
8         finalRate = tolerance * (currentRate + finalRate)
9  END IF
```

**Table 1.5** Keyword types in FORTRAN

| Type of keyword | Appears in...              | Examples                                 |
|-----------------|----------------------------|--|
| Statement       | Syntax of a statement      | IF, READ, UNIT, INTEGER and etc.         |
| Argument        | Argument list              | aChar (45, KIND = 1)                     |
| Component       | Constructor of a structure | person (NAME = 'Chris', AGE = 21)        |
| Type parameter  | Type parameter list        | CHARACTER (LEN = 10, KIND = 1) :: myChar |

A *keyword* is a word used by the programming language which has a special meaning. Namely, it is a key ingredient of a syntax which distinguishes one syntax from another. There are four types of keywords in FORTRAN as listed in Table 1.5. A *statement keyword* is a keyword used to define a statement. The other three types, namely the *argument keyword*, the *component keyword* and the *type parameter keyword*, are used to identify a name in a list: the argument list, the constructor type and the type parameter list, respectively. The KEYWORD= syntax can be used for these three types. This syntax can be optional in some cases, as it increases the readability of the code. In addition, it omits the chance of mistakes when the right order of items is not considered or some optional items are not being used.

It is a legacy of FORTRAN 77 to use keywords in all capital letters. Following the same legacy, upper-case letters are used in the current book to add more emphasis to them. A list of FORTRAN keywords is made available in Table 1.6. Note that in this table, the newer versions of FORTRAN have all the keywords of the previous versions plus the new ones.

### 1.1.3 Statement Order

A *statement* (stmt) is a sequence of one or more complete or partial lines satisfying a syntax rule. If a statement carries out a specific action or control, then it is an *executable statement*. However, if it configures an environment in which an action takes place, then it is a *non-executable statement*. There are some restrictions and obligations for where a statement is allowed to appear. These rules define the *statement order*. The general syntax for the order of every program unit in FORTRAN is as the following:

```
[ program-stmt ]
  [ specification-part ]
  [ execution-part ]
  [ internal-subprogram-part ]
end-program-stmt
```

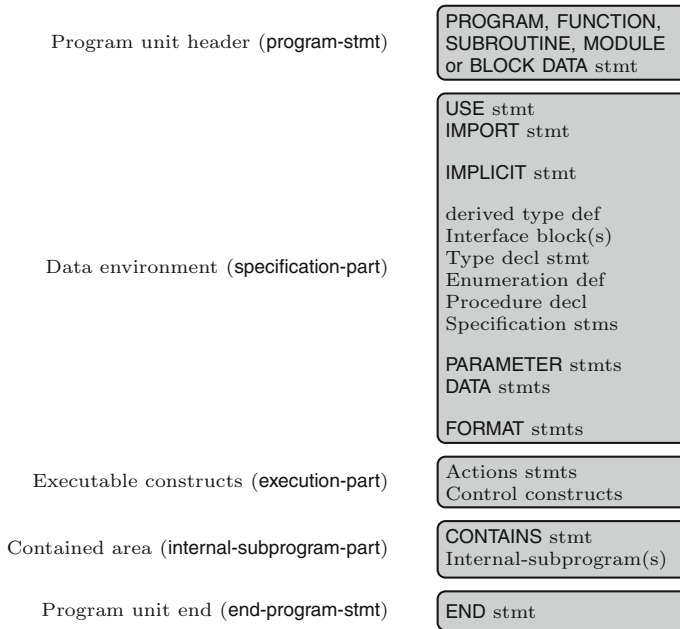
**Table 1.6** List of FORTRAN keywords. Adapted from [3]

| FORTRAN version | Keywords   |
|-----------------|--|
| 77              | ASSIGN BACKSPACE BLOCK DATA CALL CLOSE COMMON CONTINUE<br>DATA DIMENSION DO ELSE ELSE IF END ENDFILE<br>ENDIF ENTRY EQUIVALENCE EXTERNAL FORMAT FUNCTION GOTO<br>IF IMPLICIT INQUIRE INTRINSIC OPEN PARAMETER PAUSE<br>PRINT PROGRAM READ RETURN REWIND REWRITE SAVE<br>STOP SUBROUTINE THEN WRITE |
| 90              | ALLOCATABLE ALLOCATE CASE CONTAINS CYCLE DEALLOCATE<br>ELSEWHERE EXIT INCLUDE INTERFACE INTENT MODULE<br>NAMELIST NULLIFY ONLY OPERATOR OPTIONAL POINTER<br>PRIVATE PROCEDURE PUBLIC RECURSIVE RESULT SELECT<br>SEQUENCE TARGET USE WHILE WHERE  |
| 95              | ELEMENTAL FORALL PURE  |
| 2003            | ABSTRACT ASSOCIATE ASYNCHRONOUS BIND CLASS DEFERRED<br>ENUM ENUMERATOR EXTENDS FINAL FLUSH GENERIC<br>IMPORT NON_OVERRIDABLE NOPASS PASS PROTECTED VALUE<br>VOLATILE WAIT  |
| 2008            | BLOCK CODIMENSION DO CONCURRENT CONTIGUOUS CRITICAL<br>ERROR STOP SUBMODULE SYNC ALL SYNC IMAGES SYNC MEMORY<br>LOCK UNLOCK  |

The break-down of this general syntax is illustrated in Fig. 1.1. Although some statements are not permitted in every program unit, the general ordering requirements are shown in this figure. Therefore, it can be used as a blueprint for all types of program units. Based on this figure, the following major parts can be recognized in a FORTRAN program:

1. the *program statement* (program-stmt) is the *header* of the *program unit* and it indicates the type of program unit and its allowable statements. Program units are described in Sect. 1.5.1.
2. The *specification part* (specification-part) prepares the *data environment* in which data entities are declared or brought in from the outside of the unit (Sects. 1.6 and 1.7). This task is carried out by means of non-executable statements consisting of the following elements:
  - a. USE statement (Sect. 1.5.1)
  - b. IMPORT statement (Sect. 1.8.8)
  - c. IMPLICIT statement (Sect. 1.6.2)
  - d. Derived type definition(s) (Sect. 1.7.7)
  - e. Interface block(s) (Sect. 1.8.8)
  - f. Type declaration statement(s) (Sect. 1.7)





**Fig. 1.1** Schematic statement ordering of a FORTRAN program

- g. Enumeration definition which are integer constants used to interact with C programming language (not covered),
  - h. Procedure declaration which are related with polymorphisms and object oriented programming (not covered),
  - i. Specification statements among which are PARAMETER and DATA statements are all described in Sect. 1.6
  - j. FORMAT statement (Sect. 1.9.6)
3. The *execution part* (execution-part) is for executable constructs (discussed later in this subsection).
  4. The *internal subprogram part* (internal-subprogram-part) starts with the CONTAINS statement and is followed by internal subprograms or module subprograms (Sect. 1.5.2).
  5. The *end program statement* (end-program-stmt) is a single END statement which dictates the end of the program unit (Sect. 1.5.2).

Setting up the data environment starts with the IMPLICIT statement and is finished by the first executable statement (described in Sect. 1.7). Although between USE and CONTAINS statements, non-executable statements are in priority to the executable ones, there are exceptions such as FORMAT, DATA and ENTRY which may appear anywhere in the code.

**Table 1.7** Statement restrictions in scoping units. Adapted from [19]

| Statement                | Main Program | Module | Block data | External subprogram | Module subprogram | Internal subprogram | Interface body |
|--------------------------|--------------|--------|------------|---------------------|-------------------|---------------------|----------------|
| USE                      | ✓            | ✓      | ✓          | ✓                   | ✓                 | ✓                   | ✓              |
| IMPORT                   | ✗            | ✗      | ✗          | ✗                   | ✗                 | ✗                   | ✓              |
| ENTRY                    | ✗            | ✗      | ✗          | ✓                   | ✓                 | ✗                   | ✗              |
| FORMAT                   | ✓            | ✗      | ✗          | ✓                   | ✓                 | ✓                   | ✗              |
| Misc. decl. <sup>a</sup> | ✓            | ✓      | ✓          | ✓                   | ✓                 | ✓                   | ✓              |
| DATA                     | ✓            | ✓      | ✓          | ✓                   | ✓                 | ✓                   | ✗              |
| Derived type             | ✓            | ✓      | ✓          | ✓                   | ✓                 | ✓                   | ✓              |
| Interface                | ✓            | ✓      | ✗          | ✓                   | ✓                 | ✓                   | ✓              |
| Executable               | ✓            | ✗      | ✗          | ✓                   | ✓                 | ✓                   | ✗              |
| CONTAINS                 | ✓            | ✓      | ✗          | ✓                   | ✓                 | ✗                   | ✗              |

<sup>a</sup>Misc. declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, enumeration definitions, procedure declaration statements, and specification statements

The FORMAT statement is used to format the output and it is best to be placed after the specification part (described in Sect. 1.9). The ENTRY keyword is used for multiple entries to a subprogram which is better to be avoided since it is against structured programming rules (Sect. 1.3).

In addition to the mentioned ordering, Table 1.7 lists the allowed statements in various scoping units; for instance, in a block data, no executable statements are allowed or the only place that an IMPORT statement can be used is within an interface body.

An *executable construct* is either a *control construct* (Sect. 1.4) or an *action statement*. A brief list of action statements is available in Table 1.8. All these action statements, manipulate the data environment to produce the intended results of a program.

### 1.1.4 Source File Format

A FORTRAN source file is simply a text file<sup>1</sup> with a ‘.f’, ‘.for’ or ‘.f90’ file extension. Normally, a FORTRAN character set is used to form the source file. This file consists of several sequential lines of statements, comments and/or file inclusions. MARC only recognizes the ‘.f’ extension which emphasizes the fact that it only accepts a fixed format FORTRAN text file as opposed to ‘.f90’ which indicates a free format text file. Hence, the fixed format will be focused on in this book.

In the fixed format, the columns of the source file are banded and a statement must be indented to be placed in its corresponding band. A FORTRAN77 compiler

---

<sup>1</sup>An ASCII-file.

**Table 1.8** Brief list of action statements in FORTRAN

| Action statement                   | Description  |
|------------------------------------|--|
| Variable = expression              | Assignment of an expression to a variable          |
| CALL                               | Used to call a subroutine                          |
| GOTO                               | Jumps to another label                             |
| CONTINUE                           | Place-holder for a label                           |
| CYCLE                              | Terminates the current cycle of a DO construct     |
| RETURN                             | Returns the control of the program to invoker      |
| STOP                               | Stops the execution of the program                 |
| EXIT                               | Terminates a DO construct                          |
| PRINT                              | Prints the items of the list on the screen         |
| WRITE                              | Writes data to a file unit or output               |
| READ                               | Reads data from a file unit                        |
| OPEN                               | Establishes a connection between a file and a unit |
| CLOSE                              | Terminates the connection of a file to a unit      |
| Data-pointer-object => data-target | Associates a pointer to a target                   |
| ALLOCATE                           | Allocates memory for dynamic entities              |
| DEALLOCATE                         | Releases the allocated memory of dynamic entities  |
| NULLIFY                            | Disassociates a pointer from any target            |

only recognizes columns 1–72 which comprises of four distinguished bands as the following:

1. Column 1: can be blank or otherwise if the line is a *comment line* then it starts with the character ‘c’, ‘\*’ or ‘!’. Note that the exclamation mark can be used in other columns as well and the compiler will ignore whatever comes after that.
2. Columns 1–5: can hold a label which is optional.
3. Column 6: any characters including the exclamation mark in this column indicate a continuation of the previous line. One usual approach is using numbers after the initial line, i.e. for the first continuation line ‘1’, for the second ‘2’ etc. In addition, the ‘c’, ‘\*’, ‘+’ or ‘&’ character are also used to indicate a continuation by convention. However, an ampersand (&) will be used in this book.
4. Columns 7–72: any FORTRAN statement must be placed in this band. One exception is the *compiler preprocessor directives* which start from the first column (see Sect. 2.4.4).

It is possible to use tab characters instead of blanks or even a combination of both. However, it is advised to use only blanks to form a consistent format. Tab characters are interpreted differently based on the editor program and hence, using only blanks as leading spaces (column 1–6) is advised. It is also possible to have multiple statements in a single line, each terminated by a semicolon. However, it is not recommended

**Table 1.9** Comparisons of fixed- and free format of a FORTRAN source file

| Restriction                  | Free format   | Fixed format                           |
|------------------------------|---------------|--|
| Start of a line              | Anywhere      | Column 7                               |
| Empty lines                  | Allowed       | Allowed                                |
| Maximum line length          | 132           | 72                                     |
| Blank character              | Insignificant | Significant                            |
| Commenting character         | !             | !, c or * in column 1 <sup>a</sup>     |
| Statement termination char.  | ;             | ; <sup>b</sup>                         |
| Statement continuation char. | &             | Any character in column 6 <sup>c</sup> |
| Maximum continuation lines   | 255           | 19                                     |
| Label                        | Columns 1–5   | Columns 1–5                            |

<sup>a</sup>Exclamation mark ‘!’ can be used in any column except column 6 which is considered a continuation character

<sup>b</sup>Semicolon is optional as a termination character but can be used in between each two statements in a single line as a separator. However, this method reduces the readability of the code and is not advised

<sup>c</sup>Ampersand ‘&’ is common in the sixth column of a continuation line for a fixed format source. In contrast, in a free format file, the last non-blank character of the initial line must be an ampersand to indicate the next line as a continuation line

because it reduces the readability of the line. Finally, a summary of these points and some additional details of these two formats are presented in Table 1.9.

### 1.1.5 Programming Conventions

There are several considerations when it comes to programming but one of the most important ones is being clear; both to the compiler and to the user. While working on a piece of code, everything seems crystal clear at the time being but reconsidering the same code after a while may be confusing and time consuming. Depending on to which extent the code is organized, less time will be required in the future references. Hence, a rule of thumb is to keep things as simple and straightforward as possible.

In order to have crystal-clear parts of code, it is necessary to understand and to incorporate concepts such as *structured programming*, *modular programming* and *encapsulation*. Using these concepts will aid the programmer to manage the frequently-used pieces of code in a neat and clean way. In addition to this systematic approach, it is advised to use some *programming conventions* consistently throughout a code to increase its textual readability and clarity. A few of these customs will be discussed here briefly and be used throughout the listings of this book (for a more descriptive list see [9]):

1. FORTRAN does not distinguish between the lower- or the upper-case characters, i.e. it is case-insensitive. Therefore, a programmer can take advantage of this fact

to generate an easy-to-read code by using only lower-case characters throughout the listing, minding the following exceptions:

- *named constants* are used in capital letters and sometimes to improve readability some underscores are added between the words, e.g. POISSON, PI and TOTAL\_NODES.
  - *Keywords* are used in capital letters, e.g. PROGRAM and INTEGER.
  - *Action statements* are used in capital letters, e.g. WRITE, CALL and GOTO.
  - lowerCamelCase format is used for naming variables, e.g. tangentStiffness and curvatureRadius.
  - UpperCamelCase format is used for the name of user defined subprograms and modules e.g. ExtractStiffness and CalcRadius.
  - Start intrinsic subprograms with a capital letter, e.g. Sin, Sqrt and Abs.
2. Blank spaces are significant in columns 1–6 of a fixed format code but in the columns 7–72, they can be used for indentation. Using indentations in a source file makes it easier to recognize related lines of code. Both tab and blank characters can be used for this purpose but in this book blank characters are utilized. It is good practice to consistently use a two-space indentation to align related lines of the code for more emphasize, such as the following items:
- nested control structures, e.g. IF-ELSE-END IF and DO-END DO.
  - Elements of a data environment, e.g. declarations of named constants, variables and derived type definition.
  - *Scope* changes, e.g. statements of a subprogram are indented with respect to the program unit header.
  - Use a single space after a comma, e.g. CALL SumUp (a, b, c).
  - Use a single space after the name of a subprogram especially in the case of a function to distinguish it from an array, e.g. in  $a = \text{test1}(i, j) + \text{Test2}(i, j)$  statement, test1 is an array and Test2 is a function.
  - Use a single space after and before an *operator* in an *expression*, e.g.  $a = b + c$  instead of  $a=b+c$ .
3. Continuation lines are inevitable especially in a fixed format code, for that the following points are advised:
- whenever possible, break a long line into meaningful pieces to increases the readability of the code,
  - use an ampersand (&) to mark continuation lines.
  - use a consistent indentation for continuation lines with respect to the initial line.
4. *Short statements* which are separated by a *statement separator*, i.e. a semicolon (;) are allowed but not suggested.
5. Comments are not considered by the compiler but if used correctly they will clarify the code to a great extent. Consider the following points in commenting:

- avoid using comments at the end of code lines, i.e. *trailing comments* by exclamation mark (!). Instead, group a series of similar code and add a full-line comment before it to increase the readability.
- Use a consistent comment marker throughout the code, e.g. an asterisk (\*) is a good choice because it can be used to separate blocks of code by forming a border around comments, or an exclamation mark (!) can be used as a simpler alternative.
- A comment line cannot be continued because the continuation mark itself is considered as a part of the comment.
- In managing large databases, often a *data dictionary* is used which is the description of the data fields. It is possible to apply the same concept in modular programming. In other words, for a program unit, especially a subprogram or module, it is a useful practice, while establishing the data environment, to create a list of used data objects plus a brief description of each. This list is usually placed at the beginning of the subprogram in the form of several comment lines. Using such a data dictionary, makes the maintenance process of a program easier and it is strongly advised while working with modular programming and subprograms. For instance in a subprogram, add comments consisting of an explanation of the data environment, i.e. input, output, auxiliary variables and a brief description of the procedure. But at the same time, try to avoid repeating the details which are evident from the subprogram itself. The following listing is the comments part for the `GetNodeCoord` function of the `MarcTools` module (see Sect. 4.1). These lines clearly describe the inputs and outputs of the function by providing the details of the data environment:

```

1  !*****
2  !
3  ! Objective:
4  ! Returns the undeformed/deformed coordinates of a node ID.
5  ! Returns three zero coordinates in the case of an error.
6  !
7  ! Input(s):
8  !   nodID      INTEGER      external node ID
9  !   nodState   INTEGER(OPTIONAL)  the state of the node
10 !                                           1: undeformed state
11 !                                           2: deformed state
12 ! Output(s):
13 !           REAL*8(3)      coordinates of the nodID
14 !
15 ! Auxiliary variable(s):
16 !   nComp      INTEGER      number of returned components
17 !   dataType   INTEGER      the returned data type
18 !   nodCoord   REAL*8(3)    coordinates of the nodID
19 !   nodDisp    REAL*8(3)    displacement of the nodID
20 !   i          INTEGER      loop counter
21 !   coordState INTEGER      holds the same state as nodState
22 !                                           if the latter is present
23 !
24 ! Required common blocks:
25 !   none
26 !
27 ! Required subprogram(s):
28 !   none
29 !
30 ! Restrictions:
31 !   none

```

```

32  !
33  ! Future Updates:
34  !   none
35  !
36  ! Last update: 23/3/2016
37  !*****

```

This much detail may seem excessive but it extremely reduces the effort in the maintenance of the code. It is advised that the comment area and the data dictionary to be build during the coding and not after finishing the program. They both add vital notes for keeping and updating the code. Such notes are usually forgotten after the code is finished. This importance is magnified specially for the subprograms of a module which are constantly required to be updated during the development of the program.

Although most of the listed remarks are customary among programmers, they depend on personal preferences. It is advised to keep these preferences consistent through the programming project. In other words, one should consider them as a personal code of practice which finally generates a homogeneous result.

### 1.1.6 Naming Identifiers

One last topic which may be closely akin to one’s preferences is naming the identifiers in the program. Identifiers are used to distinguish the entities from each other, namely to identify program units, variables, named constants, and others. An identifier is merely a name that is assigned to an entity for further referrals. Therefore, an identifier is simply called a ‘name’.

It is good to start any work with a personal convention and select some abbreviations for the quantities which will be dealt with. It is important to stick to your, even if not perfect, convention throughout the project, than to follow no conventions at all. Note that similar to the capitalization conventions discussed earlier, the following points are merely suggestions, not rigid rules, for choosing a proper name for an entity.

Generally, a proper name should be *context-related, clear in meaning and as short as possible*. Satisfying all these factors may not be possible in every situation but it is advised to keep to them as much as possible.

In naming entities, context shall be considered. In other words, a name shall be chosen in such a way that implies the context of the entity. For instance, it is advised that a verb should be used to name a subprogram. This is due to the fact that subprograms are used to carry out tasks. For example CalcArea is a good name for a subroutine which implies its functionality, i.e. calculating the area. Aside from the capitalization custom, the same name is not as clear for a variable holding the calculated area, instead area would be a better choice.

Since there is a trade-off between being clear and using less characters, selecting a name should be done be considering both its meaningfulness and length. Obviously, it

is possible to completely clarify a name by using more characters but in programming languages like FORTRAN, there is a limitation in the number of characters of a name.<sup>2</sup> Although the facility is provided, it is recommended to avoid long names. It is due to the fact that MARC does not support the free format yet. Another reason for avoiding long names is because they make the typing process a tedious job. The remedy to this is using some abbreviations instead of long phrases.

A long name is usually downsized to an abbreviation with 3–5 characters. There are some suggestion in doing so. Usually if a name consists of four letters or less, omitting any letters will cause the loss of meaning. Therefore, all the letters should be kept, e.g. the node name can be used for a node or the name area for an area. Further omitting can be done only if it does not affect the meaning. For example, the name node can be reduced to nod and the meaning is still preserved. But further truncation will cause the loss of meaning and should be avoided, i.e. the names no or n. Also the name area cannot be shortened anymore to are.

For names with more than four letters, an abbreviation will be used which usually consists of the three to four initial letters. Generally, to preserve the meaning, the trailing letters are omitted, e.g. coord is used for ‘coordinates’. However, one may prefer to use elm instead of ele to stand for the name element but generally removing the vowels is not recommended.

Adhering to a consistent rule will keep the abbreviations uniform. However, a guidance table should be created holding the description of each entity. For example see Table 5.1 which holds the name of the subprograms and the explanation of their tasks. Another example is Table 5.2 which holds the summary of the variables.

Generally, it is good to use more descriptive names for ubiquitous entities such as entities with a global scope. In contrast, use short but clear names for entities with limited use such as local variables. Naming local variables is less important than a global one. Similarly, naming a frequently used subprogram is of utmost importance.

The mentioned rules apply for naming the subprograms. However, one can be more generous here in terms of the number of used characters. Since subprograms are more frequently encountered rather than a local variable, longer names are preferred. Note that a long name should be used if it adds to the meaning. In addition, the identifier of a subprogram usually starts with a verb such as Get, Set, Calc, and so on.

For a quantity which can be named by a phrase, the mentioned rules can be applied for each of its words. For instance in the context of finite elements, the nodCoord name can be used for a quantity which holds the coordinates of a node.

The order of the multi-word identifiers should be kept the same. Also, it is good to order the names by importance. Namely, for the name of a variable which holds a list of elements there are two ordering options: elmLst or lstElm. Because this name indicates in this case that it is related to some elements, the elm part of the identifier is more important. Therefore, it must come in the first place, i.e. the elmLst name is preferred. This type of sorting will ease the search for similar name since one knows that anything related to elements start with the elm.

---

<sup>2</sup>In FORTRAN 95, names can be 31 characters long. This length is improved to 63 characters in FORTRAN 2003.



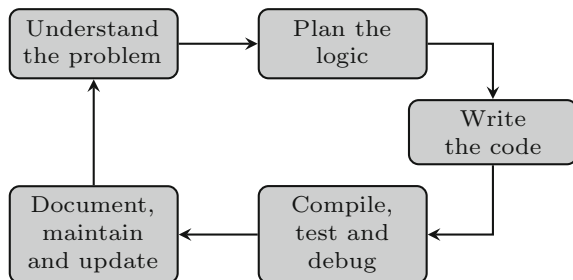
Aside from the mentioned points, there are some conventions among programmers. For instance, the letters *i*, *j* and *k* are usually used as loop counters (possibly originated from the old FORTRAN codes with implicit type declaration) or the letter *n* is used as a prefix for the number of a quantity. One could get familiar with such conventions by acquiring more experience. To get some starting ideas, one may refer to [4, 5, 8, 9, 20, 21, 32].

## 1.2 Programming - Phases and Tools

Programming is arranging instructions for a computer to manage a task. During this process, some errors might be introduced which may be due to a wrong use of a command such as misspelling a keyword, called a *syntax error*. In contrast, sometimes the syntax is right but the logic is not sound; a *semantic error*, i.e. the program executes but the result is not as expected; it is rather hard to debug this kind of error. In order for a program to execute properly, not only must the instructions obey the correct syntax, but also the development of the code must be based on the right logic. Figure 1.2 illustrates the iterative process of developing a program.

The problem must be described as in detail as required pertaining to and not limited to its necessary capabilities and producible outputs. The solution of the problem must be logically planned in a, not necessarily unique, series of steps; called an *algorithm*. In the next steps, the programmer codes, compiles and tests the program for debugging purposes and then applies updates or maintenance modifications as required. Among the mentioned phases, the logic planning is the most complicated one because in that, semantic errors might be introduced to the cycle. The two common tools for logical planning are *flowcharts* and *pseudocodes* which both are independent of the programming language and they will be investigated briefly in the following sections.

**Fig. 1.2** Iterative stages of programming. Adapted from [13]



### 1.2.1 *Planning the Logic*

In programming, there exists a task/problem at hand, a goal which will be achieved by a series of steps of calculations and/or actions. In order to solve such a problem, there may be several methods, each represented by an algorithm, i.e. a sequence of small steps which will finally produce the solution of the problem. In computer science, the word ‘algorithm’ is used in a more specific way than merely a method: “...a precise method usable by a computer for the solution of a problem” [15]. This definition necessitates that finite steps of any algorithm to be ordered, well-defined, precise and unambiguous resulting in a clear *computational procedure*.

As mentioned earlier, pseudocodes and flowcharts are two useful tools for structured programming. The transition from an algorithm to a programming code is bridged using a *pseudocode* and/or a *flowchart*. A pseudocode is nothing but a step-by-step plain English language representation of the algorithm and a flowchart is a graphical representation for that. Although choosing between these two is subjective, recognizing patterns via a flowchart is sometimes much easier because of its more visual nature. Using these tools may look trivial for short programming codes but in long run and especially for comprehensive projects they could save a lot of confusion and time.












### 1.2.2 *Pseudocode Conventions*

Generally, there are no specific rules about how pseudocodes must be used since they are just step-wise instructions on how to carry out an algorithm. However, the following points may be useful during logical planning of the program:

- Pseudocodes are not language dependent. Therefore, no correct syntax is defined for a pseudocode. In addition, grammatical considerations are also trivial, i.e. capitalizing, punctuation, indentation etc. are all optional and user-dependent. However, it is recommended to stay loyal to a unique style throughout the project. For instance, it is good to avoid continuation lines and use indentations. Also it is easier to use lowercase letters for every statement.
- Note that a pseudocode is used to outline what is going to be done in the program. Therefore, the level of details depends on the skills of the programmer, i.e. the more experienced the user, the fewer details required.
- Usually, a pseudocode starts with start or begin and ends with stop or end. The rest of the statements are placed in between with an indentation, e.g. two blank spaces.
- After the start keyword, the declarations are placed in which declaring the variables, constants, and others, takes place.
- Interaction with the user is done by input, read or get and output, write, print or display terms.
- Processes are simply written in plain English using calculate or compute terms. In order to write, for example, a calculation of an expression such as  $x = \sqrt{y} + x$ , it

is possible to write calculate x as square root of y plus x or calculate x=square root of y plus x as the corresponding pseudocode.

- In addition to using the UpperCamelCase format, it is advised to use a couple of parentheses after the name of the subprogram such as GetAdditionalData(). This adds more emphasize on the subprogram and distinguishes from variables.
- Instead of using begin and end, the pseudocode of a subprogram is delimited between the identifier and the return keyword. Also all the code is indented in between.

| Symbol  | Name                         | Function  |
|---|------------------------------|---|
|    | Terminal                     | It is used to mark the start and end of a flowchart.  |
|    | Input/output                 | It is used to get inputs or display outputs.  |
|    | Process                      | It is used to indicate variable/constant declarations, calculations or manipulations of data. Also used for a block of processes. |
|    | Intrinsic subprogram         | A predefined subprogram (serie of processes) within the programming language.   |
|    | Internal/external subprogram | A user defined subprogram (serie of processes).   |
|  | Decision                     | Used to indicate a comparison or decision which leads to different paths.   |
|  | Junction                     | Designates the flow.  |
|  | Off-page connector           | Used to indicate the continuation of the process in the next page.  |
|  | Connections                  | Connects sequential symbols.  |
|  | Comments                     | Used to add more clarification to a symbol.   |
|  | Group                        | Used to indicate a group of several symbols.  |

**Fig. 1.3** Basic symbols used in a flowchart

### 1.2.3 Flowchart Conventions

Similar to pseudocodes, flowcharts are used to illustrate how an algorithm works. Flowcharts are more helpful in terms of visualizing and thus, more convenient for beginners. In contrast, pseudocodes are preferred by professionals because they may be considered one step closer to the real programming codes. However, using both of them for a program or a mix of them in larger programs is possible. A flowchart consists of several rows/columns of block symbols which are related to each other using connections. Basic flowchart symbols with the corresponding descriptions are listed in Fig. 1.3.

## 1.3 Structured Programming

As a programmer, dealing with complicated pieces of code and debugging them gets harder as the size and complexity of the task increases. It seems much easier to make a mess trying to make the code work and get entangled with a so-called *spaghetti code*. Further complications would arise trying to modify the code.

Another case would be trying to understand and update a code written by somebody else. Even if the documentation of the code is quite good, it would be very hard to follow the steps of an *unstructured program*. A similar condition will be encountered when transferring a part of an unstructured code to another program. If not impossible, it could be laborious and troublesome.

An unstructured programming style may be used successfully for small programs but it is definitely not suitable nor recommended for the larger ones. Note that the whole point of the argument is of concern to the user/programmer because the compiler does not care if the program is well-structured or not; the reader does.

The foundation of the program is constructed during the logic planning phase. That is the most important of the phases since it prevents an unstructured program. In addition, keeping to a structured layout will make the proceeding phases of a programming cycle much easier. Debugging, maintaining, updating, documentation, understanding and planning additional parts of the program will be much easier and more straightforward.

Let us elaborate what is considered as a structured program. A structured logic is based on the *divide-and-conquer* concept, namely it breaks down large tasks into one of the three basic structures: a *sequence*, *selection* or *loop*. It is proven that with the help of these basic structures any task can be broken down to understandable pieces—no matter how complicated the task is. Each one of these components can have different variations based on what is required to carry out (see Fig. 1.4).

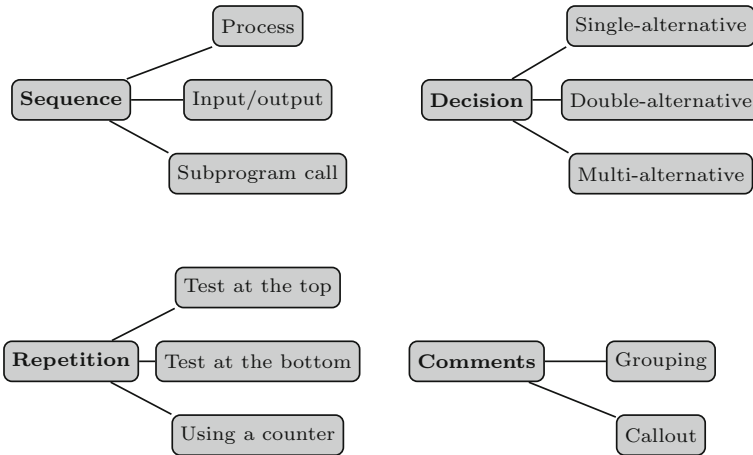
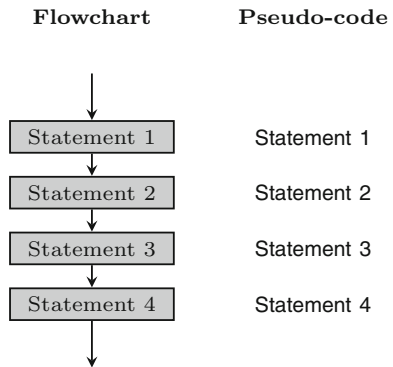


Fig. 1.4 Components of structured programming

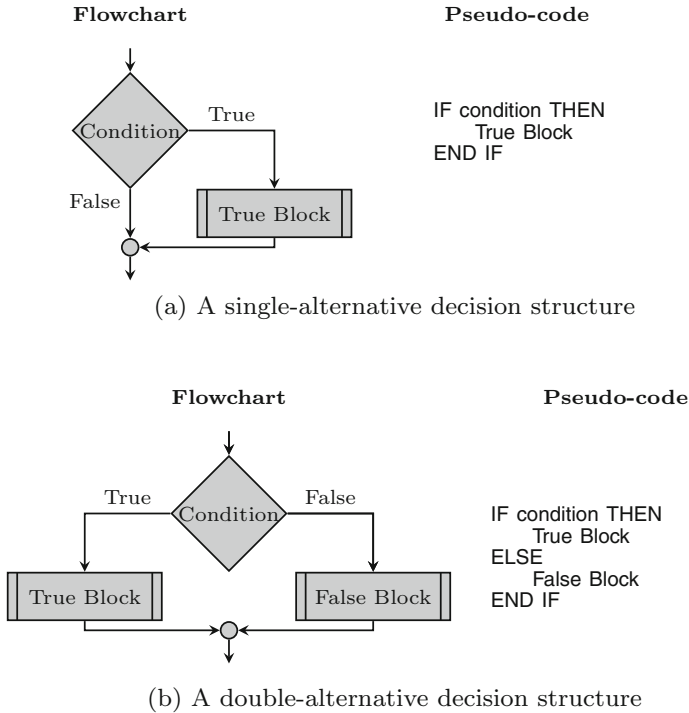
Fig. 1.5 A sequence structure consisting of four statements



### 1.3.1 Sequence, Selection and Repetition

A sequence structure is nothing else than a series of processes which must be done one after another to achieve the intended results. It is analogous to a cooking recipe which must be followed step-by-step to produce a delicious meal. This structure can be shown by either a flowchart or pseudocode (see Fig. 1.5). It can be a combinations of processes, I/O or subprogram calls but no branching is allowed.

Any combination of processes and I/O statements is called a *block*, i.e. a group of related statements which will be executed sequentially. In FORTRAN, a block will be equivalent to a sequence of statements each denoted by a keyword. Schematically, a process symbol is also used for a block of code for a concise demonstration. The FORTRAN equivalent of this structure is almost any statement but a *control structure*. A control structure is used for either a *selection* or *repetition*. Small simple programs usually consist of a sequence of statements.



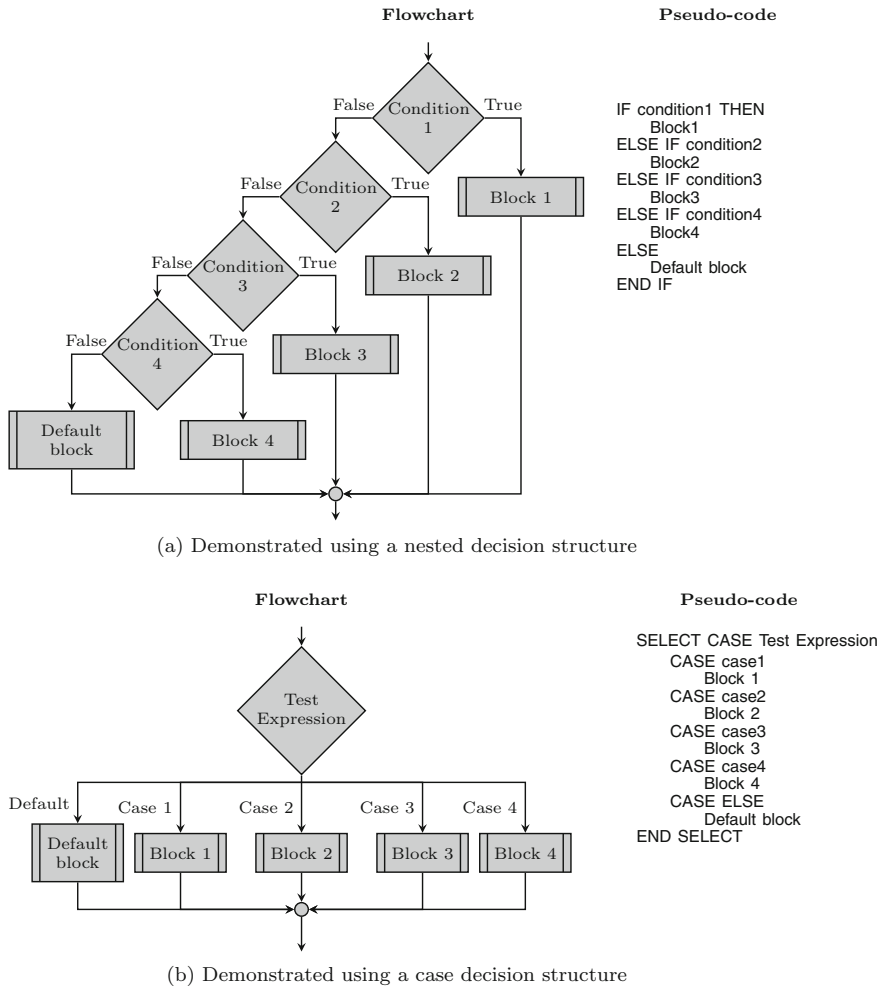
**Fig. 1.6** Simple decision structures

A selection structure, sometimes called a *decision structure*, is a structure which branches based on the result of a Boolean expression. Note that the branches must join each other at the end of the structure. A selection structure can have various forms based on the number of provided alternatives. A one-alternative structure which is equal to a IF-THEN control structure is illustrated in Fig. 1.6a with the corresponding pseudocode. If a two-alternative situation is at-hand then the IF-THEN-ELSE control structure should be used (Fig. 1.6b).

More alternatives are possible, e.g. four alternatives and a default one which will be executed if none of the other alternatives was selected (Fig. 1.7a). This multi-alternative example can be expressed using a CASE construct (Fig. 1.7b) which looks more organized and easier to understand than the previous structure.

The third basic structure is a *repetition, iterative* or *loop* structure which repeats a block of code; called a *loop body* (Fig. 1.8). In this structure, a Boolean expression determines how many times the loop body will be repeated. Based on when this Boolean expression is tested, at the beginning (top) or at the end (bottom) of each loop, different control structures will be used.

The most common type of a repetition structure is a WHILE structure which runs a loop block until a Boolean expression is true and the test is done at the beginning



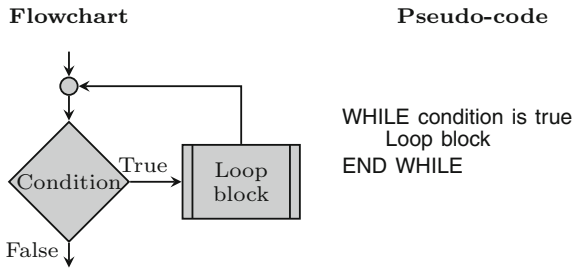
**Fig. 1.7** A multi-alternative decision structure

of each loop. Therefore, if the Boolean expression is false then the loop will not be executed. This type is called a test-at-the-top repetition structure.

In contrast, a test-at-the-bottom structure, a REPEAT UNTIL or DO UNTIL structure, will execute the loop block at least for once without any regards to the value of the boolean expression. Then, the Boolean expression is tested; if it is true, the loop is repeated again.

In all of the mentioned loops, the number of repetitions is not known. Therefore, a condition is tested for the execution of the body of the loop. However, it is also possible to use a *counter* and run the loop body for an explicit number of times using

**Fig. 1.8** A loop structure



a DO construct. This construct is used when the number of repetitions is known a priori.

The important property of every loop structure is the fact that after executing the loop body, the flow of the program must go back to the decision symbol in order to keep the logic structured.

In addition to the three mentioned fundamental structures, optional *comments* are also used for informative purposes although they are not considered as basic structures. The role of comments is usually overlooked despite the fact that they can be very elaborating especially in future referrals to the code. Therefore, using a uniform style of commenting throughout the code is recommended. In terms of flowchart symbols, callouts and curly-brackets are used for commenting and grouping the blocks, respectively.

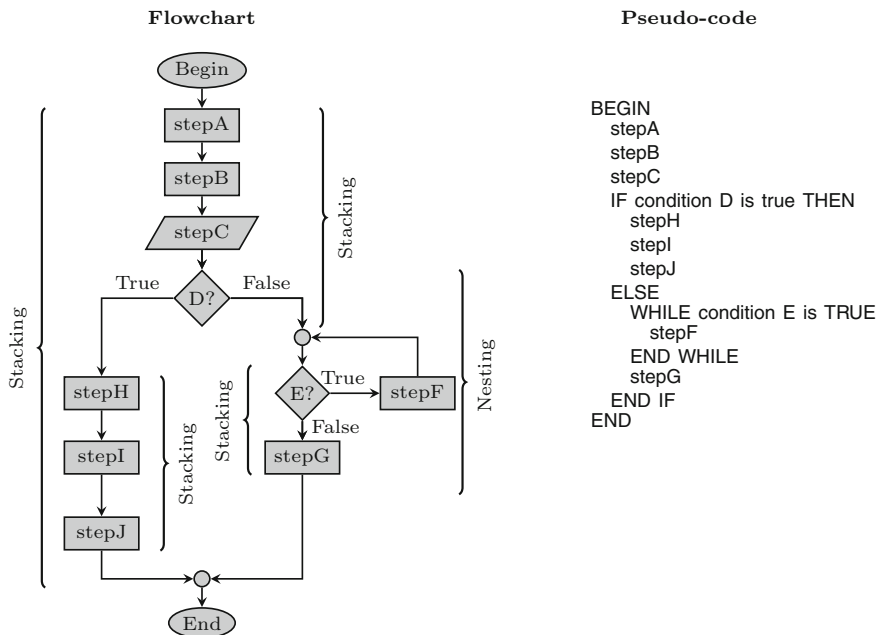
### 1.3.2 Combining Structured Logic

By combining the three basic structures, every problem can be solved. There are two ways of combining structures: *stacking* which is using structures one after another and *nesting* which is placing a structure within another one. With the help of these combination methods, an infinite number of structures can be developed. The multi-alternative decision structure is an example of a nested decision structure within another decision structure (Fig. 1.7a).

Usually nesting and stacking are used together such as the structure in Fig. 1.9. The indentation used in the pseudocode designates the level of the code for instance, three first steps stepA, stepB and stepC are all in the same level with the condition control (D?). Therefore, these four symbols are stacked one after another. The WHILE loop is nested and thus, it is indented. stepG is stacked with the While loop and they are in the same level. Similarly, stepH, stepI and stepJ are stacked together. Using indentation in a pseudocode as well as the code itself, increases the readability of the code and makes the debugging process easier and thus, it is recommended.

Considering what has been indicated earlier, the following characteristics can be summarized:





**Fig. 1.9** Combination of stacking and nesting in a sample structured flowchart and the corresponding vertically aligned pseudocode

- Every structured program consists of sequences, selections, loops and/or any stacked or nested combination of those.
- Every structure has only one entry and one exit point; attaching to other structures is only done by these points. Therefore, jumping from one point of the code to another one by means of statements such as GOTO is prohibited. Also sudden jumps out of a loop and jumps into a loop from outside are prohibited.
- In a loop structure, the control of the program goes back to the decision symbol after executing the loop block. However, in a selection structure one of the two paths is selected. Then, both sides are joined together, not to the decision symbol, and the program continues.

Therefore, in order to convert an unstructured flowchart to a structured one, the program must be checked from the beginning. The basic structures and their combination must be recognized while checking for valid connections, i.e. one-entry, one-exit. If any connection violates any of the mentioned rules, the program must be re-structured. This is usually done by means of additional statements and/or variables. Among the many benefits of a structured logic, *modularization* is the most important benefit which will be discussed later in this chapter.

## 1.4 Control Constructs in Fortran

Simple programs consist of normal execution of processes, i.e. sequential execution. In contrast, sophisticated programs will not just flow in a linear fashion but will tend to have branches and loops. This will cover possible conditions and deal with tedious recurring events.

In FORTRAN, control constructs are used to carry out the selection and repetition structures of the program logic. IF and CASE constructs are dedicated to selection structures whereas DO and DO WHILE are designated to loops. Note that there is no REPEAT UNTIL construct in FORTRAN. However, it can be mimicked using the labels and the GOTO statement. This is not advised because it is against the one-entry one-exit rule of structured programs.

### 1.4.1 IF Construct

Simple decision making is usually done by means of an IF construct. More complicated situations can be simulated using nested IF constructs or compound relational expressions as the condition of the decision. The syntax of the IF construct in FORTRAN is as the following:

```
[if-construct-name:] IF (scalar-logical-expr) THEN
    block
[ELSE IF (scalar-logical-expr) then [if-construct-name]
    block]
[else [if-construct-name]]
    block
END IF [if-construct-name]
```

As mentioned earlier, a block in flowcharts consists of several sequence structures. Similarly in this syntax, a FORTRAN block is also a sequence of FORTRAN statements. The important component of this syntax is the scalar-logical-expr which is a logical expression.

Note that the scalar-logical-expr must always be delimited within parentheses. It can incorporate logical operators such as .AND., .OR., .NOT. etc. to form a compound expression. The use of if-construct-name is optional but it is not very common in programming.

Indentation of the blocks is recommended because the logic will be easier to understand. The pseudocode of the IF construct is quite similar to the code itself.

Consider the following one-alternative structure example:

```
1 IF (flag) THEN
2   WRITE (*,*) 'Flag is flown.'
3 END IF
```

If the value of flag is equal to `.TRUE.`, the message will be shown. *Flags* are logical variables which are used in programming quite often to indicate a special state. For example, a `firstRun` flag within a subprogram indicates its first run. The true-block consists of only one statement, it is also possible to use a more concise syntax:

```
1 IF (flag) WRITE (*,*) 'Flag is flown.'
```

This syntax is not applicable to blocks with multiple statements. If a message is needed for the `.FALSE.` value then the following two-alternative structure can be used:

```
1 IF (flag) THEN
2   WRITE (*,*) 'Flag is flown.'
3 ELSE
4   WRITE (*,*) 'Flag is not flown.'
5 END IF
```

In the condition of the `.FALSE.` value for the flag variable, the `ELSE` block will be executed. A nested `IF` construct may be used to cover multiple cases, such as the following:

```
1 IF (delta .GT. 0.0D0) THEN
2   WRITE (*,*) 'Two distinct roots '
3 ELSE IF (delta .EQ. 0.0D0) THEN
4   WRITE (*,*) 'One double root '
5 ELSE
6   WRITE (*,*) 'No real roots '
7 END IF
```

This example determines the number of roots for a quadratic equation based on its discriminant (`delta`). Sometimes nested structures can be converted to a single `IF` construct with a more complicated logical expression, for instance:

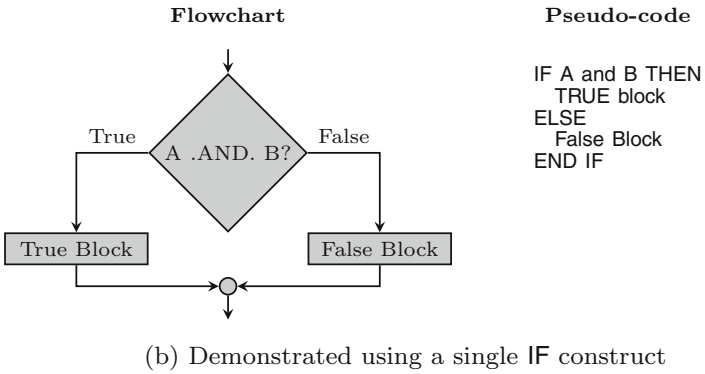
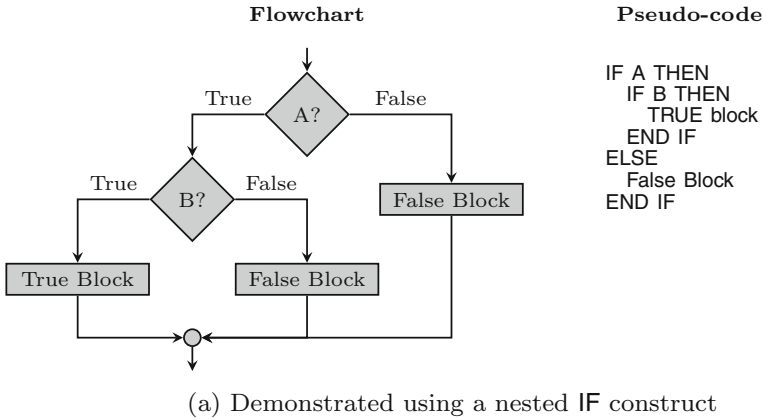
```
1 IF (i .EQ. 0.0D0) THEN
2   IF (j .EQ. 0.0D0) THEN
3     CALL twoZeros()
4   END IF
5 ELSE
6   CALL notTwoZeros()
7 END IF
```

This code can be written with only one `IF` construct and the logic operator `.AND.`:

```
1 IF ((i .EQ. 0.0D0) .AND. (j .EQ. 0.0D0)) THEN
2   CALL twoZeros()
3 ELSE
4   CALL notTwoZeros()
5 END IF
```

This construct is more understandable while carrying out the same decision making process. Note that each individual logical expression is put within parentheses. Generally, `.AND.` and `.OR.` logic can be expressed using nested structures. However, using the compound logical expressions is simpler in terms of programming codes as well as the flowcharts (Figs. 1.10 and 1.11).

It is worth mentioning that comparing a real variable with an exact value is usually not a good practice. It is a rare case to have an exact real number after a calculation. Therefore, it is better to examine the variable to be close to the intended value within a tolerance. For instance, consider the following line:



**Fig. 1.10** .AND. logic relational expression

```

1  IF (aReal .EQ. 1.0) THEN

```

This code tests if the variable aReal is equal to one. However, it is better to be replaced by the following:

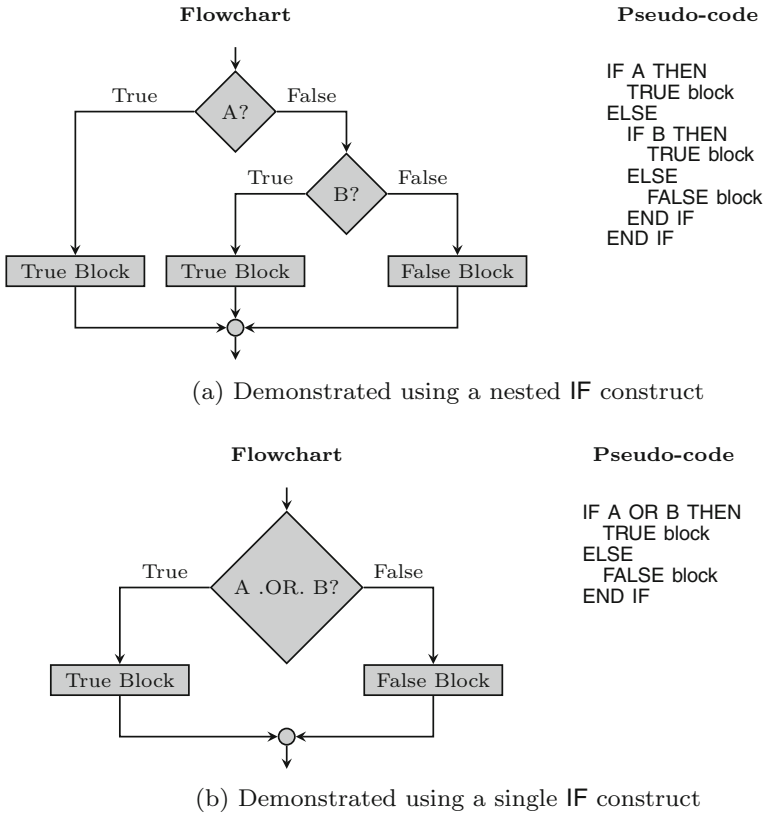
```

1  IF (ABS(aReal - 1.0) .LT. 0.0001) THEN

```

### 1.4.2 CASE Construct

The IF construct is good enough to test several conditions. However, as the number of conditions increases, the nested IF construct might be rather confusing. In such cases, using a CASE construct is advised. In this construct, the value of an expression, called a *case index*, is tested against one of the several values or ranges of values, called *case values*. Unlike the IF construct, comparison of ranges is possible and



**Fig. 1.11** .OR. logic relational expression

thus, the type of the case index and its case-values for a CASE construct is limited to discrete types. Namely, they can be either an integer, a character or a logical type. The syntax of the CASE construct is as the following:

```

[case-construct-name:] SELECT CASE (case-expression)
[CASE case-selector [case-construct-name]
  block] ...
END SELECT [case-construct-name]
                    
```

In this syntax, case-construct-name is an optional name for the construct, case-expression is the case index and case-selector is either a case value or the DEFAULT keyword. This keyword indicates the default block if none of the other case values match the case index.

The case values must be selected in a manner so that no overlapping happens. Namely, no case index can be assigned to multiple case values. However, it is possible for a case value to cover a range of not overlapping values as demonstrated in

**Table 1.10** Possible formats for case value ranges in a CASE construct

| Case value range        | Equivalent logical expression            |
|-------------------------|--|
| case-value              | case-index == case-value                 |
| case-value              | case-index <= case-value                 |
| case-value:             | case-index >= case-value                 |
| case-value1:case-value2 | case-value1 <= case-index <= case-value2 |

Table 1.10. Note that case values must be delimited in parentheses and can cover multiple non-overlapping cases. For example, CASE (1:10, -2, :-10) covers integer numbers 1–10, -2 and all the integers less than or equal to -10.

As an example, consider again the example of finding the number of the roots for a quadratic equation based on its discriminant (*delta*). This was carried out earlier using a nested IF structure. Because *delta* is a REAL type variable and not a discrete value, it cannot be used directly as a case index. However, the sign of *delta* is a discrete value and thus, it can be used.

Two intrinsic functions are needed: *Int* (A) which truncates the real value of A and returns an integer; *Sign* (A, B) which returns +|A| if B is positive and -|A| otherwise. Note that A, B and the function return value are of the same type.

Similarly, in our example the *delta* variable is of the real type and thus, the return value is a real number. Therefore, the *Int* intrinsic function must be used to convert the result to an integer value. The equivalent CASE construct is as the following:

```

1  SELECT CASE (Int (Sign (delta , delta)))
2  CASE (+1:)
3    WRITE (*,*) 'Two distinct roots '
4  CASE (0)
5    WRITE (*,*) 'One double root '
6  CASE (:-1)
7    WRITE (*,*) 'No real roots '
8  END SELECT

```

Considering this example, it can be seen that sometimes using a CASE structure is not in favor of simplicity. This example can be modified using a DEFAULT case, to be less specific and just state if any roots exist or not:

```

1  SELECT CASE (Int (Sign (delta , delta)))
2  CASE (0:)
3    WRITE (*,*) 'Real Root(s) '
4  CASE DEFAULT
5    WRITE (*,*) 'No real roots '
6  END SELECT

```

More complicated case recognitions can be developed using nested CASE constructs with another one or even with an IF construct to take advantage of its capabilities.

### 1.4.3 DO Construct

A DO construct is designed to execute a block for a number of times. The syntax of this construct is as the following:

```
[do-construct-name:] DO [label] [loop-control]
  block
[label] END DO [do-construct-name]
```

This syntax indicates that the label and do-construct-name are optional. Although using both is possible, it is not advised. Three different types of loop-control can be selected for a DO construct. All of them have the following general syntax:

```
[.] do-variable = scalar-integer-expression1, scalar-integer-expression2 [, scalar-integer-expression3]
[.] WHILE (scalar-logical-expression)
```

The first form is called the DO construct with an iteration count. In this form, a counter is used to control the number of iterations. The counter or the do-variable in the syntax is merely an integer. The first, second and third scalar-integer-expression are the starting, finish and step (increment) values for the counter, respectively. In other words, after assigning the first integer value to the counter the block runs for the first time and then the step value is added and the block runs again. This loop continues until reaching the finish value and then, the program will execute the first statement after the END DO statement.

The default value for the step is 1 and hence, the finish value must be greater than or equal to the starting value. Generally, any non-zero value is permitted for the step value which makes running the block possible either by increasing or decreasing the counter. The number of iterations is equal to than the  $\text{MAX}((\text{finish}-\text{start}+\text{step})/\text{step},0)$ . Therefore, for positive step values if the start value is greater than finish value, zero iterations will result. For the negative step values it is the other way around. A simple example is summing up all the values of an array named myData with n elements:

```
1      sum = 0.D0
2      DO i = 1, n
3          sum = sum + myData(i)
4      END DO
```

If the array is two-dimensional with n1 and n2 ranks, a nested loop must be used to sum all the elements:

```
1      sum = 0.D0
2      DO i = 1, n1
3          DO j = 1, n2
4              sum = sum + myData(i, j)
5          END DO
6      END DO
```

Another example is printing the even numbers between 100 and 1 in a decreasing manner:

```

1      DO i = 100, 1, -2
2          Write (*,*) i
3      END DO

```

The second form of a DO construct is with a WHILE control. In this form, the block will be executed while a condition is true. The evaluation of the condition expression is done prior to executing the block and hence if the result is false then the block will not be executed. This form is a test-at-the-top form of loops. It is possible to convert from the first to the second form and vice versa. For instance, the following code can be used to sum up the elements of an array:

```

1      sum = 0.D0
2      i = 1
3      DO WHILE (i <= n)
4          sum = sum + myData(i)
5          i = i + 1
6      END DO

```

In this example, initializing the counter *i* and updating its values by incrementing is done manually. In contrast, in the iteration form of the DO construct, these two tasks were carried out automatically.

The third form of the DO construct is an obsolescent one with a non-block structure. This form was the only loop construct before FORTRAN 90 and thus, it is encountered frequently in older code. The other two forms were introduced afterwards and have covered all the programming demands. However, the third form is still supported by the current compilers for compatibility issues. The non-block DO has the following form:

```

[DO label [loop-control]]
  [execution-part-construct]
label action-statement

```

Every statement between the DO statement and the label will be repeated (including the label itself). Note that in the syntax, branching or DO altering constructs are not allowed as an action-statement. The following example prints the even numbers between 100 and 1 using this form:

```

1      DO 100 i = 100, 1, -2
2      100 Write (*,*) i

```

Although this is completely right, as a convention the CONTINUE statement is used just as a place-holder for the label. Namely, it does nothing. Rewriting the same code will result in the following:

```

1      DO 100 i = 100, 1, -2
2          Write (*,*) i
3      100 CONTINUE

```

Nested loops are also possible in this approach. For instance, to sum up the two-dimensional array *myData* we will have:

```

1      sum = 0.D0
2      DO 100 i = 1, n1
3          DO 200 j = 1, n2
4              sum = sum + myData(i, j)
5      200 CONTINUE
6      100 CONTINUE

```



It is evident that the third form can readily be converted to other forms. Therefore, there is no critical need to use this outdated form.

### 1.4.4 REPEAT UNTIL

Out of the three repetition constructs used in a structured logic, the test-at-the-bottom structure, i.e. a DO UNTIL or a REPEAT UNTIL loop, is not formally supported by FORTRAN. However, it can be mimicked using an IF construct in junction with a GOTO statement, if necessary. The following is a not-advised example which prints a line for 10 times:

```

1      counter = 10
2      100 CONTINUE
3         Write (*,*) 'simulating...', counter
4         counter = counter - 1
5         IF (counter >= 1) GOTO 100

```

Again, using a GOTO statement violates the structured logic and it is strongly recommended to be avoided.

### 1.4.5 Altering the DO Construct

There are two special statements which can alter the execution process within a DO construct: EXIT which immediately terminates the construct and CYCLE which terminates the current increment and jumps to the next one. Although these are supported by the language, it is strongly advised to avoid using them because they violate the structured logic of the program. Alternatively, by using a better design it would be possible to avoid incorporating these statements into the construct. For instance, it is possible to create an infinite loop by choosing a relational expression which is always true, e.g. (.TRUE.). In such a case, the control of the repetitions is done by the EXIT and CYCLE statements. This approach not only violates the structured program logic but also adds unnecessary complications to the program. In addition, it makes the understanding and debugging of the program troublesome. It is recommended to avoid such constructs and try to rearrange the logic to a structured form. In this context, a flowchart will come in handy to untangle such a structure.

### 1.4.6 Branching

The act of transferring the execution process from the point of a *branch statement* to the point of a *branch target statement* is called branching. It is usually done by means of a GOTO statement targeting a labeled statement. The targeted label usually

holds a trivial CONTINUE. Although it is not permitted to jump to a statement within a block from outside of the block, it is possible to jump from the inside of a block to the outside. However, using the GOTO statement is against structured logic and it is not advised. This is significant to the extent that a structured program is sometimes called a *GOTO-less* program.

In this context, a CONTINUE statement may be used to indicate a non-block DO construct. If a GOTO statement aims for this CONTINUE statement, it will act the same way as a CYCLE statement. In other words, jumping to the CONTINUE statement of a DO construct is the equivalent of starting the next cycle of the loop. However, in general, any other CONTINUE statements would act just as a place-holder for the label. Note that the jump is made within the DO construct. To elaborate on this, consider the following lines of code:

```

1      DO 100 i = 1,10
2          Write (*,*) i
3      GOTO 100
4          Write (*,*) 'I am here!'
5 100   Continue
6 200   Continue

```

In this example, the ‘I am here!’ sentence will never be printed. Because the GOTO 100 actually moves the control of the program to line 5 which is trivial and finally to line 1 to start the next cycle of the loop. If in the same code, a GOTO 200 statement is placed before the DO construct, simply the construct will not be executed. Note that branching from the outside to anywhere inside of a non-block DO construct (including its CONTINUE statement) is prohibited. For instance, the following code will not run:

```

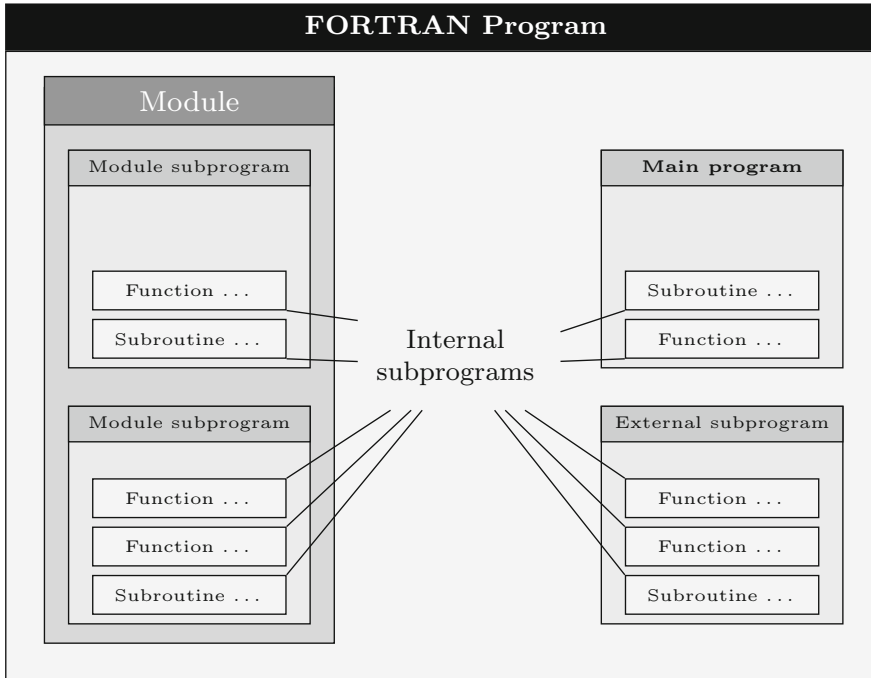
1      GOTO 100
2      DO 100 i = 1,10
3          Write (*,*) i
4      GOTO 100
5          Write (*,*) 'I am here!'
6 100   Continue

```

Another branching statement is the STOP statement which terminates the program immediately. The STOP statement violates the one-entry one-exit rule of the structured logic. Therefore, it is advised not to be used.

## 1.5 Procedural/Modular Programming

Two general ways of implementing the structured programming method are *procedural programming* and *modular programming*. In both methods, the idea of breaking a large problem into smaller tasks is considered and a main program is responsible for the coherence of the subtasks. In procedural programming, a single file is used containing a main program plus several subprograms. In this method, the main program is responsible of calling other subprograms located within the same file. In contrast, modular programming incorporates several independent programs, called *modules*,



**Fig. 1.12** Structure of a sample FORTRAN program

to perform the job. Although both of these methods are faithful to incorporate sub-tasks, the modular programming approach has the advantage of the independence of the modules, viz. modules can be run, tested and updated separately from the main program.

In FORTRAN, both of the mentioned methods can be used. Procedural programming is carried out by means of *internal subprograms*, i.e. internal functions and/or subroutines whereas the modular concept can be incorporated using *modules* (Fig. 1.12). Normally a program is developed using a mixed-mode (procedural/modular approach). Namely, each of the mentioned methods is used to some extent. However, it is recommended to shift the approach from procedural to modular as much as possible.

The main idea behind the modular approach is to divide a task into independent sub-tasks. This results in more manageable and understandable pieces of code which makes the programming easier in the current and following references. In a modular system, instead of dealing with a program in whole, several discrete *subprograms* or *procedures* will carry out the same purpose but in a well-structured fashion. The basis of this system is in accordance with the structured logic mentioned in earlier sections and that is why the phrase *structured programming* is sometimes used for the same meaning. A modular design for a problem affects the program development cycle in every phase as described below:

- better understanding of the idea behind the whole program,
- easier maintenance and application of modifications to the code,
- improved testing due to the fact that individual testing of the sub-tasks is possible,
- easier debugging and detection of errors,
- achieving a better logic design with less time and effort,
- easier changing of the updated parts with regard to the whole program,
- easier team-work because parallel working is possible due to breaking up the whole project to sub-task, and
- improved portability of the code to other programs and re-usability of the same pieces of code in the future projects.

In FORTRAN, a *program* consists of *program-units* which may be written in a language other than FORTRAN. A program-unit is composed of several constructs and statements. Based on the heading statement of the unit, it can be either a *main program*, a *module*, an *external subprogram* or a *block data*. In other words, a program unit defines a data environment and performs some calculations with this data.

Every FORTRAN program consists of one *main program* and in accordance to the procedural/modular programming concept, it will call other subprograms. A subprogram can be either a *function subprogram* or a *subroutine subprogram*. A subprogram executes one or more procedures that is why it is sometimes called a *procedure* or a *method* despite the fact that a subprogram executes a procedure and it is not a procedure itself. In any case, both of the terms are used in programming context but in the current book the subprogram term is preferred.

Additionally, the terms ‘subroutine’ and ‘function’ are used in short for ‘subroutine subprogram’ and ‘function subprogram’, respectively. Subprograms and the main program are executables whereas modules and block data are not. In simple words, a module is used for sharing procedures and/or data while a block data is used to initialize the shared data.

When a subprogram is a program unit itself then it is an *external subprogram* or *external procedure*. If it is contained in another program unit, it is called an *internal subprogram* or *internal procedure*. In other words, external subprograms can have internal subprograms within themselves but further nesting for internal subprograms is not allowed. Namely, internal subprograms cannot contain any other subprograms. If a subprogram is defined within a module, it is called a *module subprogram* or *module procedure*.

To elaborate more on these concepts consider Fig. 1.13 which illustrates a program consisting of the following program units:

- Main, the main program,
- ExternalSub, an external subprogram,
- MyModule a module,
- ModSub1 and ModSub2 are module subprograms,
- InternalSub1 and InternalSub2 are internal subprograms of the main program,
- InternalSub3 is an internal subprogram of the external subprogram, and
- ModInSub1 and ModInSub2 are internal subprograms of the module subprograms.

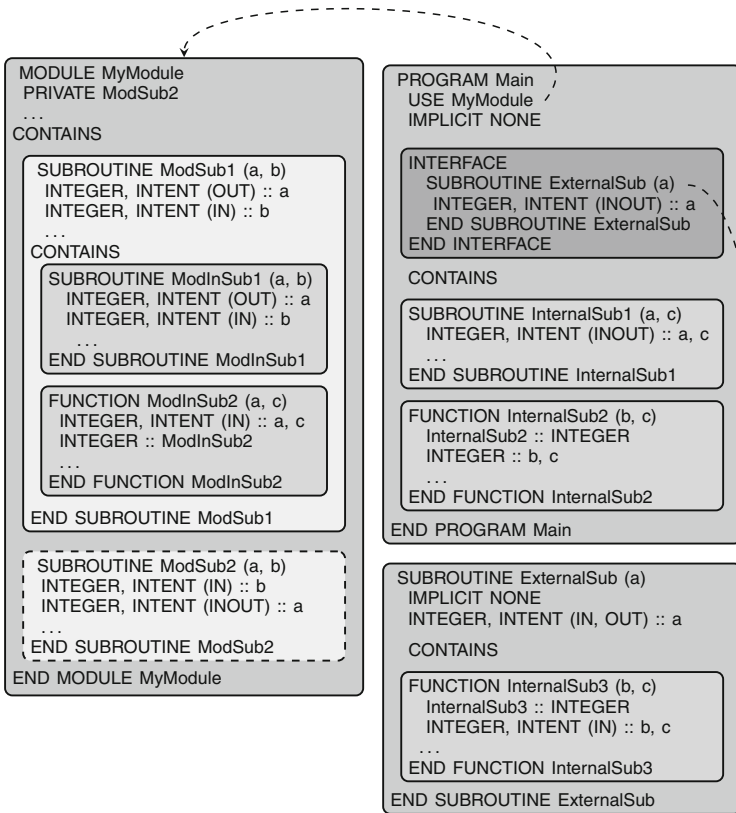


Fig. 1.13 Detailed sample arrangement of program units in a Fortran program

The access to the external subprogram is provided by means of an interface (Sect. 1.8.8) in the main program and the access to the module is made available through the USE statement (Sect. 1.6.1). In addition, the arguments are fully declared using the INTENT attribute (Sect. 1.8.5) and the access to ModSub2 module subprogram is declared as PRIVATE (Sect. 1.8.2).

### 1.5.1 Structure of Program Units

The following syntax can be used for every program unit, i.e. a main program, a module, a subprogram and a block data, provided that some minor modifications are applied:

```

[heading]
[specification-part]
  
```

```

    [execution-part]
  [CONTAINS
    internal-subprogram
    [internal-subprogram] ...]
  [ending]

```

The required modifications are as the following:

- The heading (heading) and the corresponding ending (ending) must be altered based on what type of program unit is being used. In other words, the heading defines the type of program unit.
- In the specification part (specification-part), the data environment will be set up.
- An execution part (execution-part) contains the procedure of the programming logic. This part only exists in executable program units, i.e. the main program and subprograms. This part is prohibited for the non-executable programming units, i.e. modules and block data.
- An internal subprogram (internal-subprogram) is prohibited for a block data and for an internal subprogram itself.

The execution of a typical program starts from the main program with a syntax such as the following:

```

  [PROGRAM program-name]
    [specification-part]
    [execution-part]
  [CONTAINS
    internal-subprogram
    [internal-subprogram] ...]
  [END [PROGRAM [program-name]]]

```

Note that the heading for a main program is optional and in MARC/MENTAT using a main program is not allowed.

## 1.5.2 Subprograms

Essentially, a procedure can be done by a subprogram which is either a function or a subroutine. Aside from the syntactic differences, a function and a subroutine are similar in many aspects. However, each has slightly different characteristics.

Generally, a subroutine interacts with its arguments and the result of its procedure is reflected by modification of one or more of its arguments. Hence, a subroutine may return several values via modifying its arguments or no values at all. Dealing with several inputs and outputs is usually an indicator of a complicated procedure and a subroutine is a good choice to perform such a job.

In contrast to the subroutines, the result of a procedure done by a function is normally a single value. It is called the *result of function*. The result is returned not

by the arguments but by the function itself acting as a variable. Because the name of a function is a variable, it has the advantage of being referenced in expressions while subroutines are invoked by a CALL statement. The procedure of a function is normally simpler than that of a subroutine. In other words, the purpose of a function is just calculating the result value while avoiding any manipulation of its arguments; called having minimum *side effects*.

From another point of view, subroutines are either external or internal. Although both internal and external subprograms are used in a procedure-based approach, there are a few subtle distinctions:

- An internal subprogram is a local entity and can be accessed only within the program or subprogram in which it is defined whereas an external subprogram is global (more on the scope in Sect. 1.6.4).
- An external subprogram (or a module) may contain internal subprograms but an internal subprogram cannot contain another internal one.
- The interface of an external subprogram is not known to the referencing program/-subprogram but for an internal subprogram the interface is known and therefore, an explicit interface is not required (more on interfaces in Sect. 1.8.8).
- An external subprogram (or a module) is compiled separately and thus, producing separate files with a .mod extension. On the contrary, an internal subprogram is compiled within its host (more on hosts in Sect. 1.6.4).
- The name of an internal subprogram has precedence over the external ones and even over intrinsic subprograms.

The syntax of an external subroutine is as the following:

```
SUBROUTINE subroutine-name [[(dummy-argument-list)]
    [specification-part]
    [execution-part]
[CONTAINS
    internal-subprogram
    [internal-subprogram] ...]
END [SUBROUTINE [subroutine-name]]
```

Compared to a program unit, only the header and end statement are different. The header now contains a list of arguments needed for the subroutine (dummy-argument-list) and the end statement contains the SUBPROGRAM keyword. The CONTAINS keyword indicates the internal subprograms of the subroutine. As mentioned earlier, an internal subprogram cannot contain other internal subprograms. Therefore, the syntax for an internal subprogram will be the same as an external one but without the CONTAINS part:

```
SUBROUTINE subroutine-name [[(dummy-argument-list)]
    [specification-part]
    [execution-part]
END [SUBROUTINE [subroutine-name]]
```

The following are some examples for subroutine headers and the corresponding end statements:

```

1      SUBROUTINE PrintWelcomeMessage
2          ...
3      END SUBROUTINE PrintWelcomeMessage
4
5      SUBROUTINE PrintMessage ()
6          ...
7      END SUBROUTINE PrintMessage
8
9      SUBROUTINE SortArray (anArray)
10         ...
11     END SUBROUTINE SortArray
12
13     SUBROUTINE ReadFile (fileName , readData)
14         ...
15     END SUBROUTINE ReadFile

```

The two first subroutines, i.e. `PrintWelcomeMessage` and `PrintMessage`, do not have any arguments whereas the next two subroutines have one and two arguments, respectively: the `anArray` argument for the `SortArray` subroutine and the `fileName` and the `readData` arguments for the `ReadFile` subroutine. The declaration of these arguments will be defined in the specification part of the subroutine. The specification and execution parts of a subprogram are similar to those of a main program only that the dummy argument list must be defined in the specification part as well. The subprogram header and the data declaration of the arguments are called the *signature* of the subprogram. For instance, the signature of the subprograms of the previous example can be as the following:

```

1      SUBROUTINE SortArray (anArray)
2          REAL, DIMENSION (10) :: anArray
3          ...
4      END SUBROUTINE SortArray
5
6      SUBROUTINE ReadFile (fileName , readData)
7          CHARACTER (LEN=32) :: fileName
8          REAL :: readData
9          ...
10     END SUBROUTINE ReadFile

```

In this example, the subroutine `SortArray` accepts an array of ten real numbers named as `anArray`. The `ReadFile` subroutine deals with two variables: a string of 32 characters named `fileName` and a real number named `readData`. Because an argument of a subroutine may be used as an input, output or both. Therefore, a better way of declaring an argument is using it with an `INTENT` attribute to sort out the intention of the argument (more details are available in Sect. 1.8.5). As another example, in the following code `Outside` is an external subroutine which is the host of the `Insider` subroutine:

```

1      SUBROUTINE Outside
2          ...
3      CONTAINS
4          SUBROUTINE Insider
5              ...
6          END SUBROUTINE Insider
7      END SUBROUTINE Outside

```



The syntax of an external function is as the following:

```

FUNCTION function-name ([[dummy-argument-list]])
  [ specification-part ]
  [ execution-part ]
[ CONTAINS
  internal-subprogram
  [ internal-subprogram ] ... ]
END [FUNCTION [function-name]]

```

The syntax of an external function is very similar to that of an external subroutine: similarly, not only it is required to declare the arguments of a function but additionally, the type of the function itself must be declared. An internal function has the same syntax but without the internal subprogram part:

```

FUNCTION function-name ([[dummy-argument-list]])
  [ specification-part ]
  [ execution-part ]
END [FUNCTION [function-name]]

```

In the following example, a function named `AddNumbers` with two real arguments named `a` and `b` is declared which returns a real value as the result:

```

1  FUNCTION AddNumbers (a, b)
2     REAL :: a, b, AddNumbers
3     ...
4     END FUNCTION AddNumbers

```

### 1.5.3 Procedure Referencing and Arguments

A *procedure reference* occurs when the name of a procedure is used in a piece of code in order to execute it. The terms *invoking* or *calling* are also used in the same meaning for both functions and subroutines. When a subprogram is referenced, the execution of the current program is suspended until after the execution of the subprogram is completed. Then, the execution of the initial program continues.

Referencing can be done for both functions and subroutines; called *function reference* and *subroutine reference*, respectively. Referencing to a subroutine is done using a `Call` statement and to a function is done when the function is used in an expression. User-defined programming units can be accessed with proper referencing.

In contrast to the user defined procedures, there are procedures provided by FORTRAN itself which are called *intrinsically defined procedures* or *intrinsic procedures*. They can be categorized in one of the following items:

- *standard intrinsic procedures*, defined by the standard, e.g. `Sign ()`, `Sqrt ()`, `Abs ()` etc.
- procedures in *standard intrinsic modules* such as C interoperability module, and
- *non-standard procedures* made available by a specific compiler.

Using non-standard procedures will make your code not portable to other compilers and hence, is not advised. Replacing the compiler-dependent procedures with one's own procedures is a better idea to resolve the problem of portability.

In a subprogram reference, some entities are passed to the subprogram to take part in the calculation procedure. This entity is an *argument* to the subprogram. From the perspective of the subprogram, it receives the value of the entity as a *parameter*. These two terms are usually used instead of each other [13].

The argument which is declared in the header of the subprogram is called a *dummy argument* because it is not actually a variable and does not occupy any memory. However, the entity which is passed as this dummy argument is actually a data entity which is called an *actual argument*.

An actual argument for a function is usually used only as an input whereas it may be used as an input, an output or both in a subroutine. The `INTENT` attribute is used to indicate this (see Sect. 1.8.5).

A subprogram takes control of the execution flow upon being invoked. In other words, execution of the program transfers to the first action statement within the subprogram in which the actual arguments are now known as dummy arguments. This process is called *argument association*. From now on, accessing the actual arguments is done through the dummy arguments; the latter serves as a link to the former.

A `CALL` statement for invoking a subroutine has the following syntax:

```
CALL subroutine-name [([[keyword=]actual-argument]])]
```

In the syntax, an actual argument can be either a variable in the simplest form or an expression which is being passed to the subroutine. The keyword here is the name of the dummy argument which is indicated in the header of the subroutine. Using this keyword is optional but when the sequence of arguments is not considered, it must be used to avoid any confusion. For instance in the following example, `sum`, `myInt` and `2*12` are actual arguments for a subroutine reference to `SumUp` and they are associated to dummy arguments `a`, `b` and `c` within the subroutine, respectively:

```

1  PROGRAM Main
2  INTEGER :: sum, myInt
3
4  myInt = 10
5  Sum = 0
6  CALL SumUp (sum, myInt, 2*12)
7  END PROGRAM Main
8
9  SUBROUTINE SumUp (a, b, c)
10 INTEGER, INTENT (IN) :: b, c
11 INTEGER, INTENT (OUT) :: a
12
13 a = b + c
14 END SUBROUTINE SumUp
```

In addition, referencing to the previously declared sample subroutines in Sect. 1.5.2, can be done as in the following example:

```

1      CALL PrintWelcomeMessage
2      CALL PrintMessage ( )
3      CALL SortArray (2 * myArray1 + myArray2)
4      CALL ReadFile ('data.txt', myText)
5      CALL ReadFile (readDATA = myText, fileName = 'data.txt')
```

In the first two lines of this listing, the `PrintWelcomeMessage` and the `PrintMessage` subroutines have no arguments. The argument for the third subroutine is the `2 * myArray1 + myArray2` expression and for the fourth one the arguments are a named constant `'data.txt'` and a variable named `myText`. In the last line, another reference is made to the `ReadFile` subroutine but the sequence of the arguments are not considered. Therefore, using the keywords of the subroutine is mandatory, i.e. `readData` and `fileName`.

### 1.5.4 Modules

As mentioned earlier, two non-executable program units are data blocks and modules. Data blocks will be discussed in conjunction with common blocks in Sect. 1.8.3. Data and common blocks are constructs used for sharing data whereas a module manifests the concept of modularization which allows the user to not only share data entities between program units but also share procedures as well. In other words, a module is a collection of encapsulated data entities and subprograms with the ability of hiding unnecessary entities from the user. This is called *encapsulation* which provides a more professional and independent package. The syntax of a module is as the following:

```

MODULE module-name
  [ specification-part ]
  [ CONTAINS
    module-subprogram
    [ module-subprogram ] ... ]
END [MODULE [ module-name ]]
```

The specification part of a module is similar to that of an external subprogram except that no execution part exists for a module. In addition, the access to the items within a module can be controlled using the `PUBLIC` and the `PRIVATE` attributes (described in Sect. 1.8.2). The entities with the `PUBLIC` attribute can be accessed from outside of the module. This is the default attribute for any module entity.

Every module is compiled to a file with a `.mod` extension and linked later to the subprograms in which it is needed. In order to use a module in any subprogram or even in another module, the `USE` statement must be used at the beginning of that program unit (described in Sect. 1.6.1). Note that a `USE` statement can be used within a module to use another module but a module cannot refer to itself.

Each program unit may have several internal subroutines. Every variable in the program unit is considered global to all its internal subprograms.

The SAVE attribute is used to save the value of a variable in a module. This feature is similar to what common blocks do. But it is a better way of transferring data between subprograms (Sect. 1.8.3). The module subprogram part (module-subprogram) is similar to the internal subprograms of other program units.

The following example is a module which works as a common block just to share data between program units:

```

1  MODULE CommonData
2      CHARACTER (LEN=30) :: MESSAGE = 'Welcome...'
3      REAL :: lengthSum
4      INTEGER, SAVE :: runCount
5      TYPE Geometric
6          REAL :: Area, Priemeter, sideA, baseB, sideC
7      END TYPE Geometric
8  END MODULE CommonData

```

In this example only the specification part of a module is used to share a constant named MESSAGE and two variables: a real LengthSum and an integer runCount with SAVE attribute. A user defined type named Geometric is declared and shared in the module. It is common practice that a derived data type is packaged with related subprograms in a module. A more complicated example will package subprograms as well:

```

1  MODULE MyModule
2      TYPE Geometric
3          REAL :: Area, Priemeter, sideA, baseB, sideC
4      END TYPE Geometric
5  CONTAINS
6      SUBROUTINE CalcArea (geoEntity)
7          TYPE (Geometric), INTENT (INOUT) :: geoEntity
8          ...
9      END SUBROUTINE CalcLength
10 END MODULE MyModule

```

It is good practice to put each module in a separate file and import them in the program using the INCLUDE statement (more details in Sect. 1.8.3).

## 1.6 Specification Part

A specification part (implicit-part) consists of several specification statements by which a data environment is prepared, e.g. types, attributes and some initialization of variables, introduction of named constants, type declarations, data statements, and others, are placed in this part. The specification part of program units is slightly different in terms of the type of that program unit. However, the general syntax of the specification part is as follows:

```

[use-statement] ...
[implicit-part]
[declaration-construct] ...

```

### 1.6.1 USE Statement

In addition to the reference types mentioned in Sect. 1.5.3, *module reference* is referencing a module by means of a USE statement. The first statement of the specification part is used for module referencing which grants a program unit the access to the public entities of other modules. This access can be either a complete access to all the public entities or a selective one, by using the ONLY option. Based on the approach, one of the following simplified syntaxes will be used:

```
USE module-name [, rename-list]
USE module-name, ONLY: [only-list]
```

If all of the public entities are used with the first syntax then a rename list (*rename-list*) might be needed to avoid any name conflicts. This conflict can be among the local entities or those of the other modules and the module in use. The renaming facility can also be used just for adapting to the current naming convention of the program. A rename list or an only list (*only-list*) is generated using the following syntax:

```
[local-name =>] module-entity-name
```

Using this syntax for a rename list, a local name (*local-name*) is selected for a module entity name (*module-entity-name*). The same syntax can be used for an only list and the difference will be that only the mentioned entities can be accessed. Consider the following examples:

```
1  USE AModule
2  USE MyLibrary , myList => sorted_data
3  USE MyLibrary , ONLY : SortArray => quick_array_sort
```

The first USE statement gives access to all the public entities of AModule and the second one, uses all entities of MyLibrary but renames *sorted\_data* to *myList*. The last USE statement gives access only to the *quick\_array\_sort* entity and renames it to *SortArray* at the same time.

### 1.6.2 IMPLICIT Declaration

A simple implicit part (*implicit-part*) consists of an IMPLICIT statement but its general syntax is as the following:

```
IMPLICIT implicit-spec-list
```

The type of a named data object can be declared *explicitly* by a *type declaration statement* in a declaration construct (*declaration-construct*) or *implicitly* using the IMPLICIT statement in the implicit part. In the latter case, the type will be assigned to the data object based on the first letter of its name and when used, it applies to every named variable and named constant. If no implicit statements exist then the default implicit mapping will be used, that is every entity starting with one of the letters I,

J, K, L, M or N is a default integer type and entities starting with all other letters are considered of default real type.

Although it is possible to change this default setting, the approach itself is dangerous because any careless mistypes will be considered as a new variable based on its first letter. This causes the program to run without any warnings. To avoid the complications of debugging such programs, the best way is to declare the types explicitly and disable the implicit typing facility by replacing `implicit-spec-list` with the `NONE` keyword:

```
1 IMPLICIT NONE
```

By this way, the compiler will detect and report any mistyped name as a name without a type and will ask for the explicit type declaration of that data entity. This statement must be used before any `PARAMETER` attributes and no other implicit statements are allowed to be used after that.

### 1.6.3 Declaration Construct

The declaration construct (declaration-construct) is the part in which, explicit declarations of types is carried out whereas for an implicit approach, an implicit declaration is done in the implicit part. In the declaration part of a program unit, the following items can be declared:

- derived type definitions (Sect. 1.7.7),
- interface block (Sect. 1.8.8),
- type declaration statements (Sect. 1.7),
- specification statements (Sect. 1.8).

First of all, derived types are defined because they may be used in the following interface using the `IMPORT` statement. Afterwards, variables are declared using type-declaration statements and the specific attributes are added using specification statements. There are two main approaches to declare an entity equipped with attributes:

- the first approach is the most concise one: it uses the attributes together with the type declaration statement, called an *entity-oriented* declaration. In this way, the focus is on the entity itself and the combination of attributes are lined one after another, making the whole construct easy to understand. The syntax of this approach is as the following:

```
declaration-type-spec [, attribute-spec] ...::] entity-declaration-list
```

The declaration type specification (`declaration-type-spec`) is the type of the entity and the attribute specification (`attribute-spec`) is the corresponding attribute.

- In the second approach, initially the type is declared and then, the attribute statements are added in separate statements called *specification statements*. The approach is called an *attribute-oriented* declaration. It is inherited from earlier versions of FORTRAN and emphasizes attributes because separate lines of code are

dedicated to them. The syntax of this approach depends on the used attribute but generally is shown as:

specification-statement

Although a mixture of these two approaches can be used, the entity-oriented one is preferred in this book. Note that no matter which approach is selected, each attribute must be assigned just once.

In conclusion, a simple declaration construct has the following syntax:

```
declaration-type-spec [ [, attribute-spec] ...:] entity-declaration-list
specification-statement
derived type-definition
interface-block
```

This syntax consists of a type declaration with optional attributes and/or specification statements plus derived type declarations (derived type-definition) and an interface block (interface-block).

### 1.6.4 Association and Scope

*Association*, *scope* and *definition* are interrelated concepts which come up while dealing with entities within nested program units. As mentioned earlier, a program unit may contain several other program units. Each one of them with multiple constructs while interactions exist among them. These relations mostly consist of referencing several named entities such as subprograms and variables within their corresponding scope. The scope of an entity is the domain where it is accessible or to be more precise if it is defined then it is accessible. Named entities, e.g. variables, named constants, subprograms and modules, as well as some unnamed entities, e.g. labels and file unit numbers, have a scope.

The scope is a limiting capability which enables several program units work together but with independent data environments. The scope of an entity can vary in size and can be as big as the whole program or as small as a part of a statement.

A *global entity* is one that can be accessed throughout the whole program, e.g. name of an external subprogram whereas a *local entity* can be accessed within the scope of a subprogram, e.g. a statement label. There are many minuscule points regarding the concept of scope and hence, only the essential ones are described here with the help of examples. To begin with, consider the following example:

```
1  PROGRAM Main
2      INTEGER :: a
3
4      ...
5      CONTAINS
6
7      SUBROUTINE MySub
8          INTEGER :: b
9          ...
```

```

10      END SUBROUTINE MySub
11      END PROGRAM Main

```

In this example, the variable *a* is a global one which is accessible in the whole program as well as within the subroutine *MySub*. On the other hand, *b* is a local variable which is only accessible within the internal subroutine but cannot be accessed from the main program. This is called access through a host association. If another variable is declared as a local one within the internal subroutine then the access to the global variable will be canceled. For example in the following code, the variable *a* within the internal subroutine is a local variable and it differs from the global one in the main program, i.e. the value 4 is not accessible because of the cancellation of host association:

```

1      PROGRAM Main
2          INTEGER :: a
3
4          a = 4
5          ...
6      CONTAINS
7
8          SUBROUTINE MySub
9              INTEGER :: a
10             ...
11         END SUBROUTINE MySub
12     END PROGRAM Main

```

For the case of an external subprogram, the host association is not valid any more and hence, the data entities will be accessed via argument association, as the following:

```

1      PROGRAM Main
2          INTEGER :: a
3          ...
4          Call MySub (a)
5      END PROGRAM Main
6
7      SUBROUTINE MySub (anArg)
8          INTEGER, INTENT (IN) :: anArg
9          ...
10     END SUBROUTINE MySub

```

Upon the execution of the external subroutine, the variable *a* will be passed to the subroutine as a local variable named *anArg*. In this example, the value of the variable is passed to the external subroutine as the dummy argument *anArg* just for reading purposes. If it is needed to change the value of the actual argument *a*, then the following code must be used:

```

1      PROGRAM Main
2          INTEGER :: a
3          ...
4          Call MySub (a)
5      END PROGRAM Main
6
7      SUBROUTINE MySub (anArg)
8          INTEGER, INTENT (INOUT) :: anArg
9          ...
10     END SUBROUTINE MySub

```

This code just differs in the *INTENT* attribute of the dummy argument. Another way of making variables global is using a common block, for instance:



```

1  PROGRAM Main
2      INTEGER :: a
3      COMMON /myblock/ a
4      ...
5      Call MySub
6  END PROGRAM Main
7
8  SUBROUTINE MySub
9      INTEGER :: a
10     COMMON /myblock/ a
11     ...
12 END SUBROUTINE MySub

```

The variable `a` is declared as an integer in the `myblock` common block and considered as a global variable. This kind of access is granted via *storage association*. Using common blocks is old school and it is not recommended; better alternatives are *argument association* or *use association*. Use association is done via modules, for instance:

```

1  MODULE MyData
2      INTEGER :: a
3  END MODULE MyData
4
5  PROGRAM Main
6      Use MyData
7      ...
8      Call MySub
9  END PROGRAM Main
10
11 SUBROUTINE MySub
12     Use MyData
13     ...
14 END SUBROUTINE MySub

```

This is a much better approach when compared to the use of common blocks. Note that through a use association the integer is made available for the main program. If an internal subprogram existed then it could access the same variable via the *host association* such as in the following example. Both `MySub` and `MyFunc` internal subprograms have access to the integer `a` which is provided by means of use association to the main program and then by host association to the internal subprograms:

```

1  MODULE MyData
2      INTEGER :: a
3  END MODULE MyData
4
5  PROGRAM Main
6      Use MyData
7      ...
8  CONTAINS
9      SUBROUTINE MySub
10         ...
11     END SUBROUTINE MySub
12
13     FUNCTION MyFunc
14         ...
15     END FUNCTION MyFunc
16 END PROGRAM Main

```

Generally, a host can be either a main program, an external program or a module. In the following example, the host is the module and the integer is accessible to `MySub` and `MyFunc` module subprograms through host association:

```

1  MODULE MyData
2      INTEGER :: a
3
4      CONTAINS
5          SUBROUTINE MySub
6              ...
7          END SUBROUTINE MySub
8
9          FUNCTION MyFunc
10             ...
11         END FUNCTION MyFunc
12     END MODULE MyData

```

It is worth mentioning that although local variables of internal subprograms are not accessible to each other but the internal subprograms themselves can be invoked by each other. In the following example, for instance, the variable `a` is not accessible within the `MyFunc` function but the subroutine itself can be invoked:

```

1  MODULE MyData
2
3      CONTAINS
4          SUBROUTINE MySub
5              INTEGER :: a
6              ...
7          END SUBROUTINE MySub
8
9          FUNCTION MyFunc
10             CALL MySub
11         END FUNCTION MyFunc
12     END MODULE MyData

```

While based on the encapsulation concept, the programmer's intent should be modularizing, i.e. breaking down the whole program to independent chunks. On the other hand, it is required for these portions to interact with each other. The interaction is mainly exchanging data entities which takes place through associations. This has been demonstrated in the earlier examples. Hence, it can be concluded that associations are working against the limitations of the scope.

From another point of view, the concept of scope permits the user to incorporate the same name for different entities in various parts of a program without any conflicts. For example, local variables in different subprograms can have identical names but different declarations, provided that they are not referring to the same variable.

In contrast, association is a means to use an entity in the same scope or other scopes of the program with different names, i.e. while using common blocks, several names can be used to refer to the same entity. Although these two concepts help the programmer to use entities with desirable names, it is recommended to use meaningful names. In addition, it is advised to avoid choosing similar or identical names unless there exists a greater purpose behind this selection.

As mentioned earlier, the association mechanism is somehow working against the scopes. In other words, by means of association it is possible to remove the borders between scopes. Therefore, it should be used with care and consideration to keep faithful to the concept of data encapsulation.

Now, let us investigate various associations which are already introduced by means of some examples. An association can occur in any of the following forms:

- *Name association* establishes an association between names in different scoping units with one of the following forms:
  - *Argument association* is established between an actual argument located in the scope of the subprogram reference and the dummy argument within the scope of the subprogram. The association and dissociation of the argument is done upon the entering and exiting the subprogram, respectively. An argument association is present in every subprogram invocation. In the example previously provided in Sect. 1.5.3, the external subroutine SumUp gained access to the variables of another scope using an argument association.
  - A subprogram or program can gain access to all or some of the public entities of a module by means of a USE statement (more details in Sect. 1.6.1). This association is a *Use association*. In this sort of association to resolve name conflicts, a renaming capability of entities is provided with the ONLY statement.
  - By means of a *host association*, the data environment of the *host* is made available to internal subprograms and derived type definitions within the host. However, there is no renaming capability. Therefore, a local entity overrides a host association, namely for an entity which exists with the same name both in the host and in the subprogram, the access to the entity in the host will be denied. A *host* is a program unit which can contain the internal subprogram, i.e. a main program, an external subprogram or a module.
- *Pointer association* is a dynamic association for a pointer during the execution of the program. During this period, the association status and definition status of a pointer can either be undefined, disassociated, associated or defined. Association of a pointer is done either by a TARGET attribute or an ALLOCATE statement.
- *Storage association* occurs when several variables share the same memory storage unit. In other words, the same data can be referenced using different names. This can be done with EQUIVALENCE and COMMON statements.
- *Sequence association* is a special case of argument association which can apply to characters and arrays, e.g. assumed shape arrays and assumed-length characters.

A program unit is made of several non-overlapping *scoping units*. For an entity, a scoping unit is considered as one of the following:

- a program unit or subprogram excluding any scoping units in it, i.e. derived type definitions and interface bodies,
- a subprogram interface body, excluding any scoping units in it, i.e. derived type definitions, interface bodies and subprograms, or
- a derived type definition.

When it comes to scope, there exists many subtle points among which the followings are mentionable:

- Labels have a local scope.
- I/O file unit numbers have a global scope.

A variable which is defined in the scope of a main program, module or subprogram is called a *local variable*.

## 1.7 Data Type Declaration

In FORTRAN terminology, every piece of data is called a *data entity*. A data entity is either a *data object*, result of an expression or result of a function reference; called a *function result*. A data object is either a *variable* or a *constant*. If a name is assigned to the data, it is called a *named data object*. A data object can be *static*, i.e. it has a fixed memory location, or *dynamic*, i.e. it does not have a fixed memory location. The *definition status* of a data object depends on whether a value is assigned to it or not: it is *defined* when a value is assigned otherwise the status is *undefined*. In addition to the definition status, dynamic data objects have an additional *allocation status* which can be either undefined, associated, or disassociated (Sect. 1.8.6).

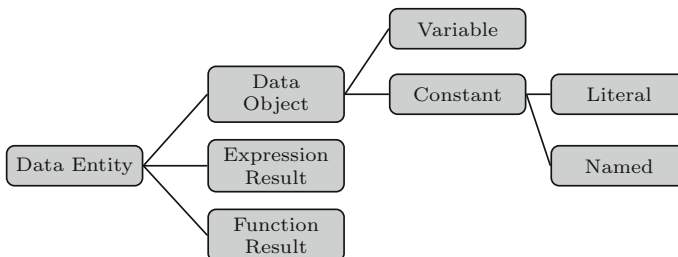
Every data entity has a *rank* which defines if it is a *scalar*, i.e. a single value, or an *array*, i.e. a collection of homogeneous values (Sect. 1.7.8).

A *variable* is a named data object with a variable definition status and/or allocation status. During the program execution, a variable can be *defined*, *undefined* or even *redefined* and take various acceptable values.

A *constant* is a syntactic notation of a mathematical constant value. Without a name, it is called a *literal constant*, e.g. 3.14 but if a name is assigned, it is called a *named constant*, e.g. PI. During the program execution, the value of a constant will not change, i.e. the status of a constant is always defined. Figure 1.14 illustrates the relation between the mentioned terms.

For instance, if one considers a variable named aRealNum then expressions like 2.0\*aRealNum or .NOT. .TRUE. or a function result like 2\*\*SQRT(5) are all considered data entities; they are all representing a data or in simpler terms a value.

In addition to the reference types mentioned in Sects. 1.5.3 and 1.6.1 such as function referencing, there is another type called *data object referencing* which has the appearance of a *data object designator*. Named scalars and named arrays are referenced by a name where *subobjects* are referenced using an object designator. A subobject is a portion of a data object that can be referenced independently of any other portion. A data object may consist of many subobjects. Derived types and arrays consist of subobjects which can be accessed by means of one or more



**Fig. 1.14** Data entities in FORTRAN

qualifiers following the parent object name, e.g. `anArray(2:5)`, `aType%node` or `aType-dArray(2)%node` (Sects. 1.7.7 and 1.7.8).

But prior to any valid data object references, it is required to prepare a proper data environment. A data environment is set up by choosing the right properties for data objects. In a more technical way, a data environment is where the type declaration of data objects takes place. Type declaration is selecting the *type* for data objects. FORTRAN uses *data types* to determine how to deal with data entities engaged in various operations.

There are two *types* of data objects: *intrinsic* and *derived types*. An intrinsic data type is a standard built-in FORTRAN type which can be either an INTEGER, a REAL, a COMPLEX, a LOGICAL, or a CHARACTER type. In contrast, a derived data type is a user defined combination of the intrinsic types, i.e. it is derived from the intrinsic types. By defining the *type* of a data object, the following properties will be determined:

1. a type name, e.g. INTEGER,
2. a set of valid values, e.g. { .TRUE., .FALSE. },
3. constant forms to assign a valid value, e.g. 0.012D-12,
4. *type parameters* to fine-tune the properties, e.g. KIND = 4, and
5. valid operations and procedures to manipulate the values, e.g. 120 + 11.

As mentioned earlier, there are no reserved words in FORTRAN but when it comes to type names, the story is different. Intrinsic data type names cannot be used as the name of a derived type. Another exception is DOUBLEPRECISION which cannot be used either.

In any data environment a variable can be declared using a declaration statement by means of the following syntax:

```
decl-type-spec [type-parameter-selector] [, attr-spec] ...:: ] entity-decl-list]
```

In this general form of a declaration statement, a type attribute specifier (*attr-spec*) is used to select an appropriate attribute which will be discussed in more detail in Sect. 1.8. A type parameter selector (*[type-parameter-selector]*) is used to select the kind or length parameter which will be discussed in Sect. 1.7.1. A declaration type specifier (*decl-type-spec*) is a name which is used for a specific type, either intrinsic (*intrinsic-type-spec*) or derived type (*derived type-spec*), e.g. INTEGER or TYPE. A list of entity names (*entity-decl-list*) consists of objects (separated by commas) with the mentioned attributes or function names. For instance, in the following example the type of the function result for a function named `calcFrequency` is declared to be an integer:

```
1  INTEGER :: calcFrequency
```

In the following example the type of a list of objects is declared:

```
1  INTEGER, DIMENSION(10) :: myInt1, myInt2
```

Two arrays of 10 integer elements are declared with the variable names of `myInt1` and `myInt2`. The type specifier is INTEGER with the DIMENSION attribute. These variables are declared but still undefined because no values have been assigned to

them. *Initializing* is the act of assigning a value to a variable or a named constant in the declaration construct which is optional for a variable but mandatory for a named constant. It is also possible to assign an *attribute* to a type declaration to make it more specialized: for instance, to convert the declaration of a real type variable to an array of real type numbers or to a named constant, among others (Sect. 1.8). But the simplest way to initialize a variable is assigning a value. Whenever a valid value is assigned to the variable using an *assignment statement* the status will be changed to defined. The syntax for a simple scalar assignment is as follows:

```
variablename = expr
```

In the following example, a variable named `r1` is declared of type `REAL` and then, a value of 0.001 is assigned to it and changes its status from undefined to defined:

1  
2  
3

```
REAL :: r1
r1 = 0.001
```

If the variable is of a compound type, such as an array or a structure, then only when every subobject of that compound variable is defined, the whole variable will be considered as a defined variable. The same concept is true for a character variable. For instance in the following code, variables named `c` and `string2` are undefined because no values are assigned neither to `c` and nor to the last character of `string2`:

1  
2  
3  
4  
5  
6  
7  
8

```
REAL :: a, b, c
CHARACTER(LEN = 5) :: string1, string2
a = 1.0
b = 2*a
string1 = 'abcde'
string2 = string1(1:4)
```

### 1.7.1 Type Parameters

Each intrinsic type can only accept certain values, which are processor-dependent for `INTEGER`, `REAL` and `CHARACTER` types. A *type parameter* is used to parameterize a data object. Namely, it is used to select between different representations of a type and thus, different valid values. For instance, in the case of an integer type, it is possible to select between different ranges of integers or for a logical type to select a packed logical representation and save memory.

There are two type parameters in `FORTTRAN`: *kind* which can be used for every intrinsic type and *length* which can be used for the character type to specify its size.

There are a few intrinsic functions which help us dealing with kinds. But before investigating the kinds more deeply, it is a very good idea to understand what is going on in the memory while dealing with data objects and how approximations are introduced. These topics will be investigated in the following subsection and after that the intrinsic types will be elaborated in more detail.

### 1.7.2 Data Representation

Using high-level programming languages to handle data objects, obscures the underlying in-progress processes. Nevertheless, catching a glimpse of how the compiler handles the raw data will improve a programmer's understanding of data handling and the errors involved. The higher the level of a programming language the more abstraction is involved, i.e. more details are hidden.

In reality at the lowest level, the computer memory consists of cells which store either 0 or 1 bits representing a binary system. Memory cells are usually grouped into 2, 4, 8, 16, 32, 64 and 128 bits which are used to store any sort of data. A  $n$ -bit storage can hold  $2^n$  unique combination of bits, e.g. a 4-bit block of memory can generate the  $2^4 = 16$  combinations (see Table 1.11).

Each one of these combinations can be used to *represent* any type of data; 16 patterns to represent 16 characters, 16 different country names, numbers 200–216 etc. It can be concluded that the computer is dealing with on/off currents, making bit patterns in one level where on the higher levels, these patterns of signals can be interpreted in any way required by a user. The interpretation of binary patterns is called *data representation*. Due to the fact that a *fixed* number of bits is used to form a pattern and make up its storage unit, the number of representable combinations is limited.

A computer treats and processes numbers in two major groups: *fixed point numbers* (i.e. integers) and *floating point numbers* (i.e. real numbers). In this context, the term *radix point* is used to separate the fractional part of a number from its integer part. In other words, radix point is a more general concept compared to *decimal point*; the former is not only used for a decimal number but it is also used for bases other than ten. In a fixed point number, the radix point is always after the least-significant digit whereas in a floating point number the radix point can be positioned anywhere.

A fixed number of bits can be designated for an integer number and based on the interpretation of the patterns, two types of integers can be represented: *unsigned integers* which only represent positive integers plus zero, and *signed integers* which represent positive and negative integers plus zero. There are three schemes to represent signed integers: *sign-magnitude representation*, *1's complement representation* and *2's complement representation*.

The value of an unsigned integer is simply the converted value from binary to decimal, i.e. the unsigned decimal integer is just equal to the magnitude of the binary pattern. Hence, an unsigned  $n$ -bit integer can represent only positive numbers from 0 to  $2^n - 1$ . Unsigned integers are not a part of standard FORTRAN.

**Table 1.11** Combinations of a 4-bit block of memory

|              |      |      |      |      |      |      |      |      |
|--------------|------|------|------|------|------|------|------|------|
| Pattern      | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    |
| Binary value | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| Pattern      | 9    | 10   | 11   | 12   | 13   | 14   | 15   | 16   |
| Binary value | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

**Table 1.12** Comparison of different integer representation schemes

| Representation scheme | $n$ -bit covering range                     | Summary   |
|-----------------------|---|---|
| Unsigned              | $[0, 2^n - 1]$                              | Only zero and positive numbers are presentable  |
| Sign-magnitude        | $[-(2^{n-1} - 1), 0] \cup [0, 2^{n-1} - 1]$ | Zero is defined twice, separate processes are required for positive and negative integers   |
| 1's complement        | $[-(2^{n-1} - 1), 0] \cup [0, 2^{n-1} - 1]$ | Zero is defined twice, positive and negative integers are processed separately  |
| 2's complement        | $[-(2^{n-1}), 0] \cup [0, 2^{n-1} - 1]$     | One representation for zero, positive and negative integers are treated together in addition and subtraction. Subtraction can be carried out using the addition logic |

On the contrary, the value of a signed integer can be interpreted differently based on which scheme is used. In all schemes, one of the storage bits is required to be used to indicate the sign of the number. This bit is called a *sign bit*. In a binary number, the bit position with the greatest value is the left-most bit which is also called the *most significant bit* (MSB) or the *high-order bit*. The MSB is used as the sign bit of a binary number; representing positive values with 0 and negative values with 1. Hence, in a  $n$ -bit signed integer, only the *remaining*  $(n - 1)$ -bits are used to store the magnitude of the number and the  $n$ th-bit is used to hold the sign. The scheme determines how this  $(n - 1)$ -bit part is evaluated:

- In the *signed magnitude method*, the absolute value of both positive and negative integers is the value of the remaining  $(n - 1)$ -bit part which is directly converted to decimal.
- In the *1's complement method*, again for positive numbers the remaining bits are converted to decimal but the absolute value of negative integers is equal to the magnitude of complement of the remaining bits. Complement of 0 is 1 and vice versa, e.g. complement of  $11001010b$  is  $00110101b$ .
- In the *2's complement method*, again for positive numbers the remaining bits are converted to decimal but the absolute value of negative integers is equal to the magnitude of complement of the remaining bits plus one.

The covering range of these schemes is summarized in Table 1.12. According to the table, the 2's complement method manages to extend the representable range by one extra negative number. In addition, no extra work is required to add and subtract numbers in this scheme. Because of these efficiencies, FORTRAN uses this scheme to handle integer numbers.

As an example, the comparative interpretation of binary patterns in different schemes using 8 bits of memory storage is illustrated in Fig. 1.15. In this figure, binary numbers are increased from top to bottom. Notice if you subtract the first number which is zero by 1, the generated binary pattern will be  $11111111b$  which



| Unsigned          | Sign-magnitude | 1's complement | 2's complement | 8-bit binary |
|-------------------|----------------|----------------|----------------|--------------|
| +0                | +0             | +0             | +0             | 0000 0000    |
| +1                | +1             | +1             | +1             | 0000 0001    |
| +2                | +2             | +2             | +2             | 0000 0010    |
| ...               | ...            | ...            | ...            | ...          |
| +125              | +125           | +125           | +125           | 0111 1101    |
| +126              | +126           | +126           | +126           | 0111 1110    |
| +127              | +127           | +127           | +127           | 0111 1111    |
| ----- Break ----- |                |                |                |              |
| +128              | -0             | -127           | -128           | 1000 0000    |
| +129              | -1             | -126           | -127           | 1000 0001    |
| +130              | -2             | -125           | -126           | 1000 0010    |
| ...               | ...            | ...            | ...            | ...          |
| +253              | -125           | -2             | -3             | 1111 1101    |
| +254              | -126           | -1             | -2             | 1111 1110    |
| +255              | -127           | -0             | -1             | 1111 1111    |

**Fig. 1.15** Decoding binary patterns in different schemes

corresponds to  $-1$  only in 2's complement method. In addition, increasing the largest positive number, i.e. 127, will result in a negative number in complement representation and minus zero in sign-magnitude method. This indicates that the range of a signed integer must be considered prior to use because out-of-range values may lead to absurd results.

In FORTRAN, the 2's complement method is used to deal with signed integer numbers and it is incorporated using a mathematical representation with the following general form:

$$i = s \sum_{k=0}^{q-1} w_k r^k, \tag{1.1}$$

where

- $i$  is the integer value
- $s$  sign, +1 or  $-1$
- $r$  is the base, an integer greater than 1
- $q$  is the number of digits, an integer greater than 0
- $w_k$  is the  $k$ th digit, an integer  $0 \leq w_k < r$

Note that this formula is based on the *least significant bit* (LSB) numbering scheme in which the numbering of bits starts from the right-hand side, i.e. the least significant bit is the rightmost one. This model will cover a total of  $r^q$  integer numbers within the range  $[-r^q - 1, r^q - 1]$ . In the binary computer architecture, the base is  $r = 2$  and thus, the only usable digit is  $w_k = 1$ . Finally every integer can be indicated using the following form:

$$i = \pm \sum_{k=0}^{q-1} w_k 2^k = \pm (w_{q-1} w_{q-2} \dots w_2 w_1)_b. \tag{1.2}$$

Depending on how big the dedicated memory to an integer is, i.e.  $q$ , the range will be bigger. For example, if the storage unit is 1 byte then  $q = 8$  bits or digits in binary-base will be used for an integer. Therefore,  $2^8 = 256$  integer numbers (including zero) can be shown with this amount of memory. Because FORTRAN implements the 2's complement method, the range of  $[-128, 127]$  will be used.

Although fixed point numbers, i.e. integers, are quite simple, they are ample to manage whole numbers. In contrast, the representation model and the corresponding calculations of the floating point numbers, i.e. real numbers, are not that simple. Hence, in order to be efficient in terms of computational costs, choosing a real number should be done just when an integer cannot manage the same task.

It is worth reminding that there is an infinite number of real numbers in a range. Because a limited amount of storage memory is designated to the computer representation of the real numbers, it is not possible to represent every real number. Consequently, the unrepresentable numbers are approximated to the nearest representable number and hence, during the process they suffer from loss of precision. However, the floating point scheme manages to represent several very small and very large numbers in a range but obviously, not all existing real numbers.

A floating point representation of a real number has the form of  $m \times r^e$  in which,  $m$  is a fractional number called *mantissa*, *significant* or *fraction*,  $e$  an integer number called *exponent* or *characteristic* and  $b$  is the *radix* or *base*. Although the common base for a floating point representation in a computer is 2, normal mathematical problems use the base ten or the decimal system. For instance consider the number  $0.12 \times 10^{-3}$  in a decimal system then 0.12 will be the mantissa and  $-3$  will be the exponent of the floating point representation and the base will be ten.

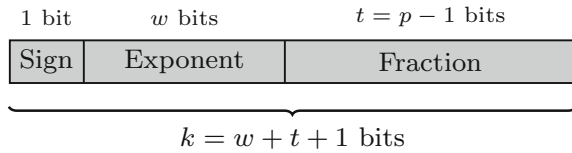
In a real number, a single number can be written with different fractional parts and exponents depending on the number of its *significant digits*. From the point of engineering, significant digits or *significant figures* of a number are the meaningful digits that can be used with confidence. For example, by using a millimeter ruler a number such as 12 mm can be read. This reading has two significant figures and reliable enough. It is possible to say that the reading is approximately close to 12.5 mm. However, the last digit is an approximation and we still have just two significant figures.

Leading zeros in a number are not significant. For instance, 0.012 m still has two significant figures but changing the unit may cause problems concerning the identification of the significant figures, i.e. the trailing zeros are usually misleading. For example, 12000  $\mu\text{m}$  may have two, three, four or even five significant numbers. In order to avoid this ambiguity and preserve the accuracy, using a *scientific notation*<sup>3</sup> is advised, since  $12 \times 10^3 \mu\text{m}$ ,  $1.2 \times 10^{-2} \text{ cm}$  and  $1.2 \times 10^{-5} \text{ km}$  all have two significant figures but represented in different units. More uniformity can be achieved by using the *normalized scientific notation* which is writing the fractional part with

---

<sup>3</sup>In engineering applications, it is common practice to keep the exponent a factor of 3 and use SI prefixes to facilitate reading. This is called an *engineering notation* for example,  $12 \times 10^{-3} \text{ m}$  is easier to be read and understood as 'twelve millimeters' instead of  $1.2 \times 10^{-4}$  which is read as 'one-point-two times ten-to-the-negative-four meters'.

**Fig. 1.16** Layout of a binary floating point representation



one non-zero digit before the decimal point, e.g.  $1.2 \times 10^{-2}$  cm. This concept holds for radices other than decimal, i.e. a normalized scientific notation in base  $b$  would be in the following form:

$$m \times b^e \quad \text{provided that } 1 \leq m < b$$

For instance, a normalized scientific notation in binary will look like  $1.01b \times 2^{11b}$ .

A real number is *encoded* to its binary pattern and then stored in memory using a binary floating point representation. Hence, the radix is always equal to two and the fraction and exponent parts are calculated for every number. As illustrated in Fig. 1.16, the designated memory for a floating point representation is comprised of three fields:

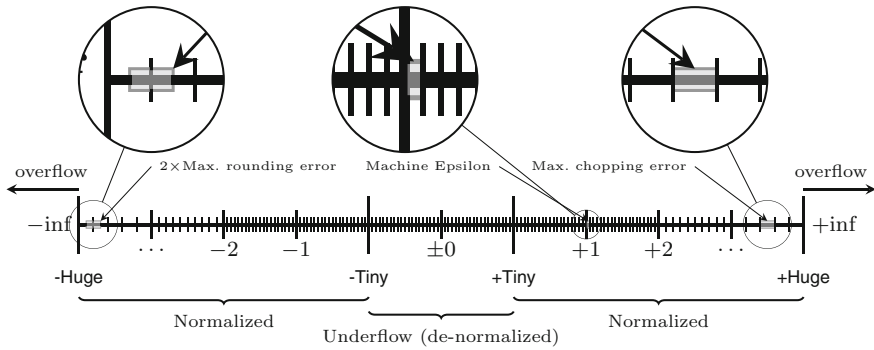
1. the first field is just one single bit which is the sign bit of the real number; similar to the integer representation, 1 indicates a negative number and zero is used for a positive one.
2. The second field is  $w$  bits long and is an unsigned integer which represents the exponent as a power of two. For a normalized number the exponent is stored in a biased mode (E) but for a denormalized number an unbiased exponent (e) is used.
3. The last field holds  $t$  bits of the fraction which are the significant digits of the number.

Obviously, the range and precision of representable real numbers using the floating point representation is highly dependent on the size of the dedicated memory which is called the *encoding length* ( $k$ ) with the usual values of 32, 64 and 128 bits.  $p$  is the number of significant digits of a normalized number which is equal to  $t + 1$ ; one bit more than the actual bits used for storage. For real types, FORTRAN uses a rather complicated representation model in which a real number such as  $x$  can be modeled by the following:

$$x = \left( s \sum_{k=1}^p f_k b^{-k} \right) b^e = m b^e, \tag{1.3}$$

where

- $x$  is a real value
- $s$  sign, +1 or -1
- $b$  is the base (or radix), an integer greater than 1
- $e$  is an integer,  $e_{\min} \leq e \leq e_{\max}$



**Fig. 1.17** Schematic map of real numbers in a floating point representation

- $p$  is the number of mantissa digits, an integer greater than 0
- $f_k$  is the  $k$ th digit, an integer  $0 \leq f_k < b$   
normally  $f_1 \neq 0$  but it can be zero only if  $e$  and all the  $f_k$  are zero

Note that this formula is based on the most significant bit numbering scheme in which the numbering of bits starts from the left-hand side, i.e. numbering the bits starts from the most significant bit. With the introduced model, the common approach is using a binary base; the radix is equal to 2:

$$x = \pm(0.f_1f_2f_3 \dots f_{p-2}f_{p-1}f_p)_b \times 2^e = m2^e. \tag{1.4}$$

Although the representation model formulates the main idea of how numbers are stored in a floating point format, there is more subtlety to it. Using the representation model and considering the restrictions made by a limited encoding length, numbers fall into various ranges. A map of these ranges is illustrated in Fig. 1.17. In this map, vertical lines (tick marks) are normalized numbers which are exactly representable by the model. All normalized numbers are placed in a range from the smallest normalized number (Tiny) to the largest normalized number (Huge). When a number is exactly equal to a normalized number which is a rather rare case, no approximation occurs. However, usually the floating point number is placed between normalized numbers and therefore, an error is introduced.

Generally, because the range and precision of the representable numbers are limited, several cases may arise while dealing with arithmetic operations. These special cases are called *exceptions*. The possible exception categories are listed in Table 1.13. Based on the range which the number falls into, approximations and/or exceptions may be introduced while storing. In this table,  $w$  is the width of the exponent field. An exception may also be raised due to the lack of required precision for a number, namely there are not enough digits to represent a very precise number. Inevitably, this leads to rounding the number; this case is called an *inexact* exception. This is a very

**Table 1.13** Categories of numbers in floating point representation

| Number (x)     | Range                                   | Sign | Exponent          | Fraction      |
|----------------|---|------|-------------------|---------------|
| ± Normalized   | +Tiny < x <+Huge or -Huge<br>< x <-Tiny | ±    | $1 < e < 2^w - 2$ | Any number    |
| ± Denormalized | -Tiny < x <+Tiny excluding zero         | ±    | 0                 | Any number    |
| ± 0            | Positive/negative zero                  | ±    | 0                 | All zeros     |
| ± INF          | +Huge < x or x <-Huge                   | ±    | $2^w - 1$         | All zeros     |
| NaN            | Not a number                            | ±    | $2^w - 1$         | Some one bits |

common exception since just at the first step of using a model as a representative of a real number, errors are being introduced because quantities like  $\pi = 3.14159265 \dots$  are in need of infinite significant digits to be represented accurately which is not possible nor practical with the limited available resources. Consequently, some *significant digits* must be omitted and an *approximation* will be introduced from there on.

This approximation can take place in two ways: *chopping*, which is just omitting the extra significant digits and *rounding* which is just rounding-off the number to the nearest representable number (ticks of the map). Chopping is a faster way for the processor but the results are biased whereas rounding involves less error and less bias because positive and negative errors may occur and cancel each other. The shaded areas in Fig. 1.17 illustrate the maximum error possible for these types of approximations. The corresponding exception for these approximations is inexact exception which happens all the time during evaluation of expressions in FORTRAN.

A famous special case of inexact exceptions is named the *machine epsilon* (Epsilon) which is a very small number and negligible in comparison to 1 or in other words, it is the maximum chopping error that can happen around 1. The maximum value of the chopping is two times the maximum value of rounding. The machine epsilon can be evaluated using the following formula:

$$\text{Epsilon or machine epsilon: } \epsilon = b^{1-p} \quad (1.5)$$

In this formula,  $b$  is the base and  $p$  is the number of significant digits of the fraction part of the floating point number. Alternatively, the following code can calculate the machine epsilon in FORTRAN:

```

1      REAL (KIND = 4) :: r4
2      LOGICAL :: notDifferent
3
4      r4 = 1.
5      notDifferent = .FALSE.
6      DO WHILE (notDifferent .EQV. .FALSE.)
7          r4 = r4/2.
8          IF (r4+1. <= 1.) notDifferent = .TRUE.
9      END DO
10     Print *, 'Machine Epsilon = ', r4*2.

```

In the map considered in Fig. 1.17, zero is placed in the middle with two representations: positive and negative. While moving from zero to either extremities, the density of tick marks decreases, namely the density of normalized numbers decreases as they increase in magnitude. The numbers beyond the maximum representable number cannot be presented, and are considered as infinity ( $\pm\text{inf}$ ) in both negative and positive directions. This exception is called an *overflow*.

On the other hand, the numbers smaller than the  $\pm\text{Tiny}$  threshold, lose one of their significant digits and are called *denormalized numbers*; the occurred exception is an *underflow*.

An *invalid exception* is used for values that are not mathematically presentable, i.e. not a number (NaN). Note that for the especial case of dividing a number by zero, the exception *divided-by-zero* will occur (Table 1.14).

In FORTRAN, a representation model can be selected using the *KIND* parameter which must be set to a processor-dependent value; usually a number which is equal to the storage size of the kind in bytes. The *KIND* parameter for each data type will be discussed in Sect. 1.7.3.

In Table 1.15 real kinds used in INTEL® FORTRAN are listed with the description of each kind, encoding length ( $k$ ), significant digits in binary format ( $p$ ), number of bits used for the exponent ( $w$ ), the smallest number compared to 1 (epsilon), minimum and maximum value of the exponent for a normalized number ( $e_{\min}$  and  $e_{\max}$ , respectively).

In order to understand how a number is stored using the introduced representation model, let us consider a floating point number with a 32-bit memory storage ( $k = 32$ ). The sign bit of the pattern is readily understood; it is the sign of the whole number. The fraction part holds the significant digits of the number using  $p$  bits for normalized number and  $p - 1$  bits for denormalized numbers, i.e. a loss of precision occurs. Denormalized numbers are necessary because it is not possible to present zero with a normalized pattern. And finally the exponent part is stored in a *biased* format for normalized numbers. The bias is a constant used to make the exponent non-negative. For the case of 32-bit floating point numbers, the bias is equal to 127 for normalized numbers. Based on the category of the number, one of the following cases will happen:

**Table 1.14** List of exceptions regarding floating point numbers in FORTRAN

| Exception       | Reason   | Examples  |
|-----------------|--|---|
| Invalid         | No mathematical value NaN  | $\frac{0.0}{0.0}$ , $0.0^{0.0}$ or $\text{Inf}-1$ |
| Overflow        | Numbers larger than Huge   | $\text{Huge} \times \text{Huge}$                  |
| Divided-by-zero | Division of a non-zero value by zero                               | $\frac{1.0}{0.0}$                                 |
| Underflow       | Numbers between $-\text{Tiny}$ and $-0$ or $+0$ and $+\text{Tiny}$ | $\text{Tiny}/0.2$                                 |
| Inexact         | Is in the range but between the ticks as in Fig. 1.17              | $(1.0 + 2^{-40})_b$ in $\text{Real}^*4$           |

**Table 1.15** Models for real numbers in FORTRAN

| Kind | Storage memory ( $k$ ) | Digits ( $p$ ) | Exponent width ( $w$ ) | Common name      | Epsilon                       | $e_{\min}$ | $e_{\max}$ |
|------|------------------------|----------------|------------------------|------------------|-------------------------------|------------|------------|
| 4    | 32 bits                | 24 bits        | 8 bits                 | Single precision | $1.192 \dots \times 10^{-7}$  | -126       | +127       |
| 8    | 64 bits                | 53 bits        | 11 bits                | Double precision | $2.220 \dots \times 10^{-16}$ | -1022      | +1023      |
| 16   | 128 bits               | 113 bits       | 15 bits                | Quad precision   | $1.925 \dots \times 10^{-34}$ | -16382     | +16383     |

1. Normalized numbers are usually encountered in normal arithmetic calculations and can acquire either a positive or negative sign with a range enclosed with Tiny and Huge values.
  - a. Fraction: Considering the mathematical model and the fact that in normalized numbers  $f_1$  is always equal to 1, there is no point in storing it. In other words, normalized numbers always have an implicit 1 before the radix point and therefore, storing this implicit number is trivial. By avoiding the storage of this digit, an extra bit is gained to store the fraction with more significant digits. For instance, the 32-bit single-precision real number occupies 32 bits of storage of which 1 bit is used as the sign bit, 8 bits are used to store the exponent and the rest, namely  $t = 32 - 1 - 8 = 23$  bits are used for the mantissa. Since one bit is always implied, practically  $p = 24$  bits are being used to represent the fractional part. The result is acquiring more precision.
  - b. Exponent: Because it is required for a normalized number to represent very large numbers as well as very small ones, the exponent must cover both negative and positive numbers. For this, the exponent for a normalized number is stored in a biased format.
2. Zero is stored in both negative and positive value.
  - a. Fraction: Filled with zero bits.
  - b. Exponent: Filled with zero bits.
3. Denormalized numbers cover the range slightly less than the +Tiny to slightly larger than zero and therefore, there is a small gap between them. The same holds for the negative side. These numbers are very small numbers which have less accuracy than the normalized ones.
  - a. Fraction: There is no implicit 1 in the fraction part and instead an implicit zero is used; during the process one significant digit is lost.
  - b. Exponent: For a denormalized number the exponent is always zero but it will be evaluated as  $-126$  for 32-bit real numbers to represent small numbers.

|                             | <b>32-bit floating point binary</b>      | <b>Equivalent decimal</b>   |
|-----------------------------|--|---|
| Negative zero               | 1 0000 0000 000 0000 0000 0000 0000 0000 | = -0  |
| Positive zero               | 0 0000 0000 000 0000 0000 0000 0000 0000 | = +0  |
| Smallest denormalized       | 0 0000 0000 000 0000 0000 0000 0000 0001 | = $\pm 2^{-126} \times 2^{-23}$<br>= $\pm 2^{-149} \approx \pm 1.4013\text{E} - 45$         |
| Largest denormalized        | 0 0000 0000 111 1111 1111 1111 1111 1111 | = $\pm 2^{-126} \times (1 - 2^{-23})$<br>$\approx \pm 1.17549421\text{E} - 38$              |
| Smallest normalized (Tiny)  | 0 0000 0001 000 0000 0000 0000 0000 0000 | = $\pm 2^{1-127} \times 2^0$<br>$\approx \pm 1.175494351\text{E} - 38$                      |
| Largest normalized (Huge)   | 0 1111 1110 111 1111 1111 1111 1111 1111 | = $\pm 2^{254-127} \times (2 - 2^{-23})$<br>$\approx \pm 3.402823467\text{E} + 38$          |
| Positive infinity (+inf)    | 0 1111 1111 000 0000 0000 0000 0000 0000 | = $+2^{255-127} \times (2^0)$<br>= $+3.402823669\text{E} + 38 \approx +\text{inf}$          |
| Negative infinity (-inf)    | 1 1111 1111 000 0000 0000 0000 0000 0000 | = $-2^{255-127} \times (2^0)$<br>= $-3.402823669\text{E} + 38 \approx -\text{inf}$          |
| Smallest not a number (NaN) | 0 1111 1111 000 0000 0000 0000 0000 0001 | = $\pm 2^{128} \times (1 + 2^{-23})$<br>= $\pm 3.402824075\text{E} + 38 \approx \text{NaN}$ |
| Largest not a number (NaN)  | 0 1111 1111 111 1111 1111 1111 1111 1111 | = $\pm 2^{128} \times (2 - 2^{-23})$<br>= $\pm 6.805646934\text{E} + 38 \approx \text{NaN}$ |

**Fig. 1.18** Decoding floating point binary patterns to decimal equivalents

4. Infinity can be either positive or negative and it is a representative of very large numbers.
  - a. Fraction: Filled with zero.
  - b. Exponent: Filled with one.
  
5. Not-a-Number (NaN) is a representative of non-mathematical entities.
  - a. Fraction: a non-zero value.
  - b. Exponent: Filled with one.

It is worth mentioning that some processors can distinguish between a negative and positive zero but in FORTRAN both of these zeros are treated equally in relational expressions. Anyway, if it is required to detect the sign, the Sign intrinsic function may be used. The aforementioned categories, their border values for a 32-bit floating point representation, and the corresponding decoding are all illustrated in Fig. 1.18.

Generally, there are intrinsic functions by which various values regarding a representation model can be obtained; a list of these functions and the corresponding short description are given in Table 1.16. The first column is the name of the function with the argument x, the second column is a brief description of the result value of the function and finally the third column is the type of argument x. These functions are important because they will provide us with values which determine the precision,



**Table 1.16** Intrinsic functions for numeric inquiries of representation models in FORTRAN

| Function        | Result of the function   | Type of argument x   |
|-----------------|--|----------------------|
| Kind (x)        | Kind parameter   | Real/Integer         |
| Digits (x)      | Number of significant digits of the mantissa, $p$ for real and $q$ for integer   | Integer/Real/Complex |
| Epsilon (x)     | Negligible value relative to 1, $(b^{1-p})$  | Real                 |
| Tiny (x)        | Smallest positive number, $(b^{e_{\min}-1})$   | Real                 |
| Huge (x)        | Largest positive number representable by the model, $((1 - b^{-p})b^{e_{\max}}$ or $r^q - 1$ )   | Real/Integer         |
| MaxExponent (x) | Maximum value for model exponent, $(e_{\max})$   | Real                 |
| MinExponent (x) | Minimum value for model exponent, $(e_{\min})$   | Real                 |
| Precision (x)   | Equivalent decimal precision in the model, $(\text{int}((p - 1) \log_{10}(b) + k); k = 1$ if $b$ is an integral power of 10 otherwise $k = 0$ )                      | Real/complex         |
| Radix (x)       | Base of the model, $b$ for real and $r$ for integer  | Real/Integer         |
| Range (x)       | Decimal exponent range of the model, $\text{int}(\log_{10}(\text{huge}))$ for integers, $\text{int}(\min(\log_{10}(\text{huge}), -\log_{10}(\text{tiny}))$ for reals | Integer/Real/Complex |
| Exponent (x)    | Exponent of a real value   | Real                 |
| Fraction (x)    | Fractional part of a real value  | Real                 |
| Sizeof (x)      | Returns the size of occupied memory by the variable x in bytes   | Any type             |

accuracy and range for a type with a specific kind. For instance, Huge (x) is the largest positive number that a variable of type x can contain.

In conclusion, not everything is covered here on floating point representation and for a meticulous eye, there are more points to understand. However, a good description is presented on how a real number is dealt within a digital computer. Note that especially in releases before FORTRAN 95, handling the mentioned exceptions is not supported. Hence, the acquired knowledge will be helpful when programming. For more details on exception handling and floating point representation, one may refer to [16].

### 1.7.3 Intrinsic Data Types

In the previous subsection, the internal representation of integer and real numbers were investigated to obtain a better understanding of the background of data manipulation. This introduction also makes the selection of the appropriate type parameter easier. In this subsection, declaring the types and their type parameters within FORTRAN will be introduced.

FORTTRAN provides five default types and their proper operations; called *intrinsic types*. The first step of declaring a data entity is declaring the data type and thus, extra attributes can be assigned to it. Intrinsic data types consist of *numeric* types, i.e. integer (INTEGER), real (REAL) and complex, (COMPLEX) as well as *non-numeric* types, i.e. character (CHARACTER) and logical (LOGICAL) type. Most of the time, these intrinsic data types have satisfactory properties for a wide range of applications and it is almost evident what type of data object must be selected in a specific context. However, fine-tuning the properties can be done using *type parameters*. There are two type parameters: *kind* and *length* type parameter.

A kind parameter is represented by a non-negative integer number by which a specific representation of that type is selected. The number of kinds for an intrinsic data type is compiler- and processor-dependent. A default kind value will be assigned if no specific values are selected for a data object. Although usually the kind number indicates the number of memory bytes used for the type, it is not a standardized fact.

Kinds were introduced to omit the problems of program portability because sometimes errors may be introduced to the calculations due to the fact that different processors handle data differently in terms of precision. For example, when a program containing a double precision floating point number declaration is transferred to a machine which basically has a 64-bit architecture, a 128-bit representation will be more than necessary and it would be a good idea to change the type back to an ordinary real number. Such cases require changing the code in each transfer which is not a practical approach.

For normal cases, the default kind will produce satisfactory results but it is necessary to know how to select an appropriate kind when it is necessary, e.g. when a wider range of numbers is required.

As stated before, the following syntax is used for a type declaration:

```
decl-type-spec [type-parameter-selector] [, attr-spec] ...:: ] entity-decl-list]
```

The type parameter selector ([type-parameter-selector]) is either a kind or a length parameter selector. A length type parameter can only be used for a CHARACTER type and will be introduced later. The syntax of a kind parameter selector is as the following:

```
([KIND =] kind-value)
```

The kind specifier (KIND) assigns a positive integer (kind-value) representing the kind. A Kind parameter can be assigned to a variable, function or a named constant. It is also possible to assign kind to a literal constant using an underscore after the literal constant followed by the kind value, i.e. the following syntax:

```
literal-constant_kind-value
```

For example 12.45\_8 is a literal constant of kind 8.

**Table 1.17** Models for integer numbers in FORTRAN

| Kind | Storage memory | Common name | Lower bound                      | Upper bound                     |
|------|----------------|-------------|----------------------------------|---------------------------------|
| 1    | 8 bits         | –           | –128                             | 127                             |
| 2    | 16 bits        | Short       | –32, 768                         | 32, 767                         |
| 4    | 32 bits        | Long        | –2, 147, 483, 648                | 2, 147, 483, 647                |
| 8    | 64 bits        | Longlong    | –9, 223, 372, 036, 854, 775, 808 | 9, 223, 372, 036, 854, 775, 807 |

### 1.7.4 Numeric Data Types

There are four integer kinds in FORTRAN, each corresponding to a representation model as listed in Table 1.17. Based on the results of this table, the biggest representable integer number has 19 digits but the most common one is called a *long* integer.

Suppose that we want to make sure that the right kind is always selected for a particular type of integer number, in other words, for our purpose an integer with  $R + 1$  number of decimal digits is required. Therefore, an integer type is needed to support a range from  $-10^R$  to  $+10^R$ . Using the intrinsic function `Selected_int_kind(R)` which results in the kind corresponding to the aforementioned range, one can make sure that the adequate range is provided even if the code is transferred to another machine. For instance, if an integer number is needed with at least 15 digits, the following code will find the corresponding integer kind:

```

1  INTEGER, PARAMETER :: MYKIND = selected_int_kind (15)
2  INTEGER (KIND = MYKIND) :: anInteger
3  ! The alternative syntax may be used:
4  INTEGER*MYKIND :: anotherInteger

```

In this code, first a named constant `MYKIND` is defined which corresponds to an integer type with the  $-10^{15} < i < 10^{15}$  and then it is used to declare `anInteger` variable with our intended range. If the code is transferred to another machine, the `MYKIND` may be different but this trick will ensure the selection of the correct kind. If an appropriate kind does not exist, the `Selected_int_kind(R)` function returns  $-1$ . In other words, `Selected_int_kind(R)` returns the most appropriate of kinds among the available ones, depending on the required range.

The following code determines the number of kinds of the system:

```

1  INTEGER :: i, defaultKind, temp, kindCount
2
3  i = 0
4  temp = 1
5  kindCount = 1
6  DO WHILE (selected_int_kind(i) > 0)
7      IF (temp .ne. selected_int_kind(i)) THEN
8          kindCount = kindCount + 1
9          temp = selected_int_kind(i)
10     END IF
11     i = i + 1
12 END DO
13 PRINT *, kindcount, ' integer kinds are available on this system.'

```

If a kind is not specified, the default kind will be selected. The importance of knowing the kind of an integer is realizing its range. This will help preventing or at least understanding the possibility of an exception such as an overflow. The range and the precision can be realized for a real type.

In the case of an overflow, increasing the largest integer number of any kind by 1, results in the smallest integer of that kind. This property can be used to determine the lower boundary for the range of an integer kind. The Huge() intrinsic function returns the biggest allowable number for a type. This function along with the mentioned property is used to obtain the range of the default integer type. The listing is as follows:

```

1      INTEGER :: i, lowerBound, upperBound
2
3      defaultKind = Kind ( i )
4      lowerBound = Huge ( i ) + 1
5      upperBound = Huge ( i )
6      PRINT *, 'The default kind is ', defaultKind
7      PRINT *, 'and it ranges from', lowerBound, ' to ', upperBound

```

As mentioned earlier, it is possible to assign an integer kind parameter to a literal integer constant. This capability is used in the following listing to illustrate that overflow of a number will produce undesired results without generating any compiler errors:

```

1      INTEGER, PARAMETER :: SHORT = selected_int_kind (4),
2      & LONG = selected_int_kind (8)
3      INTEGER (KIND = SHORT) :: a
4      INTEGER (KIND = LONG) :: b
5
6      a = 1234_SHORT
7      b = 12345678_LONG
8      PRINT *, 'a = ', a, ' b = ', b
9      ! output: a= 1234 b=12345678
10
11     b = 1234_SHORT
12     a = 12345678_LONG
13     PRINT *, 'a* = ', a, ' b* = ', b
14     ! output: a* = 24910 b* = 1234
15
16     b = 1234_LONG
17     PRINT *, 'b** = ', b
18     ! output: b** = 1234

```

A *real* type is an intrinsic data type representing an approximation for mathematical real numbers using the floating point number concept. INTEL® FORTRAN uses IEEE three basic binary formats for representing real numbers. For a real type, the KIND parameter sets the precision and range.

In declaring real numbers, the specifier REAL is identical to a single-precision floating point number and is the default real type. However, it is possible to change the default real type by setting a compiler option.

A double precision real number can be introduced either with a DOUBLE PRECISION specifier or a REAL specifier plus a kind type parameter indicating a higher precision. However, no kind parameters are defined for the DOUBLE PRECISION type specifier.

The appropriate kind number depends on the processor and the compiler but one single precision and one double precision real number type are the minimums that a

FORTRAN compiler must provide. The syntax for a real type is identical to the integer type declaration. Only in the latter, the REAL type specifier will be used:

```

1     REAL    :: aSinglePrecisionReal
2     REAL*4  :: anotherSinglePrecisionReal
3     REAL*8  :: anotherSingleOne
4     DOUBLE PRECISION :: aDoublePrecisionReal
5     REAL*16 :: anotherDoubleOne

```

It is worth noticing that a real literal constant is identified by one of the following:

1. a decimal point, e.g. 1.0 or 1.
2. an exponent letter (E, D or Q), e.g. 1.E0, 1E0, 1D0 or 1Q0

To be more precise, the syntax for a real literal constant is as follows:

significant [exponent-letter exponent][\_kind-param]

The fractional part of the real number (significant) is followed by an exponent letter (exponent-letter) and the exponent (exponent) itself. exponent-letter is either E or D.

In addition, when dealing with operands of different type or kind in an expression, called a *mixed mode expression*, FORTRAN converts the result to the strongest of the two types/kinds; consider the following example:

```

1     REAL    :: r4
2     REAL*8  :: r8
3     REAL*16 :: r16
4
5     r4 = 1 / 3
6     PRINT *, r4
7 ! output: 0.0000000E+00

```

In this example, the output is 0.0000000E+00 because both 1 and 3 are integers. The result is as an integer, i.e. 0, and the type implicitly will be converted to a real number that is 0.0000000E+00. However, if one of the two numbers was declared as a real literal constant then the result will be of the strongest form, i.e. a real type and the assignment to r4 will be correct:

```

1     r4 = 1. / 3
2     PRINT *, r4
3 ! output: 0.3333333E+00

```

The following example demonstrates the limitations of precision when dealing with a real number. Note that the number of significant figures is kept the same even the target has more precision:

```

1     r4 = 1E0 / 3
2     print *, r4
3 ! output: 0.3333333
4     r4 = 1. / 3
5     print *, r4
6 ! output: 0.3333333
7     r8 = 1. / 3.
8     print *, r8
9 ! output: 0.333333343267441
10    r16 = 1. / 3.
11    print *, r16
12 ! output: 0.333333343267440795898437500000000

```



(real-part, imaginary-part)

The complex type has the same number of kinds as a real type. Therefore, the memory used by a complex type is twice that of the corresponding real type. The intrinsic functions Kind, Range and Precision can be used for this type as well (see Table 1.16). Similarly, the Selected\_Real\_Kind intrinsic function can be used for a complex type. For instance:

```

1  COMPLEX :: c1
2  COMPLEX (Selected_Real_Kind(10,20)) :: c2
3  COMPLEX (Kind = 8) :: c3, c4
4
5  c1 = (1, 3.2)
6  c2 = (0.00001, .002e20)
7  c3 = (1., 3.2D0)
8  c4 = (1._8,3.2_8) + c3

```

### 1.7.5 Non-Numeric Data Types

The kind parameter can be used for non-numeric data types as well, i.e. for character and logical types. The kind parameter for a character introduces other character sets which permits using additional graphical symbols. However, the default character kind, i.e. a single-byte character, is enough for most cases and normally no kind selection is required.

In addition to the KIND parameter, character types have the length parameter, named LEN. The length parameter indicates the number of characters of the string which can even be equal to zero. The syntax for declaring a character type can be one of the following:

```

CHARACTER [(LEN =] length-value, [KIND =] kind-value)]
CHARACTER [(LEN=] length-value)
CHARACTER [(length-value, [KIND =] kind-value)]
CHARACTER [(KIND = kind-value, [LEN =] length-value)]
CHARACTER*length-value

```

If the kind value is not stated, then the default one will be selected. The only KIND parameter for the character type in INTEL® FORTRAN is 1. If the length is not specified, a length equal to 1 will be selected, i.e. equal to one character.

The delimiters of a character literal constant are either a pair of quotation marks or apostrophes. If it is needed to represent one of these delimiters as a character within the string, several choices are available: using doubled delimiters or switching between the delimiters. The following example elaborates these methods:

```

1  CHARACTER(LEN=40) :: char1
2
3  char1 = 'This is an apstrophe: ' ' '
4  char1 = "This is a quotation mark: " " "
5  char1 = "This is another apstrophe: ' "
6  char1 = 'This is another quotation mark: " '
7  char1 = "These are a pair of apstrophes: ' ' "
8  char1 = 'These are a pair of quotation marks: " " '

```

A special case for the length-value is using an asterisk (\*) which is handy in many cases of which the following cases are more frequent:

- a dummy argument which assumes the length of the associated actual argument, for instance:

```

1  SUBROUTINE MyString (Str1)
2  CHARACTER (LEN = *) :: Str1
3
4  Print *, 'The length of my dummy argument is ', Len (Str1)
5  END SUBROUTINE

```

Note that the intrinsic function Len returns the length of Str1 depending on the actual argument being used.

- a named constant, for instance:

```

1  CHARACTER (LEN = *), PARAMETER :: MYNOTE = 'This is very long not&
2  &e and I have no idea how many characters are needed for this one &
3  &so let 's use an asteriks!'

```

Note that the INTEL® FORTRAN 2003 continuation line style is used; each line embodies an extra ampersand at the end which is not required in a fixed format of the earlier FORTRAN versions.

Another special case for length-value is using a colon (:). For this case, an ALLOCATABLE attribute must be used (Sect. 1.8.6) such as the following:

```

1  CHARACTER (LEN = :), ALLOCATABLE :: AString

```

This statement declares the AString variable with an unknown length. The length can be set using an Allocate statement such as the following example:

```

1  ALLOCATE (AString(60))

```

A logical literal constant is either one of the .TRUE. or .FALSE. values. In INTEL® FORTRAN there are four kinds of logical types: 1, 2, 4 and 8; the number represents the number of bytes used for the storage. Normally, the most compatible kind is selected by default and thus, no changes are necessary when dealing with logical data types.

No matter which kind is used, there are only two values that can be assigned to a logical variable. A stored zero in the variable indicates .FALSE and any non-zero values indicates .TRUE. Obviously, using a single bit would be enough to represent this type. However, for CPU<sup>4</sup> performance reasons the default type is set to KIND = 4. The intrinsic function KIND is usable with logical types but there are no intrinsic functions available to select a logical type.

## 1.7.6 Expressions, Operators and Operands

An *expression* (expr) is usually a calculation, called an *operation*, resulting in a scalar or an array. It is made up of a combination of *operands*, *operators* and parentheses.

---

<sup>4</sup>Central Processing Unit.



**Table 1.18** FORTRAN operators. Adapted from [19]

| Operator  | Responsibility                  | Category     | Precedence |
|---|---------------------------------|--------------|------------|
| User-defined  | Unary defined operator          | User-defined | Highest    |
| **  | Exponentiation                  | Numeric      | .          |
| *, /  | Multiplication, division        | Numeric      | .          |
| +, -  | Unary identity, or negation     | Numeric      | .          |
| +, -  | Binary addition, subtraction    | Numeric      | .          |
| //  | Concatenation                   | Character    | .          |
| .EQ., .NE., .LT., .LE., .GT.,<br>.GE., ==, /=, <, <=, >, >= | Comparison                      | Relational   | .          |
| .NOT.   | Negation (unary)                | Logical      | .          |
| .AND.   | Conjunction                     | Logical      | .          |
| .OR.  | Inclusive disjunction           | Logical      | .          |
| .EQV., .NEQV.   | Equivalence,<br>non-equivalence | Logical      | .          |
| User-defined  | Binary defined operator         | User-defined | Lowest     |

An operand is any scalar or array engaged in an operation. The operation is done by means of operators.

An operator is called *binary* or *dyadic* if it acts on two operands, e.g. addition in  $a+b$ . It is called *unary* or *monadic* if it acts on one operand, e.g. the negative sign in  $-a$ . Some of the commonly-used operators are listed in Table 1.18. In addition, it is possible to define custom operators using operator overloading (see [8]).

### 1.7.7 Derived-Data Types

Although the aforementioned intrinsic data types are usually enough for handling simple tasks, sometimes it is required to define customized data types. This is the case when dealing with complex structures especially in a large number. Analogous to an intrinsic data type, a derived data type has also a name, a set of type parameters, a set of values, a set of operations, and a means to represent the constants. The very first thing to use a derived type, is defining the type. The general syntax for a type definition is:

```

TYPE [[, type-attribute-list] ::] type-name [(type-parameter-name-list)]
    [type-parameter-definition-statement] ...
    [private-or-sequence-statement] ...
    [component-definition-statement] ...
    [procedure-binding-part] ...
END TYPE [type-name]
    
```

An example will help elaborating this syntax: suppose that a new type is needed to represent a node in a finite element code which contains the following:

- the identification number of the node (an integer),
- coordinates of the node (real numbers),
- a field that indicates if the node is assigned to an element or not (a logical variable).

The following can be a data type definition of various possible ones providing the mentioned characteristics:

```

1  TYPE NODE
2      INTEGER :: Id
3      REAL*8  :: X, Y, Z
4      LOGICAL :: Connected
5  END TYPE NODE

```

The NODE is called a *type specifier*. The definition of this new derived type named NODE consists of 5 scalar *components* or *fields*: Id is an integer indicating the identification number of the node, x, y, z are real numbers indicating the coordinates of the node and Connected which is a logical variable declaring the connection status of the node to an element.

Scalar objects of a derived type are usually called *structures* and the statement used for defining the type is called a *derived type statement*. In order to declare a structure of this type, a type declaration statement is used in the following manner:

```

1  TYPE (NODE) :: myNode

```

This line of code declares a variable named myNode of type NODE. There are two ways of assigning values to a type: one is using parentheses as delimiters and commas as separators of the fields which forms a literal constant of this derived type, for instance:

```

1  myNode = Node (1,100.0,50.0,20.0,.FALSE.)

```

The other way is accessing fields separately by means of a *component selector* character, i.e. a percentage special character such as the following:

```

1  myNode%Id = 1
2  myNode%X = 100.0
3  myNode%Y = 50.0
4  myNode%Z = 20.0
5  myNode%Connected = .FALSE.

```

The lines assigning values to the variable are called *assignment statements*. When a type is declared it can be incorporated as a part of another type declaration to produce even more sophisticated types. Suppose that it is needed to declare an element type in a finite element code which consists of two nodes, one declaration can be as the following:

```

1  TYPE ELEMENT
2      TYPE NODE :: FirstNode , SecondNode
3  END TYPE ELEMENT

```

In addition, the type declaration and assignment statements are in the form of the following:

```

1  TYPE (ELEMENT) :: myElement
2
3  myElement = ELEMENT (NODE (1, 10, 20, 30, .TRUE.),
4  &                    NODE (2, 20, 30, 40, .FALSE.))
5  myElement%SecondNode%Connected = .TRUE.

```

Note that although it is possible to define operators for a specific type, there are not any available automatically. An easier way is operating on the fields separately. Namely, depending on the type of the field, the operations corresponding to that type are valid. For example, `myElement1 + myElement2` is not valid but `myElement1%FirstNode%ld + myElement1%SecondNode%ld` is a valid assignment statement.

### 1.7.8 Arrays

Derived types were discussed in the previous section with the main purpose of gathering various types of data in one compound structure. Another case is using several variables of the same type of data with the same specific data parameters (i.e. kind and/or length) in one structure, called an *array*. In other words, an array is a sequence of homogeneous data objects. Each single member of an array is called an *element*. Note that the data type of each element can be in any scalar form, i.e. intrinsic or even a derived type.

From another perspective, an array is a data object with the attribute of `DIMENSION` which enables it to have multiple elements of the same kind. The number of dimensions of an array is called its *rank* and is equal to the number of comma-separated items in the array specification. The maximum allowable number of dimensions is 7 in FORTRAN 90, which has been extended to 15 in FORTRAN 2008.

A familiar example for an array is a mathematical vector. A vector in mathematical terms is a column of real numbers which is representable in FORTRAN by a one-dimensional array of real numbers (rank = 1). Similarly, a mathematical matrix consists of rows and columns of real numbers which can also be represented by a two-dimensional array of real numbers (rank = 2).

The *extent* of each dimension in an array, i.e. the number of elements along that dimension, is defined by its lower- and upper bound. A couple of parentheses are used to specify either the extent or the bounds of the array for each dimension. A colon is used to separate the lower bound from the upper bound. For instance in a one-dimensional array specified by (L:U), L is the lower bound, U is the upper bound and the extent of this dimension is equal to  $U-L+1$ . A bound can be any integer number but if the lower bound is larger than the upper bound, a zero-size array results.

The *shape* of an array is determined by the rank and the extent of array in each dimension. Therefore, the shape can be presented by a vector containing the extent for each dimension, e.g. a shape of the form (3, 4, 5) indicates that the corresponding array has three elements in the first dimension, four elements in the second dimension

**Table 1.19** Terminology of arrays in FORTRAN

| Term                | Description   |
|---------------------|---|
| Array               | A collection of values of the same type and type parameter      |
| Element             | Each individual value of the array                              |
| Array element order | Is the sequence in which array is stored in memory              |
| Dimension           | Analogous to dimension of a matrix in mathematics               |
| Rank                | Number of dimension(s) (equal to the number of indices)         |
| Subscript or index  | An integer used to access each element                          |
| Bounds              | Upper and lower limits of the index in each dimension           |
| Extent              | Number of elements along a dimension                            |
| Size                | Total number of elements in an array                            |
| Shape               | Is determined by the rank and extent of array in each dimension |
| Dummy array         | An array used as an argument of a procedure                     |
| Subarray or section | Any portion of an array ranging from 1 to the size of array     |
| Conformable arrays  | Arrays of the same shape  |

and five elements in the third dimension but it says nothing about the lower- and upper-bounds. Namely, arrays with different bounds may have the same shape. Arrays with the same shape are called *conformable arrays*. Conformable arrays can take part as operands of an operator in an expression.

The *size* of an array is the number of total elements of the array and it can be calculated by multiplying all of the extents of an array, e.g. an array with the shape of (3, 4, 5) consists of  $3 \times 4 \times 5 = 60$  elements.

Each element of an array can be accessed by a *subscript*. A subscript is an integer in the range of the lower- and upper bounds of each dimension. Using any number out of this range raises an *out-of-bound exception*. A summary of the described array terminology is gathered in Table 1.19.

The general syntax for an array is not different than that of a variable. Similar to a variable, an array can have several attributes among which the DIMENSION attribute must be used. Nevertheless, this is not the only way to declare an array. The first syntax for an array is as the following:

```
type-spec, DIMENSION (array-spec) [[, attr-spec]...:] entity-decl-list
```

The second syntax for an array is as the following:

```
type-spec [[, attr-spec]...:] object-name (array-spec) [[, object-name] ...]
```

An array specifier (*array-spec*) determines the type of the array. It is the only part of the array syntax that differs from one array to another. Generally, there are four array specifiers which form four main types of arrays and a fifth special type. These five types of arrays are listed in Table 1.20. As it is evident from this table, the rank of every array must be known prior to the compilation of the code, but the shape can be defined based on the type of array being used.

**Table 1.20** Array forms and their characteristics

| Array Type     | Dummy | Allocatable | Rank | Extent | Bounds | Shape | Size |
|----------------|-------|-------------|------|--------|--------|-------|------|
| Explicit shape | ✓     | ✓           | ✓    | ✓      | ✓      | ✓     | ✓    |
| Automatic      | ✓     | ✗           | ✓    | ✗      | ✗      | ✗     | ✗    |
| Assumed shape  | ✓     | ✗           | ✓    | ✗      | ✗      | ✗     | ✗    |
| Deferred-shape | ✓✗    | ✓           | ✓    | ✗      | ✗      | ✗     | ✗    |
| Assumed size   | ✓     | ✗           | ✓    | ✗      | ✗      | ✗     | ✗    |

The simplest type of arrays is the *explicit array*. For an explicit-shape array, rank, size and bounds are declared prior to the running of the program. The array specifier for an explicit array is the explicit shape specifier with the following syntax:

specification-expr : specification-expr

A *specification expression* (specification-expr) is a restricted case of an expression which results in a scalar integer. It can be an expression consisting of constants or variables. The first and second specification expression of the syntax are separated by a colon. They dictate the lower- and upper bound, respectively; for instance:

```
1 REAL, DIMENSION (-10:10,100) :: array1
```

In this example, `array1` is a two-dimensional array. The lower bound and the upper bound of the first dimension are -10 and 10, respectively. For the second dimension, an extent of 100 is specified which corresponds to lower- and upper bounds of 1 and 100, respectively. The extent of first and second dimensions are 21 and 100 and thus, the shape of this array is (21,100). Note that when the lower bound of an array is not declared, it will be assumed to be equal to one.

As mentioned earlier, it is possible to declare an array without the `DIMENSION` attribute. The same example could have been written using the second syntax:

```
1 REAL :: array1 (-10:10,100)
```

A special case of explicit arrays is encountered within a subprogram where at least one of the array bounds is a dummy argument of that subprogram. This special explicit array is called an *automatic array*. The size and the bounds of an automatic array is defined upon running the subprogram. For example:

```
1 SUBROUTINE ArraySample (a, b)
2   INTEGER, INTENT (IN) :: a, b
3   REAL :: autoArray (a, a + b)
4   ...
5 END SUBROUTINE ArraySample
```

In this example, `autoArray` is a two-dimensional automatic array. The rank is already known before the compilation but the extents of each dimension, and thus, its shape, is only defined within the subroutine upon its invoking. The bounds of the first and the second dimension are 1:a and 1:a+b, respectively.

The bounds of an automatic array can be defined more explicitly. For instance in the following example, the bounds of autoArray are -a and b:

```

1  SUBROUTINE ArraySample ( a, b)
2  INTEGER, INTENT (IN) :: a, b
3  REAL :: autoArray (-a:b)
4  ...
5  END SUBROUTINE ArraySample

```

As it is summarized in Table 1.20, an explicit array can be a dummy argument and an allocatable variable. In addition, the rank, extent, bounds, shape and size of an explicit array is defined before compilation. An automatic array can be a dummy argument but not an allocatable one. Prior to compilation only its rank is known and the other characteristics will be defined upon entering the subprogram.

The third type of arrays is the *assumed shape* array which has the following syntax for the array specifier:

[lower bound] :

The assumed shape array is a dummy argument which takes the shape of the actual argument. The rank is equal to the number of colons used in the specifier; for each colon, an optional lower bound can be specified otherwise the default value is 1. The upper bound will be evaluated upon entering the subprogram by adding the extent of that dimension minus 1. Therefore, the extent, bounds, shape and size of an assumed-array is not known during compilation and in addition, an assumed shape array cannot be of a dynamic type, i.e. an ALLOCATABLE or POINTER one. Consider the following example:

```

1  PROGRAM Main
2  REAL :: anArray (-10:10)
3  ...
4  CALL SumUp (anArray)
5  ...
6  SUBROUTINE SumUp(arr)
7  REAL :: arr(:)
8  ...
9  END SUBROUTINE SumUp

```

In this example, the extent of the actual argument anArray is 21 and the assumed shape array named arr takes its shape. Therefore, upon invoking the subroutine SumUp, the boundaries of arr will be (1:21).

The fourth type is a *deferred-shape* array which is a dynamic array: either an allocatable array with ALLOCATABLE attribute or an array pointer with POINTER attribute. The shape of this array is postponed to run-time of the program, i.e. the bounds, extent and the shape of this array can be assigned during the program based on its requirements. The array specifier for a deferred-shape array consists of a single colon for each dimension. This type is not necessarily a dummy one but it can be. The only defined characteristic is the rank. The bounds, extent, shape and size definition is postponed to the allocation or association time. Consider the following example:

```

1  SUBROUTINE AssignValue (array1)
2  INTEGER, ALLOCATABLE, INTENT (INOUT) :: array1 (:)
3
4  ALLOCATE (array1 (-5:5))
5  array1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
6  END SUBROUTINE AssignValue

```

In this example, an ALLOCATABLE deferred-shape named array1 is allocated and then defined by assigning values 1–11. The bounds, extent, shape and size of the array are defined explicitly. Another example for a deferred-shape array is an array of pointers, such as the following:

```

1  INTEGER, POINTER :: array1 (:)
2
3  ALLOCATE (array1(-5:5))
4  array1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

There are two important options which can be used with the ALLOCATE statement; the SOURCE and the MOLD options. An already-defined array can be used as the source of the array to be defined. In this case, all the elements of the defined array are copied to those of the array to be allocated. For instance:

```

1  INTEGER, POINTER :: array1 (:)
2  INTEGER :: sourceArray(3)
3
4  sourceArray = [1,2,3]
5  ALLOCATE (array1, SOURCE = sourceArray)
```

After executing line 5 of this code, the array1 array will be a defined array with the same elements of the sourceArray. In some cases, it is required to use an array, which is not necessarily defined, as a template to build another one. To do so, the MOLD option can be used:

```

1  INTEGER, POINTER :: array1 (:)
2  INTEGER :: templateArray(3)
3
4  ALLOCATE (array1, MOLD = templateArray)
```

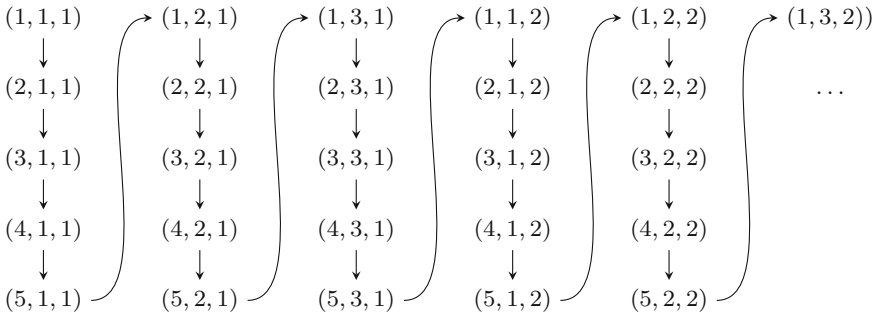
After executing line 4, the array1 array is a declared array but not yet defined. Note that even if the templateArray is defined, the array1 array will not be defined after executing the ALLOCATE statement.

Before investigating the properties of the fifth array type, it is necessary to understand the *array element order*. The array element order is the order in which every element is stored in the memory. The same order is used when intrinsic functions act on the array as a whole.

In FORTRAN, the *column-major order* is used. In this element ordering scheme, accessing to the elements starts by increasing the subscript of the first dimension incrementally from its lower bound to its upper bound. This results in increasing the subscript of the higher dimensions one-by-one. As it is illustrated in Fig. 1.19, the subscript of the first dimension vary more rapidly than the last dimension.

For example consider the case that an array is used in an I/O statement such as a Write statement. The array is passed to the statement without any subscripts and thus, all the elements will be printed. Elements will be printed in the saved sequencing order which is a column-major order. Note that this array element order is specific to FORTRAN and other languages may have a *row-major ordering* such as C/C++.

The order of each array element in the storage sequence can be represented by an integer. It is called the *subscript order value* of that element. The shape of a  $n$ -dimensional array with the explicit specifier  $(l_1 : u_1, l_2 : u_2, \dots, l_n : u_n)$  is



**Fig. 1.19** Column-major ordering for array element order in FORTRAN

$(e_1, e_2, \dots, e_n)$  where for  $i = 1 \dots n$  the extents are  $e_i = u_i - l_i$ . The subscript order value of an element with subscripts  $(s_1, s_2, \dots, s_n)$  can be evaluated with the following formula:

$$1 + \sum_{i=1}^n \left( \left( \prod_{j=1}^{i-1} e_j \right) (s_i - l_i) \right). \tag{1.6}$$

For instance in an array with the explicit specifier of (5, 3, 3), the element (1, 1, 1) has a subscript order value of 1 and the element (1, 3, 2) has an order value of 26.

Accessing an array is more efficient if it is done in the same order in which it is stored. The most efficient way of accessing a three-dimensional array in FORTRAN is, for example, using the following nested loops:

1  
2  
3  
4  
5  
6  
7

```

DO k = 1, l
  DO j = 1, m
    DO i = 1, n
      ! code for array element (i, j, k)
    END DO
  END DO
END DO

```

The fifth type, the *assumed size* array, is a dummy array which takes the size of the actual argument. Therefore, the shape of the actual array is not necessarily preserved. In other words, the actual argument is reshaped into the shape of the assumed size array upon entering the subprogram. During this process, the column-major ordering of the array will be preserved. To be more precise, the rank and extent of every dimension of the assumed size array except the last dimension must be declared. The syntax is as follows:

[ explicit-shape-spec-list, ] [ lower bound : ] \*

The default value for the lower bound of any dimension is 1 and all dimensions except the last one can be explicitly defined similar to what has been done for explicit arrays. Assumed size arrays are very flexible because, for instance, a one-dimensional array can be reshaped to any array with any rank and vice versa. Consider the following example:



```

1 PROGRAM Main
2   INTEGER :: array (2, 4, 2)
3
4   array = Reshape ([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16], [2,4,2])
5
6   CALL SumAndMultiply (array, Size(array))
7
8   CONTAINS
9     SUBROUTINE SumAndMultiply (anArray, arraySize)
10      INTEGER, INTENT (IN) :: anArray (*), arraySize
11      INTEGER :: i, res = 0
12
13      DO i=1, arraySize
14        res= res + anArray(i) * anArray(i)
15      END DO
16      PRINT *, res
17    END SUBROUTINE SumAndMultiply
18  END PROGRAM Main

```

The `Size ()` intrinsic function is one of the array functions which is provided as standard by every FORTRAN compiler (a brief selection of similar intrinsic functions is listed in Table 1.21). This function is used to retrieve the size of an array. However, the `Size ()` function cannot determine the size of an assumed size array. Therefore, the size of an assumed size array must be passed to the subroutine as an argument. In this example, within the subroutine, the originally three-dimensional array is interpreted as a one-dimensional array with the size indicated by the `arraySize` parameter.

The last topic on arrays is their initializing which is done by a powerful tool named the *array constructor*. An array constructor is a mechanism of initializing one-dimensional arrays. It is a combination of literal-scalar constants, arrays and implied-DO specifications which are enclosed in square brackets ([ ]) or parenthesis-slash delimiter (/ /). The syntax of an array constructor is as the following:

```
[ [type-spec ::] [ac-value-list ] ]
```

In the first line of this syntax, the outer brackets are delimiters but the inner brackets are indicators of optional part of the syntax. Alternatively, the following syntax can also be used:

```
(/ [type-spec ::] [ac-value-list ] /)
```

This syntax is the same as the first one except that the delimiters are changed to a couple of parenthesis-slash.

An array constructor value list (*ac-value-list*) is a list of array constructors which can be either an expression (scalar or array) or an *implied-DO loop*. Every expression in the array constructor must have the same type and type parameters. The syntax of an implied-DO loop is as the following:

```
( ac-value-list, ac-do-variable = scalar-int-expr, scalar-int-expr [, scalar-int-expr] )
```

An array constructor DO-variable (*ac-do-variable*) is an integer variable with a local scope within the constructor. Several implied-DO constructs can be nested but additional DO-variables must be introduced. Similar to a normal DO construct, three scalar integer expressions (*scalar-int-expression*) are used for initial, final and the step

**Table 1.21** Selected standard intrinsic subprograms for arrays in FORTRAN

| Function    | Task   |
|-------------|--|
| All         | Returns .TRUE. if all elements contain the .TRUE. value                      |
| Any         | Returns .TRUE. if any of elements contain the .TRUE. value                   |
| Count       | Returns the number of .TRUE. array elements                                  |
| Cshift      | Circular shift of the array elements   |
| Dot_product | Dot product of two matrices  |
| Lbound      | Lower dimension of an array  |
| MatMul      | Multiplication of matrices   |
| Maxloc      | Location of the maximum value within in array                                |
| Maxval      | Maximum value of an array  |
| Merge       | Selection of values under the control of a mask                              |
| Minloc      | Location of the minimum value of an array                                    |
| Minval      | Minimum value of an array  |
| Pack        | Pack an array into an array of rank one considering a mask                   |
| Product     | Product of array elements  |
| Reshape     | Reshapes a one-dimensional array to a desired shape                          |
| Shape       | Determines the shape of an array   |
| Size        | Determines the size of an array  |
| Spread      | Adds a dimension to an array   |
| Sum         | Sum of array elements  |
| Transpose   | Transpose an array of rank two   |
| Ubound      | Upper dimension of an array  |
| Unpack      | Store the elements of a vector in an array of higher rank considering a mask |

value, correspondingly. The following example is for a scalar array constructor which is filled with three scalar values:

```
1 REAL :: storage(10) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The following example uses an implied-DO structure to fill the array with the same values as the previous example:

```
1 REAL :: storage(10) = [(i, i=1, 10)]
```

A nested implied-DO structure is also possible:

```
1 REAL :: storage(10) = [((i*j, i=1, 5), j=1, 2)]
```

An array expression is also possible in an array construct:

```
1 REAL :: storage(10) = [part1(1:5), part2(1:5)]
```

A combination of these three forms plus other intrinsic functions gives even more flexibility in constructing arrays:

```
1 REAL, PARAMETER :: part(5) = [1, 2, 3, 4, 5]
2 REAL :: storage(10) = [part(1:5), (Sqrt(i + part(i)), i = 1, 4), 0.0]
```

In this example, a part of a constant array named `part` is used in another array named `storage`. Array constructors can be used both in the variable declaration part of the program and the execution part of it. However, if used in the declaration part, only constant arrays are allowed in an array constructor.

In addition, there is a *subscript triplet* form which is equivalent to the implied-DO construct. For example the following two examples are equivalent:

```
1 REAL :: storage(10) = [1:9:2]
2 REAL :: storage(10) = [(i, i=1, 9, 2)]
```

In contrast, initializing the arrays with higher ranks is done with the `Reshape` intrinsic function. It reshapes the first argument, an array of any rank, in the shape of the second one such as in the following example:

```
1 REAL :: a(2,4) = reshape([1, 2, 3, 4, 5, 6, 7, 8], [2, 4])
```

In addition to previous methods, a very powerful way of selecting arbitrary elements of an array is using a *vector subscript* which is a rank one integer array containing the indices of the selected elements. For instance to select the elements 1, 5 and 7 of the array `anArray`, the following code can be used:

```
1 REAL :: anArray(10) = [(2.0*i, i=1,10)]
2 INTEGER :: selection(3) = [1,5,7]
3
4 PRINT *, anArray(selection)
```

A vector subscript is merely a container of the selected indices which are referring to the parent array. It is also possible to use a vector subscript to select specific elements of a high rank parent array; consider the following lines of code for this case:

```
1 REAL :: anArray(2,5) = reshape([(2.0*i, i=1,10)], [2,5])
2 INTEGER :: selection(2) = [1,4]
3
4 PRINT *, anArray(:, selection)
```

The output of this code will be the values of the elements (1, 1), (2, 1), (1, 4) and (2, 4) of the array `anArray`.

## 1.8 Data Attributes

As mentioned earlier, attributes make type declarations more specific. They are not only used in type declarations but also subprogram declaration statements. For instance, to save the value and type of a variable within a subprogram the `SAVE` attribute is used. It is worth mentioning that an entity cannot possess all the available attributes because not all of them are compatible with each other. For instance, the `PARAMETER` attribute cannot be used together with the `INTENT` attribute. A number of attributes are briefly listed in Table 1.22.

**Table 1.22** List of common attributes in a declaration construct

| Attribute                 | Used for...  |
|---------------------------|--|
| PARAMETER                 | Declaration of named constants                             |
| DIMENSION                 | Declaration of arrays                                      |
| DATA                      | Initializing variables                                     |
| COMMON                    | Specifying a shared entity between different program units |
| SAVE                      | Retaining value of variables                               |
| ALLOCATABLE or POINTER    | Declaration of dynamic entities                            |
| TARGET                    | Specifying that a variable can be targeted by a pointer    |
| INTENT, VALUE or OPTIONAL | Specifying the nature of a dummy argument                  |
| PUBLIC or PRIVATE         | Specifying the accessibility of an entity within a module  |

### 1.8.1 PARAMETER Statement

Using named constants saves a lot of time when dealing with several literal constants. Usually when a literal constant has a specific meaning, it is used in multiple places of a code. In such cases, it is better to use a named constant instead of the literal constant, because at future points in time, it might be necessary to change this specific number and it is not practical to do it one-by-one. For example, the number of total employees in a company will be used in several locations of the code to prepare the data environment and carry out the calculations. It is better to use a named constant to hold this total number.

The other application of the named constants is for scientific constants, e.g. the number  $\pi = 3.141592\dots$ , or Napier's constant  $e = 2.71828182\dots$  etc. Because one may need to change the number of significant digits of a scientific constant, it is advised to use a named constant to hold the constant. Additionally, named constants increase the readability of the code.

In order to define a named constant the `PARAMETER` attribute is used.<sup>5</sup> The syntax for the `PARAMETER` statement, which can be used in a statement specification, is as the following:

```
PARAMETER (named-constant = initialization-expr, [named-constant = initialization-expr] ...)
```

The following examples are with an entity-oriented declaration:

```

1 REAL, PARAMETER :: PI = 3.14
2 REAL (KIND = 8), PARAMETER :: NAPIER = 2.7182818284590
3 DOUBLE PRECISION, PARAMETER :: COMBO = NAPIER*PI
4 CHARACTER (LEN = *) :: TEXT = 'Welcome aboard!'
5 INTEGER, PARAMETER :: ELEMENTS = 100, NODES = 200
6 CHARACTER (LEN=1), PARAMETER :: NUMBERS (0:9)
7 &= ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
8 REAL, PARAMETER, DIMENSION(3) :: MYDATA = (/PI, PI*2, PI*3/)

```

<sup>5</sup>In FORTRAN, a `PARAMETER` attribute is used to define a named constant. However, as mentioned earlier, in general programming terminology a parameter is an argument of a subprogram.

Note that it is possible to use the defined constants as a parameter for the consequent ones. Initializing the arrays is possible using either a couple of brackets [ ] or (/ /) as delimiters. Note that both DIMENSION and PARAMETER attributes are used after the type name, emphasizing more on the named constant itself which is more preferred. However, the same result can be obtained with the help of an attribute-oriented approach. For instance, the same array is defined as the following:

```

1      REAL :: PI
2      PARAMETER (PI = 3.14)
3
4      REAL :: MYDATA
5      DIMENSION MYDATA(3)
6      PARAMETER (MYDATA = (/PI, PI*2, PI*3/))

```

It is possible to declare constants after variable declarations by using an entity-oriented approach. However, it is recommended that all the constants are defined before variable declarations and after the implicit statement. This makes the program clearer and readable. In addition, some constants may be used later in the declaration of other variables.

### 1.8.2 PUBLIC Versus PRIVATE

The most powerful capability of modules is encapsulation (packaging) of entities within themselves which allows the programmer to protect unnecessary information from the user; PUBLIC and PRIVATE attributes are the tools for the job. An entity with a PUBLIC attribute can be accessed from outside of the module whereas a PRIVATE attribute allows access only for the internal subprograms of the module. By default, every entity is considered a public one unless the PRIVATE keyword is used. For instance:

```

1      MODULE MyModule
2          INTEGER :: a
3
4          PRIVATE
5          INTEGER :: b
6      END MODULE MyModule

```

In this example, the default access is private and thus, both a and b variables are considered as private ones. Only one explicit declaration of the default access is permitted; either PUBLIC or PRIVATE must be used. Consider the following:

```

1      MODULE MyModule
2          INTEGER, PRIVATE :: a
3
4          TYPE, PRIVATE :: aType
5              CHARACTER (10) :: Name
6              INTEGER :: ID
7          END TYPE aType
8
9          REAL :: anArray (1:100)
10
11         PUBLIC CalcSum
12         PRIVATE InsideCalc
13

```

```

14 CONTAINS
15     SUBROUTINE CalcSum
16     ...
17     END SUBROUTINE CalcSum
18
19     FUNCTION InsideCalc
20     ...
21     END FUNCTION InsideCalc
22 END MODULE MyModule

```

In this example, none of the keywords `PUBLIC` or `PRIVATE` are used. Therefore, all entities are public by default. As a result, the array named `anArray` and the `CalcSum` module subroutine are public entities. In contrast, the variable named `a`, the derived type `aType` and the module function `InsideCalc` are private entities.

### 1.8.3 *SAVE and COMMON Attribute*

Upon entering a subprogram, local variables are undefined. Their status can be changed by assigning a value. However, after finishing the execution of the subprogram and upon exiting the subprogram, the local variable becomes undefined again. Therefore, in every invocation of the subprogram, the variable has to be initialized again unless the `SAVE` attribute is used.

The `SAVE` attribute preserves the defined status of the variable even after the subprogram is executed. For the next execution of the subprogram without any initialization, the value of the variable will be saved. For instance, if the last used value of a variable will be needed within the subprogram to calculate the next one, this attribute can be used.

This attribute is quite convenient especially if it is used within a module. Such variables can be shared between several subprograms using a common module. In older FORTRAN programs, *common blocks* were more frequently utilized to carry out the same task.

Generally, data entities are passed to a subprogram via its arguments. Alternatively, the data can be shared among various subprograms by a `COMMON` statement. This attribute allows the blocks of data to be considered global, namely they are defined in several subprograms. This approach is against the encapsulation concept. However, it is still in use in some old FORTRAN codes and particular to our attention in the subroutines of MARC/MENTAT (see Sect. 2.3.3). In such inevitable cases, there is no point passing a ubiquitous piece of data as an argument. Furthermore, there will be no harm in sharing a few frequently-used blocks of data.

By using the `COMMON` statement, the data to be shared between program units will be placed in *common blocks*. A common block without a name is called a *blank block* and if it is used with a name, it is called a *named common block*. Note that

only for a named common block, the SAVE attribute can be applied and thus, it is recommended to use a named common block instead of a blank one.

The syntax for a COMMON statement is as the following:

```
COMMON [ / [common_block_name] / ] common_block_object_list
[ [,] / [common_block_name] / common_block_object_list ] ...
```

common\_block\_name is the name of the common block which is enclosed between two slashes whereas using just two slashes (//) will indicate a blank common block. common\_block\_object\_list is the list of named variables with exception of the PARAMETER attribute, dummy arguments, allocatable arrays, automatic objects and function names.

The same syntax must be repeated in every subprogram in which it is intended for the data to be shared. Note that the sequence of data must be kept the same. Therefore, it is good practice to put all the statements regarding the common blocks in a single text file and include the file using the INCLUDE line. The include statement substitutes the containing lines of the included file with itself. The syntax for include is as follows:

```
INCLUDE 'file-name'
```

The keyword INCLUDE is followed by a string which is the file to be included (file-name). The file used for a common block usually has a .cmn or .cbl extension. Consider the following named common block:

```
1 LOGICAL :: myFlag
2 REAL, DIMENSION (10) :: myArray
3 CHARACTER (12) :: message
4 COMMON /SharedData/ myFlag, myArray, message
5 SAVE /SharedData/
```

To preserve the same sequence of data in the common block, i.e. myFlag, myArray, message, these lines are saved in a text file named, say Myblocks.cmn. The following line is used instead of the previous listing whenever the data is needed:

```
1 INCLUDE 'MyBlocks.cmn'
```

The access to a common block is provided using a *storage association*. Namely, one memory location is associated with several variable names. In other words, it is possible to rename the variables of a common block. However, it is not recommended.

One limitation of using common blocks is not being able to initialize the variables. The solution is using a BLOCK DATA structure to hold and initialize the common blocks. BLOCK DATA is a program unit which is mainly used for initialization of the variables in a common block. The syntax is as the following:

```
BLOCK DATA [block-data-name]
[specification-stmt] ...
END BLOCK DATA [block-data-name]
```

The DATA attribute is used to initialize the values as in the following example:

```
1 BLOCK DATA
2 LOGICAL :: myFlag
```

```

3     REAL, DIMENSION (10) :: myArray
4     CHARACTER (12) :: message
5     COMMON /SharedData/ myFlag, myArray, message
6     DATA myflag /.FALSE./
7     DATA messege /'I'm defined!'/
8     END BLOCK DATA

```

A block data can be placed anywhere in the code except inside of any other program unit. Initializing the variables of a data block will occur prior to the execution of the main program.

The other restriction of common blocks is not supporting dynamic arrays. This issue can be easily resolved by replacing the common block with a module. Generally, using a module to solely hold the common data is preferred to the common blocks.

### 1.8.4 DATA Statement and Explicit Initialization

*Explicit initialization* is specifying the initial value of a non-pointer variable or dis-association of a pointer. An explicit initialization is done only once [1].

The usual way of initializing a variable is done via its entity declaration. In each scoping unit, this initialization is done only upon the first entry. Namely, a scoping unit such as a subroutine may be accessed several times during the execution of the program but the initialization is done only once in the first entry.

Another alternative for initialization is using an assignment statement. However, the result of this approach is not identical to the previous one. Since the assignment statement is executed upon every entry to the scoping unit, the initialization is done every time.

Note that for a main program explicit initialization using either methods produces the same result, because the main program is executed only once. Both methods are described in the following paragraphs.

The DATA attribute can be used in the declaration of a variable for its initialization. The important point of using a DATA attribute is the fact that it contains an implicit SAVE attribute. Therefore, in the following executions the latest value of the variable will be saved. It means that the initialization will not repeat in every execution. This facility can be used to detect the number of execution, for example:

```

1     SUBROUTINE RunMe
2         INTEGER :: runCount
3         DATA runCount / 1 /
4
5         Write (*,*) 'Current subroutine run = ', runCount
6         runCount = runCount + 1
7     END SUBROUTINE RunMe

```

In this listing, the runCount variable is initialized once by the DATA attribute with the value of 1. A couple of slashes (/) are used as delimiters for a DATA attribute.

Alternatively, in the declaration of a variable the following syntax can be used for initialization:



= initialization-expression

For instance, the previous listing can be rewritten as the following:

```

1  SUBROUTINE RunMe
2      INTEGER :: runCount = 1
3
4      Write (*,*) 'Current subroutine run = ', runCount
5      runCount = runCount + 1
6  END SUBROUTINE RunMe

```

In this listing, the runCount variable is initialized only upon the very first execution of the RunMe subroutine. The second way of initialization has the same characteristics of the DATA attribute but it is done in the variable declaration statement.

The second method is using an assignment statement. If it was required to reinitialize the runCount variable in every execution, the following code should have been used:

```

1  SUBROUTINE RunMe
2      INTEGER :: runCount
3
4      runCount = 1
5      Write (*,*) 'Current subroutine run = ', runCount
6      runCount = runCount + 1
7  END SUBROUTINE RunMe

```

Obviously, in this listing the runCount variable cannot be used to hold the number of executions.

### 1.8.5 INTENT and OPTIONAL Statement

The INTENT attribute is used for the arguments of a subprogram. An argument can be defined as read-only by using the (INTENT (IN)) statement. An output (INTENT (OUT)) statement is used for the arguments which require to be evaluated. Furthermore, it is possible for an argument to be read and written, i.e. they act at the same time as an input and an output (INTENT (INOUT)). The INTENT statement helps the compiler to optimize the code.

The OPTIONAL attribute is used to indicate that the argument is an optional one. An optional argument may or may not be used during the referencing of the subprogram. For instance, consider the following subroutine:

```

1  SUBROUTINE CalcValues (a, b, c)
2      INTEGER, INTENT(OUT) :: a
3      INTEGER, INTENT(IN) :: b
4      INTEGER, INTENT(IN), OPTIONAL :: c
5
6      IF (Present (c)) Then
7          Write (*,*) 'Optional argument exists '
8      ELSE IF
9          Write (*,*) 'No optional argumenta '
10     END IF

```

Using the OPTIONAL attribute, indicates that there is no need for the argument to be always present, i.e. it can be used optionally. The intrinsic function Present is used to

detect whether the optional argument is present or not. It returns a TRUE value if the optional argument *c* is used for invoking the subroutine, otherwise the return value is FALSE. Using the Present intrinsic function along with the OPTIONAL attribute is a good way of covering complicated argument situations.

### 1.8.6 ALLOCATABLE, POINTER and TARGET

The storage memory for a *static data entity* is dedicated based on the type of data at the beginning of a program whereas for a *dynamic data entity* the memory can be allocated, deallocated and even reallocated at any time during the execution of the program. The ALLOCATABLE attribute is used to define dynamic arrays and will be discussed in Sect. 1.7.8.

On the other hand, the POINTER attribute can be assigned to any type of static data to make it dynamic. Before defining a dynamic entity, the required memory space must be allocated. There are two ways to prepare the required memory: the first way is allocating the memory space using an ALLOCATE statement. This statement allocates a new space in memory to the pointer object. The syntax for the ALLOCATE statement is as the following:

```
ALLOCATE (allocation-object [allocate-shape-spec] [, STAT = stat-variable])
```

The allocation object (allocation-object) is either a pointer or an allocatable array, and the shape specifier is used to specify the boundaries of the array. The result of the allocation is returned by the optional state variable (stat-variable). It is set to zero for a successful execution.

Opposite to the ALLOCATE statement is the DEALLOCATE statement which deallocates the memory of a dynamic data entity with a similar syntax:

```
DEALLOCATE (allocation-object [allocate-shape-spec] [, STAT = stat-variable])
```

The disassociation can be done by means of a NULLIFY statement as well:

```
NULLIFY (pointer-object-list)
```

The pointer object list (pointer-object-list) is a list of pointer objects separated by commas.

The second way to allocate objects is by using a *pointer assignment statement* by which the pointer will use the memory allocated to another static entity; called the pointer's *target*. A pointer target can be a data object or a part of a data object declared with the TARGET attribute. The syntax for a pointer assignment statement is as the following:

```
pointer-object => target
```

A pointer object (pointer-object) is a variable with the POINTER attribute. The target of a pointer (target) is a variable with the TARGET attribute which matches the pointer in terms of type, type parameter and rank.

A pointer can have different definitions and allocation status during the execution of a program, i.e. a pointer can be defined or undefined and associated to a target or disassociated from a target. The `SAVE` attribute could be used to preserve the association status of a pointer especially when returning from a subprogram. To be more precise, consider the following lifetime scenario for a pointer:

1. If a pointer is declared without any initialization, its status is *undefined* and *disassociated*, for example:

```
1  INTEGER, POINTER :: aPointer
```

However, to make the disassociation official, a better approach is to initialize the pointer with a null target such as the following:

```
1  INTEGER, POINTER :: aPointer => Null ()
```

Or it is even possible to nullify it after declaration anywhere in the code:

```
1  INTEGER, POINTER :: aPointer
2
3  NULLIFY (aPointer)
```

Nullifying a pointer or associating a pointer with a null object makes the pointer disassociated. The association status of a pointer can be tested by the intrinsic function `Associated` which returns a `.TRUE.` value for an associated pointer.

2. If an `ALLOCATE` statement is used for the pointer then a fresh space in the memory will be appointed to it. In other words, the `ALLOCATE` statement sets the target of the pointer to be a new memory address without any values in it. Allocation of a pointer creates a data object with implied `TARGET` attribute. Now the status is associated but still undefined.
3. It is possible to disassociate a pointer at any time using a `NULLIFY` statement; the status will be disassociated and undefined. Associating a pointer to another object also disassociates the pointer from the old object.
4. If a value is assigned to the pointer, corresponding to its type, then the pointer status will be associated and defined. It is not possible for a pointer to be defined without being associated. As mentioned earlier, one way of association is using the `ALLOCATE` statement and the other one is using the pointer assignment statement which points to a non-null target.

Pointers are usually used to create dynamic linked lists. For example, in the following listing a linked list of the type `item` is created and filled with values:

```
1  TYPE item
2     REAL :: value
3     TYPE (item), POINTER :: nextItem
4  END TYPE
5
6  TYPE (item), POINTER :: myList, currentItem
7  INTEGER :: i
8
9  ALLOCATE (myList)
10 myList%nextItem => myList
11 currentItem => myList
12
13 DO i = 1, 10
14     ALLOCATE (currentItem%nextItem)
```

```

15         currentItem%value = 1.25*i
16         currentItem=>currentItem%nextItem
17     END DO
18
19     currentItem => myList
20
21     DO i = 1, 10
22         Print '(F5.2)', currentItem%value
23         currentItem=>currentItem%nextItem
24     END DO

```

A list of type definitions can be created using arrays as well. However, using pointers to make such a list has its own advantages. For instance, adding/deleting a middle element in a linked list is much easier than in the case of an array in terms of computational expenses. In addition, usually for an array a big part of memory must be allocated by a good guess of the number of required elements whereas in a linked list, the memory is allocated upon adding a new link. The disadvantage of a linked list is having a sequential access, namely a random access to a middle element is not possible without going through the previous elements.

### 1.8.7 CRAY Pointer

The modern type of pointers, which were introduced in the previous subsection, were first introduced in FORTRAN 90. In order to compromise this deficiency in earlier versions, a non-standard extension was applied to incorporate a special form of pointers similar to those of the C programming language, called *CRAY pointers*.

The CRAY pointers were first introduced in FORTRAN 77 and they can still be traced in some old pieces of code. It is not quite easy to translate the old codes written using CRAY pointers to the new type. Therefore, they still exist in some subroutines of MARC/MENTAT.

Because a CRAY pointer is nothing but an integer holding a memory address, it can be realized that the whole point of using CRAY pointers is accessing absolute memory locations. It is a rather advanced tool to access a memory storage which is completely managed by the user. Also creating linked lists with CRAY pointers are not possible because a CRAY pointer cannot point to another one.

Although a CRAY pointer is an integer, explicit declaring is not advised. A better approach is using a CRAY pointer statement with the following syntax:

```

POINTER (pointer-name, pointee-name [ (array-spec) ]) [, (pointer-name, pointee-
name [ (array-spec) ]) ] ...

```

A pointer name (*pointer-name*) is the variable which contains the address for the *pointee* (*pointee-name*). Note that the CRAY pointer statement is used together with other variable declarations and it is not an action statement. A CRAY pointer holds, like other pointers, the address of a variable. Although the CRAY pointer is genuinely an integer, it cannot be declared explicitly as an integer. This is due to the fact that an integer has a variable size in different machines. Therefore, to support the portability

of the code, it is advised to avoid declaring a CRAY pointer as an explicit integer variable. The pointee is the variable to which the pointer points and can be either a scalar variable or an array. The array can be an explicit or an assumed size but an assumed shape or a deferred-shape array is not allowed. An array which acts as a pointee is called a *pointee array* [7].

The compiler will not allocate any space for the pointee of the CRAY pointer. Namely, it is the responsibility of the programmer to ensure the validity of the address of the pointer by means of one of the following:

- by allocating a portion of dynamic memory by calling the Malloc intrinsic function,
- or by setting the pointer to the address of an existing block of data by the Loc intrinsic function by which the address of a variable is returned.

The pointee may have its type declared before or after the pointer statement, and its array specification (if any) may be declared before, during, or after the pointer statement.

Pointer arithmetic is valid with CRAY pointers, but it is not the same as the C pointer arithmetic. CRAY pointers are just ordinary integers, so the user is responsible for determining how many bytes to add to a pointer in order to increment it. Consider the following example:

```

1      REAL :: storage(100)
2      REAL :: indicator(5)
3      POINTER (pIndicator, indicator)
4
5      storage = [(i, i=1,100)]
6      print *, storage
7
8      pIndicator = loc (storage)
9      indicator = 0.0
10     print *, storage

```

In the first three lines of this listing, a CRAY pointer named `pIndicator` is declared by a CRAY pointer statement with a template consisting of a one-dimensional array of five real numbers named `indicator`. Note that no explicit declaration is used for the variable named `pIndicator` and the pointee is just a link to access the data where the pointer points and thus, does not occupy any memory itself; this is unlike any ordinary array declarations.

In other words, up to now only the link between a pointer and pointee is set up and the pointee just declares what the pointer can point to. Therefore, in order to make the pointee usable, it is required to allocate a part of the memory and set the pointer to its address or just set the pointer to a variable already declared. Up to now, it is only specified that our pointer has the potential to point to an address which holds five real numbers and nothing more.

In lines 5 and 6, an array constructor is used to fill the `storage` with numbers 1–100 and print these values. Line 7 puts the address of `storage` to the CRAY `pIndicator` and the next line puts zero to the pointee which is now, the five first elements of the `storage`.

Since using CRAY pointers is complicated, it is better to avoid them as much as possible and replace them with normal FORTRAN 90 pointers. The following points are a brief comparison between CRAY pointers and normal FORTRAN pointers:

- A CRAY pointer is nothing else but an address whereas a normal FORTRAN pointer is an attribute for a pointer variable.
- Arithmetic operations can be used on CRAY pointers but may cost them their portability whereas a normal pointer includes all the information needed to access the target; no extra manipulations are required.
- Assigning a pointee to a CRAY pointer just changes the way that pointer deals with its objects. The value of the object must be assigned to the pointee and not the pointer. However, assigning a value to a normal pointer changes the value of its target. Also changing the target is done by a pointer association expression.
- CRAY pointers are mainly used for a dynamic memory management for which the programmer is responsible whereas normal pointers are usually used for linked lists and allocatable data entities are used for dynamic memory allocation.
- In comparison to CRAY pointers which should be used with extra care, using normal pointers is a more conservative way of dealing with dynamic entities. For instance when a CRAY pointer is associated to various pointees then the role of the pointees will be more like an alias to the variable; consider the following:

```

1      REAL :: a, b, c
2      POINTER (pReal, a), (pReal, b)
3
4      pReal = loc (c)
5      a = 1.0
6      b = 2.0
7      c = 3.0
8
9      print *, a
10     print *, b
11     print *, c

```

The output of this listing is printing the number 3.0 three times, because assigning a value to either of variables a, b or c will set the other two to the same value.

### 1.8.8 Interface Block

The header of a subprogram is the subprogram name and its arguments. The return type for a function can be specified in the header. The signature of a subprogram consists of its header and the declaration of the arguments.<sup>6</sup>

The interface of a subprogram holds its signature and thus, dictates its allowable referencing form. An *implicit interface* is the case of not providing the signature to the referencing program. For instance, an interface for a module subprogram is not required and thus, the interface is implicitly specified.

---

<sup>6</sup>Note that depending on the programming language, the return type may not be a part of the signature. However, it is not of a concern in this case.

In contrast, it is required to explicitly specify the interface in some cases, e.g. for external subprograms, when optional arguments are used, or for array-valued functions. The interface of such cases is called an *explicit interface*.

Using an explicit interface ensures that the compiler does not mistake a user defined subprogram with an intrinsic subprogram. In addition, the compiler can check the consistency between the actual and dummy arguments.

The signature of the subprogram is specified in the *interface body* (interface-body) of an interface block. The typical syntax is as the following:

```
INTERFACE
  interface-body
END INTERFACE
```

For instance, in the following example an external subroutine and an external function are defined with their arguments. Note how the `INTENT` attribute is used to define a clear informative signature:

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

```
INTERFACE
  SUBROUTINE SwapValues (a, b)
    INTEGER, INTENT(INOUT) :: a, b
  END SUBROUTINE SwapValues

  FUNCTION CalcSum (anArray)
    REAL, INTENT(IN), DIMENSION(10):: anArray
    REAL :: CalcSum
  END FUNCTION CalcSum
END INTERFACE
```

Using explicit interfaces is mandatory in many cases. They help the compiler and increase the readability of the program as well. However, the same modifications applied to the signature of the subprogram must be reflected in the interface too. To avoid this, it is better to change the external subprogram to a module subprogram. Consequently, the interface will be changed to an implicit interface.

In any case, if an explicit interface is used, it is recommended to be located after the `IMPLICIT` statement. Then a quick glance will be enough to understand which external procedures will be used in the current program.

Because an interface block is a separate scoping unit, it does not have access to its host. Such an access is sometimes required, e.g. an interface block wants to access a derived type of the host. This access is granted using the `IMPORT` statement with the following syntax:

```
IMPORT [[:] import-name-list]
```

If the `IMPORT` statement is used without any optional names (`import-name-list`) then all the entities will be accessible. For instance, a subroutine is defined by the following interface which needs access to a derived type named `NODE`:

1  
2  
3  
4  
5  
6  
7  
8

```
PROGRAM MyProgram
  TYPE NODE
    INTEGER :: Id
    REAL*8 :: X, Y, Z
    LOGICAL :: Connected
  END TYPE NODE

  INTERFACE
```

```

9      SUBROUTINE ReadNode (aNode)
10     IMPORT
11     TYPE(NODE), INTENT (IN) :: aNode
12     ...
13     END SUBROUTINE ReadNode
14 END INTERFACE
15 ...

```

## 1.9 Input and Output Management

In the course of the execution of a program, most of the time is spent to deal with the I/O operations. In contrast, the arithmetic and the memory access operations of a program are quite fast procedures. Hence, special attention must be paid to this to improve the performance of the program.

It is possible to execute the program while dealing with the I/O operations in parallel. It is called an *asynchronous I/O*. Although buffered data transfer is provided in FORTRAN, in most cases the normal I/O transfer is used which is the focus of the current section.

In FORTRAN, the following standard statements are used for I/O actions:

- OPEN and CLOSE file connection statements,
- READ input statement,
- PRINT and WRITE output statements,
- BACKSPACE, ENDFILE and REWIND file position statements, and
- INQUIRE statement for file connection inquiries.

Transferring data to/from a file requires a connection to a *file unit*. Connecting a file to a unit is done by means of an OPEN statement. After establishing a connection, other I/O action statements can be used. In the end, the unit must be disconnected using a CLOSE statement. Note that since the INQUIRE statement can be executed at any time, it is an exception to this.

The general form of every I/O statement is as the following:

statement-name ( specifier-list ) [variable-list]

All the I/O statements use a list of specifiers (specifier-list) and some of them may require an additional list of variables (variable-list). The general form of a specifier is as the following:

[ keyword = ] expression

Some specifiers are common between I/O statements, e.g. the UNIT specifier, whereas some other ones are usable only in specific statements, e.g. the FORMATTED specifier.

From another perspective, a specifier can be used in two ways: either a value is assigned to it to control the execution of the statement or a value is returned via a variable as the result of the execution. However, regardless of the type, in a single statement, a specifier is permitted to be used just once.



### 1.9.1 Files, Records and Positions

There are two types of files in FORTRAN: an *internal file* which is basically a variable occupying a part of the memory or an *external file* which is saved on a storage media such as a hard-disk. As a precaution to any unforeseen software crashes, the best approach is to write the invaluable computed data in an external file instead of an internal one. However, keeping it to the safe side is accompanied by a drawback: it slows down the execution of the program.

From another perspective, a file is a sequence of data parts, called *records*. All of the records in a file are either *formatted* or *unformatted*. A formatted record consists of a sequence of characters whereas an unformatted record contains the data as it is stored in the memory. Thus, working with such records does not require any conversion which leads to a faster interaction. On the other hand, formatted records are readable as text files and more portable than the unformatted ones.

The file may end with an *end-of-file* record. The end-of-file record is a processor-dependent record which marks the end of the file. This record can be written explicitly by using the ENDFILE statement or implicitly by using the REWIND, the BACKSPACE or the CLOSE statement.

The main issue of I/O management is working with external files to deal with a huge amount of data. There are two methods of accessing a file: *sequential* and *direct* access. The connection to a file can support one of these methods at a time. However, a file may be accessible by both of the methods. To change the access method, the file must be disconnected and then reconnected with the new access method.

In a sequential access the data transfer begins at the beginning of the file and proceeds record-by-record towards its end. Therefore, no sudden jumps to specific records are possible. The most important property in this kind of access is that the last record written to the file is actually the last record in the file. For example, consider a file with several records. If a record is written to the beginning of this file, that record will be the only record on the file and other records will be discarded. In other words, a sequential written record truncates the following ones.

In contrast, in a direct access, selection of a specific record is possible by its number and thus, the data transfer can occur in any arbitrary order.

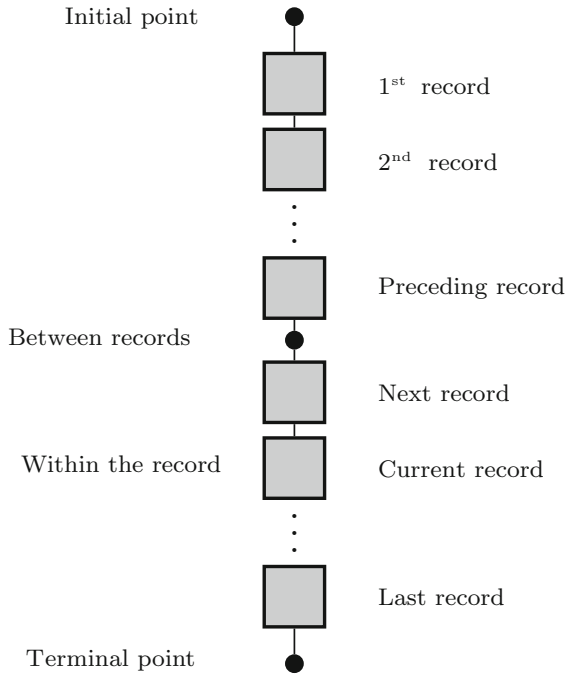
A formatted file with equal record lengths can be accessed by either one of the mentioned methods.

During the course of a data transfer, a *position* holds the place which undergoes the I/O action and then due to the same action, the position is updated to a new one. If an error occurs, the position of the corresponding file will be indeterminate and unreliable to continue the transfer. The best approach to resolve the issue is disconnecting and reconnecting the file.

A schematic presentation of positions in a file is illustrated in Fig. 1.20. The point before the first record is the *initial point* and the one after the last record is the *terminal point*. For a new file without any records, these two positions are the same.

The position in a formatted file acts in two different manners: *non-advancing* and *advancing*. In the advancing approach, after operating on a record, the position of

**Fig. 1.20** Various positions with respect to records of a file



the file will be located after the finished record and before the fresh record, called *preceding* and *next* record, respectively.

In contrast, in a non-advancing approach, the position stays within the same record until the operation is completely finished (end-of-record or end-of-file status); the record is called the *current* record in this case. To be more specific, the position is moving forward within the current record in a character-by-character base. If the position within the record is just after its last character, an attempt to read will lead to an end-of-record situation and if the record is the last one, one more attempt will lead to an end-of-file situation.

### 1.9.2 Connection Statements

The OPEN statement is the general way of connecting an external file to a file unit. However, there may be pre-connected external files which do not require establishing a connection.

In addition, an OPEN statement can be used to change the properties of an existing connection. However, bear in mind to avoid connecting a file which is already connected to a unit, to another one. Only one file at a time can be connected to a unit. By connecting a file to a unit, that file is made available globally throughout the program

via the unit number. Opening a non-existing file creates a new file automatically. The syntax of an OPEN statement is as the following:

OPEN (connect-spec-list)

The connection specifier list (connection-specifier-list) facilitates opening of the file by means of various specifiers as listed in Table 1.23. The following notes on connection specifiers are worth mentioning:

- Generally, FORTRAN tries to handle the mistakes of a programmer as much as possible. Consider, for example, the case that a file is currently connected to a unit and without using a CLOSE statement, the programmer connects another file to the same unit. The result of this act is not catastrophic: since FORTRAN closes the current file and connects the new file automatically. However, this is not an advised approach. It is even possible to write in a file either without a connection, or without a FILE specifier. In this case, FORTRAN creates a fort.n file (n is the unit number) and writes the data as a safe approach to avoid the loss of possible valuable data. Again, this is not a recommended practice.
- A unit number is a non-negative integer. The allowable values for the unit depend on the compiler and the environment being used. Usually a number between 10 and 100 is safe to use in pure FORTRAN programs. Allowable unit numbers in MARC/MENTAT are slightly different (see Table 2.1).
- For a temporary file which uses a SCRATCH specifier, no FILE specifier is required.
- The *access specifier* (ACCESS) is used for either a sequential or direct access. If a direct access method is chosen, using the RECL specifier is a must.
- The *record length specifier* (RECL) sets the record length in a direct access: for a formatted file in terms of characters and for an unformatted file in terms of bytes.
- If a *form specifier* (FORM) is absent, the default value for a direct and sequential access is unformatted and formatted, respectively.
- The *position specifier* (POSITION) can only be used in a sequential access.
- The following specifiers are only permitted in a formatted file: PAD, BLANK, DECIMAL, DELIM, ROUND and SIGN.
- If the *padding specifier* (PAD) is set to NO, the input must be exactly the same length as required. But if a YES value is assigned, the empty characters are filled with blanks.

After establishing the connection to an external file and finishing the data transfer, the connection should be terminated by means of a CLOSE statement. Nevertheless, FORTRAN manages the unclosed files after the termination of the program. But it is not recommended to leave an idle unit file connected. The syntax for a CLOSE statement is as the following:

CLOSE (close-spec-list)

**Table 1.23** Specifiers of an OPEN statement

| Specifier keyword          | Description  | Allowed values or return values                      |
|----------------------------|--|--|
| UNIT                       | Specifies the unit   | *: default unit                                      |
|                            |  | scalar-int-expr: for an external unit                |
|                            |  | char-var: for an internal unit                       |
| FILE                       | Specifies the name of the file   | scalar-char-expr: for file name                      |
| ACCESS                     | Specifies the access method of the connection  | 'DIRECT': for a direct access                        |
|                            |  | 'SEQUENTIAL (DEFAULT)': for a sequential access      |
|                            |  | 'STREAM': for a stream access                        |
| ACTION                     | Specifies the allowed actions  | 'READ': prohibits writing                            |
|                            |  | 'WRITE': prohibits reading                           |
|                            |  | 'READWRITE': permits read/write                      |
| STATUS                     | Specifies the file existence before establishing a connection                                    | 'OLD': file must be an existing one                  |
|                            |  | 'NEW': file must be non-existing to be created       |
|                            |  | 'UNKNOWN': the status is unknown (default)           |
|                            |  | 'REPLACE': a new file is created even if exists      |
|                            |  | 'SCRATCH': file is deleted after program termination |
| IOSTAT                     | Returns the status of the open statement execution by an integer                                 | >0: error condition<br>=0: successful execution      |
| RECL                       | Specifies the length of records in a direct access   | Positive integer value                               |
| FORM                       | Specifies the format of the I/O file   | 'FORMATTED': formatted file                          |
|                            |  | 'UNFORMATTED': unformatted file                      |
| POSITION                   | Specifies the position of a sequential file  | 'ASIS': position not altered (default)               |
|                            |  | 'REWIND': set to initial point                       |
|                            |  | 'APPEND': set to terminal point/before EOF record    |
| ERR                        | Error specifier  | label: a label to branch                             |
| Formatted files specifiers |  |  |
| PAD                        | Specifies using blanks in the place of missing characters of inputs shorter than required length | 'YES': blank padding (default)                       |
|                            |  | 'NO': no blanks padding                              |
| BLANK                      | Specifies the interpretation of blanks in numeric fields   | 'NULL': ignores all blanks                           |
|                            |  | 'ZERO': all trailing blanks are zeros                |

(continued)

**Table 1.23** (continued)

| Specifier keyword                     | Description                                     | Allowed values or return values           |
|---------------------------------------|---|---|
| DECIMAL                               | Sets the character indicating the decimal point | 'COMMA': ',' as decimal point             |
|                                       |   | 'POINT': '.' as decimal point             |
| DELIM                                 | Specifies the delimiting character              | 'APOSTROPHE': ''' used as delimiter       |
|                                       |   | 'QUOTE': '"' used as delimiter            |
|                                       |   | 'NONE': no delimiter characters (default) |
| ROUND                                 | Sets the rounding mode of formatted I/O         | 'UP': rounds up                           |
|                                       |   | 'DOWN': rounds down                       |
|                                       |   | 'ZERO': round to zero                     |
|                                       |   | 'NEAREST': round to nearest               |
|                                       |   | 'COMPATIBLE': compatible rounding         |
| 'PROCESSOR_DEFINED': default rounding |   |   |
| SIGN                                  | Controls the plus sign                          | 'PLUS': plus sign always appears          |
|                                       |   | 'SUPPRESS': plus sign will not appear     |
|                                       |   | 'PROCESSOR_DEFINED': default              |

**Table 1.24** Specifiers of a CLOSE statement

| Specifier keyword | Description   | Allowed values or return values                         |
|-------------------|---|---|
| UNIT              | Specifies the unit number   | scalar-int-expr: for an external unit                   |
| STATUS            | Specifies the file existence after terminating the connection     | 'KEEP': keeps the file (default)                        |
|                   |   | 'DELETE': deletes the file (default for a scratch file) |
| IOSTAT            | Returns the status of the close statement execution by an integer | >0: error condition                                     |
|                   |   | =0: successfull execution                               |
| ERR               | Error specifier   | label: a label to branch                                |

A UNIT specifier is required in the *close statement specifier list* (close-spec-list). In order to use the CLOSE statement specifiers, mind the list in Table 1.24. It is notable that the specifiers of a CLOSE statement are very much similar to those of an OPEN statement. Also remember that after a sequential write to a file, closing the file also writes an end-of-file record to the file.

### 1.9.3 Data Transfer Statements

Data transfer statements, i.e. Print or Write and Read, are used to handle all sorts of I/O actions ranging from printing messages on the screen and reading keyboard inputs to dealing with internal/external files. The syntax for these statements are as follows:

```
READ (io-control-spec-list) [ input-item-list ]  
READ format [, input-item-list ]
```

```
WRITE (io-control-spec-list) [ output-item-list ]  
WRITE format [, output-item-list ]
```

```
PRINT format [, output-item-list ]
```

The I/O control specifiers (io-control-spec-list) add a lot of flexibility to an I/O process; a brief list is prepared in Table 1.25. In addition, mind the following points while using I/O control specifiers:

- A *unit specifier* (UNIT) is the most common specifier and it is required when dealing with external files; other specifiers are optional. The default unit (\*) is used either for formatted sequential input or output. The associated number is usually processor-dependent (5 and 6). The default I/O units refer to the keyboard as input and the display as output in a pure FORTRAN program.
- The *format specifier* (format) is used to shape the data into a specific format and will be described more in detail in the next subsection.
- A *namelist specifier* (NML) lists data objects to be read or written. It cannot be used together with a format specifier.
- If a format or a namelist specifier is used in a data transfer statement, it is called a *formatted I/O statement* otherwise it is an *unformatted I/O statement*.
- The *advance specifier* (ADVANCE) is used just for a formatted sequential external file. It indicates an advancing or non-advancing positioning in the file. If a size control specifier (SIZE) or an end-of-record control specifier EOR is used, then a non-advancing approach must be set by using ADVANCE=NO specifier.
- An *end-of-file control* (END) is used to determine if during a reading process, the end of the file is reached without any errors. Then the IOSTAT variable is set to a negative integer (if present) and finally a jump is made to the indicated label.
- An *end-of-record control specifier* (EOR) is used to indicate the end of a record in a non-advancing reading approach. If the end of a record is reached, the IOSTAT variable is set to a negative integer (if present) and finally, a jump is made to the indicated label.

**Table 1.25** Specifiers of data transfer statements (I/O control specifiers)

| Specifier keyword | Description   | Allowed values or return values   |
|-------------------|---|---|
| UNIT              | Specifies the unit  | *: default unit<br>scalar-int-expr: for an external unit<br>char-var: for an internal unit          |
| FMT               | Specifies the format  | *: for a list-directed formatting<br>label: label of a format statement<br>char-expr: format string |
| NML               | Specifies a namelist group  | namelist-group-name: name of a namelist group   |
| ADVANCE           | Specifies the advancing or non-advancing formatted sequential data transfer | 'YES': advancing (default)<br>'NO': non-advancing   |
| SIZE              | Returns the number of read characters                                       | scalar-default-int-variable: size indicator   |
| EOR               | End-of-record specifier in a read statement                                 | label: a label to branch  |
| REC               | Specifies the record number to be read or written                           | scalar-int-expr: record number  |
| IOSTAT            | Returns the status of the I/O statement execution                           | scalar-default-int-variable: status indicator   |
| ERR               | Error specifier   | label: a label to branch  |
| END               | Specifies the end-of-file without any errors in a read statement            | label: a label to branch  |

- An *error control specifier* (ERR) indicates that an error has occurred. Then the position of the file becomes indeterminate and the IOSTAT variable is given a positive integer (if present). Finally, a jump is made to the designated label.
- The status of the I/O statement (IOSTAT) is returned by an integer: zero for a successful execution, a negative number for an end-of-file or end-of-record condition and a positive number for any other errors.
- A *record number specifier* (REC) indicates the record number to be read or written and it can only be used in a direct file access.
- A *size control specifier* (SIZE) returns the number of read characters in a non-advancing reading process. Note that if PAD specifier is set to YES, then the inserted blanks are not counted.

An *I/O item list* (io-item-list) is either an *input list* used for a read statement or an *output list* used for a write statement. An input list is a list of variables whereas an output list is a list of expressions. An implied-DO loop (p. 82) can be used in either cases.

### 1.9.4 File Positioning Statements

The connection and data transfer statements affect the position of the file. However, there are three specialized statements, i.e. *file positioning statements*, which enable the programmer to control the position of a file:

- the BACKSPACE statement is used for backspacing, i.e. returning the position to the previous record,
- the REWIND statement is used for rewinding, i.e. returning the position to the beginning of the file, and
- the ENDFILE statement is used for writing an end-of-file record and sets the position after this record.

All positioning statements have the following common syntaxes:

```
positioning-stmt external-file-unit
positioning-stmt (position-spec-list)
```

A positioning statement (positioning-stmt) operates on an external file unit (external-file-unit) by the help of a list of positioning specifiers (position-spec-list). The position specifiers listed in Table 1.26 can be used minding the following notes:

- Note that the position statements work on a file which is connected with a sequential access.
- If an error occurs during the execution of a position statement, the position of the file will be indeterminate.
- Backspacing in a non-advancing method returns the position of the current record to its beginning. In an advancing method in which there is no current record, the position is placed before the preceding record. For example, if the preceding record is an end-of-file record, the position will be placed before that.
- If the last I/O statement in a sequential file is an output data transfer statement, no end-of-file record will be present at the moment. In this case, backspacing the file will perform two tasks: first, it executes an ENDFILE statement to write the end-of-file record and second, it positions the file before the record preceding the end-of-file one. In a similar situation, a REWIND statement writes an end-of-file record and positions the file in its beginning.

**Table 1.26** Specifiers of position statements

| Specifier keyword | Description  | Allowed values or return values                  |
|-------------------|--|--|
| UNIT              | Specifies the unit number  | scalar-int-expr: for an external unit            |
| IOSTAT            | Returns the status of a position statement execution by an integer | >0: error condition<br>=0: successfull execution |
| ERR               | Error specifier  | label: a label to branch                         |



- After execution of an ENDFILE statement, no data transfer is permitted unless a rewind or backspacing is done.

### 1.9.5 *INQUIRY Statement*

In some cases it is required to obtain specific information regarding a file and then, deal with it correspondingly. The INQUIRY statement can extract such information based on either the file unit number, using the UNIT specifier, or the file name, using a FILE specifier. The syntaxes of an inquiry is as the following:

INQUIRE (inquire-spec-list)

INQUIRE (IOLENGTH=scalar-int-var) output-item-list

A list of inquiry specifiers (inquire-spec-list) is presented in Table 1.27. Since all of these specifiers return values, appropriate variables (of compatible types) should be used for each of the specifiers. The only exceptions are the UNIT and the FILE specifiers which can be specified only by values. The mentioned table can be used minding the following points:

- An inquiry can be made using either one of the UNIT or the FILE specifiers but not both.
- An inquiry can be made before or after connecting a file to a unit.
- All character values are returned in upper-case letters except for the NAME specifier.

The second syntax is used for inquiries by an output item list (output-item-list) which returns the I/O length (IOLENGTH) as an integer variable (scalar-int-var). It is used to calculate the required record size for the intended output list prior to establishing the connection with an unformatted file. In other words, the required size of the file to hold the unformatted output list is obtained. This value can be used later in a RECL specifier of the OPEN statement for a direct accessed unformatted file.

### 1.9.6 *Data Format*

Formatted I/O procedures are preferred for common practices because the results can be modified and checked using a text editor. As mentioned earlier, a list of entities to be read/written is called an I/O list. These items can be either expressions as outputs or variables as inputs. In addition, implied-DO lists can be used for both of them.

At the lowest level, data entities are saved in their binary representation in the memory. In contrast, the same data is saved as characters in a formatted file. Translating the items of an I/O list to their preferred character representation is done using a *format specification*.

**Table 1.27** Specifiers of an INQUIRY statement

| Specifier keyword | Description   | Allowed values or return values   |
|-------------------|---|---|
| UNIT              | Used for making inquiries by means of the unit number                                 | scalar-int-expr: for an external unit   |
| FILE              | Used for making inquiries by means of the file name                                   | scalar-char-expr: for file name   |
| ACCESS            | Returns a scalar-char-variable for the access method of the connection                | 'DIRECT': for a direct access<br>'SEQUENTIAL': for a sequential access<br>'UNDEFINED': for a disconnected file  |
| ACTION            | Returns a scalar-char-variable for the allowed actions                                | 'READ': prohibits writing<br>'WRITE': prohibits reading<br>'READWRITE': permits read/write<br>'UNDEFINED': for a disconnected file                        |
| BLANK             | Returns a scalar-char-variable for a blank specifier in numeric fields                | 'NULL': ignores all blanks<br>'ZERO': all trailing blanks are zeros<br>'UNDEFINED': unformatted or disconnected file                                      |
| DELIM             | Returns a scalar-char-variable indicating the delimiting character                    | 'APOSTROPHE': ''' used as delimiter<br>'QUOTE': '"' used as delimiter<br>'NONE': no delimiter characters<br>'UNDEFINED': unformatted or disconnected file |
| DIRECT            | Returns a scalar-char-variable indicating if a direct access is allowed               | 'YES': direct access is allowed<br>'NO': direct access is not allowed<br>'UNKNOWN': direct access status is unknown                                       |
| ERR               | Error specifier of the inquiry  | label: a label to branch  |
| EXIST             | Returns a scalar-logical-var if the file/unit exists                                  | .TRUE.: file or unit exists<br>.FALSE.: file or unit does not exist   |
| FORM              | Returns a scalar-char-variable to indicate if the file is connected for formatted I/O | 'FORMATTED': formatted file<br>'UNFORMATTED': unformatted file<br>'UNDEFINED': for a disconnected file  |
| FORMATTED         | Returns a scalar-char-variable indicating if a formatted data transfer is allowed     | 'YES': formatted transfer allowed<br>'NO': formatted transfer is not allowed<br>'UNKNOWN': unknown formatted transfer                                     |
| IOSTAT            | Returns a scalar-int-variable for execution status of the statement                   | >0: error condition<br>=0: successfull execution  |

(continued)

**Table 1.27** (continued)

| Specifier keyword | Description   | Allowed values or return values   |
|-------------------|---|---|
| NAME              | Returns a scalar-char-variable for the name of a connected file                           | Defined value: a file name  |
|                   |   | Undefined value: file without a name or disconnected                        |
| NAMED             | Returns a scalar-logical-variable indicating if the file has a name                       | .TRUE.: file has a name   |
|                   |   | .FALSE.: file does not have a name  |
| NEXTREC           | Returns a scalar-int-variable returns the number of next record in a direct access method | last record + 1: next record to be processed<br>1: no records are processed |
|                   |   | Undefined value: indeterminate position or not connected for direct access  |
| NUMBER            | Returns a scalar-int-variable for the unit number   | number: unit number   |
|                   |   | -1: no file connected to the unit   |
| OPENED            | Returns a scalar-logical-variable indicating if the unit is connected                     | .TRUE.: unit is connected   |
|                   |   | .FALSE.: unit is not connected  |
| PAD               | Returns a scalar-char-variable for the value of a PAD specifier                           | 'YES': padding is used for the file   |
|                   |   | 'NO': padding is not used for the file                                      |
| POSITION          | Returns a scalar-char-variable indicating the position of file when connected             | 'ASIS': position not altered  |
|                   |   | 'REWIND': positioned at initial point                                       |
|                   |   | 'APPEND': positioned at terminal point                                      |
|                   |   | Undefined value: indeterminate position or connected for direct access      |
| READ              | Returns a scalar-char-variable if a read is allowed for the file                          | 'YES': read is allowed  |
|                   |   | 'NO': read is not allowed   |
|                   |   | 'UNKNOWN': unknown read condition   |
| WRITE             | Returns a scalar-char-variable if a write is allowed for the file                         | 'YES': write is allowed   |
|                   |   | 'NO': write is not allowed  |
|                   |   | 'UNKNOWN': unknown write condition  |
| READWRITE         | Returns a scalar-char-variable if a read and write is allowed for the file                | 'YES': read-write is allowed  |
|                   |   | 'NO': read-write is not allowed   |
|                   |   | 'UNKNOWN': unknown read-write condition                                     |

(continued)

**Table 1.27** (continued)

| Specifier keyword | Description   | Allowed values or return values  |
|-------------------|---|--|
| RECL              | Returns a scalar-int-variable indicating the maximum record length (sequential access) or record length (direct access) | Defined value: character length (formatted) or one-byte size (unformatted)<br>Undefined value: file does not exist |
| SEQUENTIAL        | Returns a scalar-char-variable indicating if a sequential access is allowed   | 'YES': sequential access is allowed<br>'NO': sequential access is not allowed<br>'UNKNOWN': unknown access         |
| UNFORMATTED       | Returns a scalar-char-variable indicating if an unformatted access is allowed   | 'YES': unformatted access is allowed<br>'NO': unformatted access is not allowed<br>'UNKNOWN': unknown access       |

- a *data edit descriptor* which represents a data value,
- a *control edit descriptor* which controls the spacing, positioning, rounding, interpretation of blanks and dealing with the plus sign, and
- a *string edit descriptor* which is a literal constant of character type. It is usually used to add more information to the data item.

A format specification consists of several *edit descriptors*. There are three kinds of edit descriptors:

In simple words, the `FORMAT` specifier indicates the proper widths for data entities and represents them in a fashionable manner. If the `FORMAT` specifier is not used, the I/O statement determines a proper width based on the type of the data being used. This is called *implicit formatting* which can be used for some common practices. However, in more complicated cases such as formatted files an *explicit formatting*, i.e. using the `FORMAT` specifier, is the best approach.

Note that generally in a formatted file, it is not permitted to transfer data in an unformatted fashion. However, it is possible to do so with an implicit formatting. For instance, assume that the (`fileUnit = 20`) file unit is opened for a formatted writing process. Also consider the following output statement which is used to write a line of text in the file:

```
1 WRITE (fileUnit) 'an unformatted text...'
```

Because no explicit/implicit formatting is specified, the INTEL® FORTRAN compiler will raise the following error while compiling the line:

```
forrtl: severe (256): unformatted I/O to unit open for formatted transfers, unit 20, file
```

A quick way of resolving this issue is using implicit formatting. This can be done by replacing the previous line with the following one:

```
2 WRITE (fileUnit,*) 'an unformatted text...'
```

An explicit format specifier can be either a labeled `FORMAT` statement for complicated and frequently-referenced formats or a character expression for simpler and one-time cases. The syntax of a labeled `FORMAT` statement is as the following:

```
label FORMAT ([ format-item-list ])
```

A format item list (`format-item-list`), delimited in parentheses and separated by commas, makes up the format specification. This `FORMAT` statement can be referenced by means of the indicated label. Each data item on a format list must have a corresponding data edit descriptor plus optional control edit descriptors and/or string data descriptor. For instance, explicit formatting can be used to print three integer, real and character variables named `myInt`, `myReal` and `myChar` with the following code:

```
1 WRITE (*,100) myInt, myReal, myChar
2 100 FORMAT (I5, F8.4, A10)
```

The line up of the variables is in correspondence with the indicated format. The labeled `FORMAT` statement can be located anywhere in the scope of the code. For a one-time use, a character expression format can be used to obtain the same result:

```
1 WRITE (*,'(I5, F8.4, A10)') myInt, myReal, myChar
```

Any positive number before a data edit descriptor is considered a *repeat factor* with the default value of 1 which allows compact format specifications. For instance, (I4, I4, I4) can easily be expressed with a repeat factor of 3 like (3I4). It is also possible to nest formats with parentheses to define more complicated formats in a concise way. For instance, to print an array named `myArray` with 20 integer elements the following formats are possible:

```
1 WRITE (*,'(5(I2,I3,I4,I3))') myArray
2 WRITE (*,'(2(3I2,2I3),10I5)') myArray
```

Note that the array named `myArray` is broken down to its elements in order to create an *effective item list* which matches data entities and data edit descriptors. The same thing happens for a user defined data type or a complex number.

A character string edit descriptor is merely an informative character string which is printed exactly as it is. For example, to indicate the number of iteration in a loop by the 'Loop #' string and the result of the calculation by the 'Result =' string, the following code could be used:

```
1 DO i = 1, 10
2 ...
3 WRITE (*,'(Loop #',I2,' Result =',F8.2)') i, r
4 END DO
```

Note that the blanks within a character edit descriptor are significant. A list of standard data and control edit descriptors is summarized in Table 1.28. There are many subtle points using these descriptors. Nevertheless, the following points would be useful in a formatted data transfer:

- A plus sign is mandatory when the SP descriptor is used and optional if the SS or the S descriptors are used.
- The BN descriptor treats trailing blanks in a numeric input as blanks whereas the BZ descriptor treats them as zeros.

**Table 1.28** Data and control edit descriptors in FORTRAN

| Descriptor format        | Description   |
|--------------------------|---|
| Data edit descriptors    |   |
| [r]I w[.m]               | Integer data  |
| [r]B w[.m]               | Integer data (Binary representation)                      |
| [r]O w[.m]               | Integer data (Octal-base representation)                  |
| [r]Z w[.m]               | Integer data (Hexadecimal representation)                 |
| [r]F w.d                 | Real data type (without an exponent)                      |
| [r]D w.d                 | Real data type (with an exponent)                         |
| [r]E w.d[E e]            | Real data type (with an exponent)                         |
| [r]EN w.d[E e]           | Real data type (engineering notation)                     |
| [r]ES w.d[E e]           | Real data type (scientific notation)                      |
| [r]G w.d[E e]            | General intrinsic data type                               |
| [r]A [w]                 | Character type data                                       |
| [r]L w                   | Logical data type   |
| Control edit descriptors |   |
| BN                       | Ignores non-leading blanks in numeric items               |
| BZ                       | Non-leading blanks considered as zeros in numeric items   |
| RC                       | Compatible rounding                                       |
| RD                       | Rounding down   |
| RN                       | Rounding to the nearest                                   |
| RP                       | Processor-defined rounding                                |
| RU                       | Rounding up   |
| RZ                       | Rounding to zero  |
| T n                      | Moves to position n                                       |
| TL n                     | Moves left n position(s)                                  |
| TR n                     | Moves right n position(s)                                 |
| n X                      | Moves right n position(s)                                 |
| S                        | Optional plus sign based on the processor (default)       |
| SP                       | Mandatory plus sign                                       |
| SS                       | Suppress plus sign  |
| /                        | End current record  |
| :                        | Stop format processing if no further I/O list items exist |
| kP                       | Scale factor is applied to certain real numbers           |

Notes: w total width of the field including the decimal point and sign in numerals

m minimum number of digits

d number of digits after decimal point

e number of digits of the exponent

n indicates the number of positions to move

k scale factor

r repeat factor (If not indicated, it is considered to be equal to 1.)

- No scaling is present in a format specifier unless the  $kP$  descriptor is used. It indicates a scale factor of  $k$  for all the real numbers. If an input is entered without an exponent, the value is multiplied by  $10^k$ . For an output  $F$  descriptor, the value is multiplied by  $10^k$ . However, for  $E$  or  $G$  descriptors, the significant is multiplied by  $10^k$  and the exponent is reduced by  $k$ .
- A colon is used to stop formatting if the required items are not present. This is useful for the cases when the number of items is not known. For example, consider a list of maximum ten integers which is to be printed but the exact number of items is not known. To print exactly ten integers the  $10(I5,X)$  format is used. However, the  $10(I5,;X)$  format must be used for any number of integers less than or equal to ten. Note that even the edit descriptor for the space ( $X$ ) will not be used for the last item on the list.
- A slash ends the current record. Therefore, for an input file, the remaining part of the file will be discarded and the position will be moved to the beginning of the next record. In an output file, the current record will be written and then, the position will be moved. For a simple output, a slash is the same as a carriage-return and moves the cursor to the beginning of a new line.
- Obviously there is a limit for left tabbing using the  $(TL)$  descriptor. In a formatted record this limit is the beginning of the record for an advancing method and the current position for a non-advancing method.
- The rounding is set firstly using the  $OPEN$  statement. If no rounding descriptor is used in an explicit formatting, the default value will be used. The rounding occurs because of the difference between the number and its representation (see Sect. 1.7.2). However, rules are the same as for an  $OPEN$  statement and they can be summarized as the following points:
  - *Round up* results in the smallest value greater than or equal to the original value.
  - *Round down* results in the largest value less than or equal to the original value.
  - *Round to zero*, results to the closest representable value which is not greater than the original value.
  - *Round to the nearest*, results in the closest of the two representable values if there is not such a value the decision is processor-dependent.
  - *Compatible rounding*, results in the same as the nearest rounding unless the original value is halfway between two representable values. In the latter case, the value away from zero is selected.
  - *Processor-defined rounding* is a reserved type of rounding which is different from all other types.

## 1.10 Summary of Accessing Files

As mentioned earlier, two main categories of accessing external files are the sequential and the direct access method. The sequential access can be done in either a formatted manner (advancing or non-advancing) or an unformatted manner. For direct

access, only formatted and unformatted access is supported in the non-advancing mode.

It is a useful practice to check if a unit is occupied and if the file exists prior to applying any manipulations. For instance, the following code first checks the unit number. If it is disconnected, it then checks for the existence of the filename.txt file. If yes, it deletes the file and then connects the file, as a new one, to the same unit:

```

1      INTEGER :: fileUnit = 10
2      CHARACTER(LEN=250) :: fileName = 'filename.txt'
3      LOGICAL :: fileExist, fileEnded, unitConnected
4
5      INQUIRE (UNIT = fileUnit, OPENED = unitConnected)
6      IF (unitConnected .EQV. .FALSE.) THEN
7          INQUIRE (FILE = fileName, EXIST=fileExist)
8          IF (fileExist .EQV. .TRUE.) THEN
9              Print *, 'file exists.'
10             OPEN (UNIT = fileUnit, File = fileName, STATUS = 'OLD')
11             CLOSE (UNIT = fileUnit, STATUS = 'DELETE')
12         END IF
13         OPEN (UNIT = fileUnit, File = fileName, ACCESS = 'SEQUENTIAL',
14             & STATUS = 'NEW', ACTION = 'READWRITE')
15     ! I/O statements go here ...
16     CLOSE (UNIT = fileUnit)
17     ELSE
18         WRITE (*,*) 'Unit is already connected.'
19     END IF

```

Another case which is frequently encountered is determining whether an end-of-file (EOF) case or an end-of-record case (EOR) has occurred. In a READ statement, the IOSTAT specifier returns different negative values for each case which can be distinguished using the following intrinsic functions:

- `is_iostat_end (i)` returns `.TRUE.` for the I/O status variable `i` if it is the end of the file, otherwise a `.FALSE.` value is returned.
- `is_iostat_eor (i)` returns `.TRUE.` for the I/O status variable `i` if it is the end of a record, otherwise a `.FALSE.` value is returned.

It is recommended to use these functions instead of assigning labels to the EOF and EOR specifiers. This will result in a more structured program.

The most frequent use of external files is using them to generate formatted reports. Therefore, in the simplest form, a sequential formatted access will satisfy the needs of a program by producing portable text files. However, various types of access are summarized in the following subsections.

### ***1.10.1 Sequential Formatted Access - Advancing Versus Non-advancing***

Advancing and non-advancing data transfer is applicable to a file which is connected via sequential formatted access. Because the access is formatted, the file is considered as a sequence of records. In addition, an unformatted data transfer is not allowed for a formatted access. However, it is possible to switch between the advancing and



non-advancing approaches in a session. Nevertheless, the file positioning behaves differently in each case.

In the advancing data transfer, the execution of an I/O statement positions the file at the end of the record. In contrast, a non-advancing data transfer can be stopped in the middle of a record.

Consider the case that a non-advancing method is used to read a part of record and now the file position is in the middle of a record. If the method is changed to the advancing method, reading will be continued from the current file position and not the beginning of a record.

It is also possible for an advancing sequential access to be interrupted in the following cases:

- an error condition,
- an EOF condition while reading, or
- while processing the format a colon descriptor comes up in a short list of input/output data list. This will result in jumping to the beginning of the next record (if any available).

From another perspective, a simple WRITE statement can be used in both advancing and non-advancing methods to print some formatted output. An asterisk is used for the default output, i.e. the display in normal cases. The difference will be only that at the end of the advancing method a carriage return will return the cursor to the beginning of the next line. This is similar to finishing a record in a file and moving to the beginning of the next record. In contrast, in a non-advancing method, the cursor is moved to the beginning of the next character at the same line.

The following syntax is used for I/O statements of an advancing sequential access of a formatted file:

```
READ ([UNIT =] io-unit [, FMT =] format [, IOSTAT = scalar-int-var] [,ERR = label]
[, END = label] [, ADVANCE = 'YES']) [input-item-list]
```

```
WRITE ([UNIT =] io-unit [, FMT =] format [, IOSTAT = scalar-int-var] [,ERR = label]
[, ADVANCE = 'YES']) [output-item-list]
```

In this syntax, the I/O unit (io-unit) can be either an asterisk or a non-negative number. Using this syntax, in the following listing, a file is opened in a sequential formatted access. Then, a header plus a line of data is written to the file. Finally, the file is moved to its beginning to read the written data and print them on the screen:

```

1      INTEGER :: i, j, recCounter, ioResult, fileUnit = 10
2      CHARACTER(LEN=250) :: fileName = 'temp.txt'
3      LOGICAL :: fileExist, fileEnded, unitConnected
4      CHARACTER(LEN=22) :: c
5      CHARACTER(LEN=35) :: header
6      REAL :: r
7
8  99   FORMAT (A)
9  100  FORMAT (i3, 2x, F6.2, '%', 1x, A)
10
11     INQUIRE (UNIT = fileUnit, OPENED = unitConnected)
```

```

12      IF (unitConnected .EQV. .FALSE.) THEN
13          INQUIRE (FILE = fileName, EXIST=fileExist)
14
15
16          IF (fileExist .EQV. .TRUE.) THEN
17              Print *, 'file exists.'
18              OPEN (UNIT = fileUnit, File = fileName, STATUS = 'OLD')
19              CLOSE (UNIT = fileUnit, STATUS = 'DELETE')
20          END IF
21
22          OPEN (UNIT = fileUnit, File = fileName, ACCESS = 'SEQUENTIAL',
23              & STATUS = 'NEW', ACTION = 'READWRITE')
24          WRITE (fileUnit, 99, ADVANCE='YES') 'Inc# Percent Comments'
25          WRITE (fileUnit, 100, ADVANCE='YES') 345,-12.3456,'A comment'
26          ENDFILE (fileUnit)
27          REWIND (fileUnit)
28
29          recCounter = 0
30          ioResult = 0
31
32          fileEnded = .FALSE.
33          READ (UNIT = fileUnit ,FMT = 99, ADVANCE = 'YES') header
34          Write (*,99) header
35
36          DO WHILE (fileEnded .EQV. .FALSE.)
37              READ (UNIT = fileUnit ,FMT = 100, ADVANCE = 'YES',
38              & IOSTAT = ioResult) i, r, c
39              WRITE (*,100) i,r,c
40              WRITE (*,*) ioResult
41              WRITE (*,*) is_iostat_end (ioResult)
42              IF (is_iostat_end (ioResult) .EQV. .TRUE.) THEN
43                  fileEnded = .TRUE.
44              ELSE
45                  recCounter = recCounter + 1
46              END IF
47          END DO
48
49          WRITE (*,*) 'Number of read record(s) ', recCounter
50          CLOSE (UNIT = fileUnit)
51      ELSE
52          WRITE (*,*) 'Unit is already connected.'
53      END IF

```

It was not really necessary to use a WHILE loop in this specific example, because it is already known that there is only one record in the file. Also note that two different formats are used for the header and the data part. The ADVANCE specifier is not used in the WRITE statement to show that the default value for advancing is YES.

In contrast to the advancing method, the non-advancing formatted method may be more suitable for the following cases:

- a case in which only a part of data is required to be read or written. It is called reading *partial records*, e.g. only the first two columns of a table,
- records with varying lengths, or
- irregularly formatted records. Consider the case that based on the value of a field, the format and/or the length of the rest of the record may be different. Therefore, it is not possible to read/write the whole record at once.

The syntax of the non-advancing method is as the following:

```

READ ([UNIT =] io-unit [, FMT =] format , ADVANCE = 'NO' [, SIZE = scalar-int-var]
[, EOR = label] [, IOSTAT = scalar-int-var] [,ERR = label] [, END = label] [input-item-list])

```

```
WRITE ([UNIT =] io-unit [, FMT =] format , ADVANCE = 'NO' [, IOSTAT = scalar-int-var]
[,ERR = label]) [output-item-list]
```

Note that in this syntax, the ADVANCE = 'NO' specifier is required to override the default 'YES' value. In non-advancing reading, EOR and SIZE specifiers are made available to detect an end-of-record condition and read records with varied sizes, respectively. Note that the EOR descriptor cannot be used in the advancing method. A non-advancing access method can be interrupted in the following conditions:

- an error condition,
- an EOF or EOR condition while reading, or
- while processing the format a colon descriptor comes up in a short list of input/output data list. This will result in jumping to the beginning of the next record (if any available).

As a demonstration, assume that there is a file which consists of several records with varied sizes and a maximum record size of 200 characters. It is necessary to read the first field of every record which is an integer number. The rest of the data is not required and thus, just stored in a temporary variable. A sequential formatted read with non-advancing features is selected to carry out the job. The first field will be read into the variable anInt and the rest to a temporary character variable named temp in the following code:

```
1 CHARACTER (LEN = 200) :: temp
2
3 OPEN (UNIT = fileUnit , File = fileName , ACCESS = 'SEQUENTIAL' ,
4 & STATUS = 'UNKNOWN' , ACTION = 'READ' , RECL = 200)
5 REWIND (fileUnit)
6
7 totalReadChars = 0
8 recCounter = 0
9
10 DO WHILE (fileEnded .EQV. .FALSE.)
11 READ (UNIT = fileUnit ,FMT='(I3)', ADVANCE = 'NO' ,
12 & IOSTAT = ioResult , SIZE = readChars) anInt
13 totalReadChars = totalReadChars + readChars
14
15 READ (UNIT = fileUnit ,FMT='(A)', ADVANCE = 'NO' ,
16 & IOSTAT = ioResult , SIZE = readChars) temp
17 totalReadChars = totalReadChars + readChars
18
19 IF (Is_iostat_end (ioResult) .EQV. .TRUE.) THEN
20 fileEnded = .TRUE.
21 ELSE
22 recCounter = recCounter + 1
23 END IF
24 END DO
```

In this example listing, the reading will continue until EOF is reached and the total number of characters (totalReadChars) and the total number of records (recCounter) is calculated.

### 1.10.2 Sequential Access - Unformatted

In an unformatted sequential data transfer, the data is stored in the external file in the binary representation. Therefore, it is the fastest of the sequential transfer methods. The term ‘record’ is not appropriate to be used for an unformatted file. However, it can be interpreted that an unformatted file consists of one unformatted record plus an EOF record.

The syntax of I/O statements of this method is as the following:

```
READ ([UNIT =] io-unit [, IOSTAT = scalar-int-var] [, ERR = label]
[END = label]) [input-item-list]
```

```
WRITE ([UNIT =] io-unit [, IOSTAT = scalar-int-var][, ERR = label]) [output-item-list]
```

Note that when a file is connected for an unformatted data transfer, formatted access is prohibited.

In the unformatted file no records are visible. Therefore, without knowing the structure of the written data, the data will not be accessible. For the same reason, an unformatted file is not a good choice in terms of portability, because even if the structure of the data is known, the size of the data types is processor-dependent and they may differ from the original resource. In addition, the file cannot be viewed or edited using a text editor.

Considering all the mentioned properties, an unformatted file is not a suitable choice to produce a portable output which can be edited manually and distributed by the user. However, it is a very good choice to create dump files of several parts of the memory due to its quick nature. In addition, the dumped values are not rounded because they are saved in the exact binary values.

For instance, the following code writes 2000 integers stored in an array named storage into an unformatted file:

```

1  INTEGER :: ioResult , fileUnit = 10
2  CHARACTER(LEN=250) :: fileName = 'dump.dat'
3  LOGICAL :: fileExist , fileEnded , unitConnected = .FALSE.
4
5  INTEGER, PARAMETER :: MAXITEMS = 2000
6  INTEGER :: storage(MAXITEMS) = [(i, i = 1, MAXITEMS)]
7  INTEGER, PARAMETER :: REQUIREDBYTES = Sizeof (storage)
8
9  Write (*,*) 'free = ', fileUnit
10
11  OPEN (UNIT = fileUnit , File = fileName , FORM = 'UNFORMATTED' ,
12  &     ACCESS = 'SEQUENTIAL' , STATUS = 'UNKNOWN' ,
13  &     ACTION = 'WRITE' , RECL = REQUIREDBYTES)
14
15  WRITE (fileUnit , IOSTAT = ioResult)
16  & [(storage(i) , i = 1, MAXITEMS)]
17
18  ENDFILE (fileUnit)
19  CLOSE (fileUnit)
```

Note that, when dealing with unformatted files, the sizes are in bytes not characters. Therefore, the intrinsic function `Sizeof()` is used which returns the size of any variable

in the memory. The variable `REQUIREDBYTES` is used to set the `RECL` specifier to the required size.

### 1.10.3 Direct Access - Formatted Versus Unformatted

The advantage of direct access is performing I/O actions on selected records of an external file. Contrary to the sequential method, in order to access a specific record number, there is no need to read all the previous records, i.e. random access is allowed. But unlike a sequential access, deleting a record is not possible. Data can just be overwritten therefore deleting can be carried out by filling the record with blanks.

Both of the formatted and unformatted methods can be used in a direct access. However, if the file is connected for a formatted access, unformatted access is prohibited and vice versa. For both formatted and unformatted access, all the records must have the same length as specified using the `RECL` specifier in the corresponding `OPEN` statement.

The following points characterize a formatted direct access:

- Although non-advancing or advancing is not applicable to direct access, it can be stated that the access method is always advancing. The `ADVANCE` specifier is not allowed.
- The `END` specifier is not allowed.
- An attempt to read the unwritten data causes the input values to be undefined.
- The size of the data to be written must not violate the record size. However, if the number of items on an output list is less than the record size, then the rest of the record will be filled with blanks.

The syntax for a formatted direct access I/O transfer is as the following:

```
READ ([UNIT =] io-unit [, FMT =] format , REC = scalar-int-expr
[, IOSTAT = scalar-int-var] [, ERR = label]) [input-item-list]
```

```
WRITE ([UNIT =] io-unit [, FMT =] format , REC = scalar-int-expr
[, IOSTAT = scalar-int-var][, ERR = label]) [output-item-list]
```

A formatted direct access is interrupted in the case of either one of the followings:

- an error condition, and
- a colon descriptor comes up while processing the format in a short list of input/output data list. This will result in jumping to the beginning of the next record (if any available).

In the following listing, 10 records will be created using the direct access method. Then, in order to delete the fifth record, it is overwritten with blanks:

1  
2

```
INTEGER :: ioResult , fileUnit , recordLength , intData , i
CHARACTER(LEN = 250) :: fileName = 'test.txt'
```

```

3      CHARACTER(LEN = 10) :: charData
4      REAL :: realData
5
6      101  FORMAT (20X)
7      100  FORMAT (I3,1X,F5.1,1X,A10)
8
9      recordLength = 20
10     fileUnit = 10
11
12     OPEN (UNIT = fileUnit, File = fileName, FORM = 'FORMATTED',
13           & ACCESS = 'DIRECT', STATUS = 'UNKNOWN',
14           & ACTION = 'READWRITE', RECL=recordLength)
15
16     intData = 1
17     realData = 12.3456
18     charData = 'Pressure'
19
20     DO i = 1, 10
21         WRITE (fileUnit, FMT = 100, IOSTAT = ioResult, REC = i)
22     & intData, realData, charData
23     END DO
24
25     WRITE (fileUnit, FMT = 101, IOSTAT = ioResult, REC = 5)
26
27     intData = 0
28     realData = 0.0
29     charData = ''
30
31     DO i = 1, 10
32         READ (fileUnit, FMT = 100, IOSTAT = ioResult, REC = i)
33     & intData, realData, charData
34         WRITE (*, "("RECORD #",I2,":")', ADVANCE = 'NO') i
35         WRITE (*, 100) intData, realData, charData
36     END DO
37
38     CLOSE (fileUnit)

```

This code generates a formatted file, i.e. a text file which can be edited by the user.

An unformatted direct access writes the binary data of the memory to the file. The following points characterize this type of access:

- The same type of written values must be repeated while reading to assure a correct data transfer. However, it is possible to specify fewer items to be read than the number of written items.
- Exactly one record is transferred in an I/O action.
- END, FMT and ADVANCE specifiers are not allowed.
- The size of the data to be written must not violate the record size.
- An attempt to read the unwritten data causes the input values to be undefined.
- If the number of items on an output list is less than the record size, then the rest of the record will be undefined.

The syntax for an unformatted direct access I/O transfer is as follows:

```

READ ([UNIT =] io-unit [, FMT =] format , REC = scalar-int-expr
[, IOSTAT = scalar-int-var] [, ERR = label]) [input-item-list]

```

```

WRITE ([UNIT =] io-unit [, FMT =] format , REC = scalar-int-expr
[, IOSTAT = scalar-int-var][, ERR = label]) [output-item-list]

```

An unformatted direct access is terminated in one of the following cases:

- an error condition, and
- a short number of items on either an input item list or output item list.

As mentioned previously, the unformatted I/O is a fast method to dump large amounts of data. One may prefer an unformatted direct access. As an example, the formatted direct-access example is re-written using an unformatted access:

```

1  INTEGER :: ioResult, fileUnit, recordLength, intData, i
2  CHARACTER(LEN = 250) :: fileName = 'test.txt'
3  CHARACTER(LEN = 10) :: charData
4  REAL :: realData
5
6      recordLength = sizeof (realData) + sizeof (intData) +
7  &                  sizeof (charData)
8
9      fileUnit = 10
10
11     OPEN (UNIT = fileUnit, File = fileName, FORM = 'UNFORMATTED',
12 &        ACCESS = 'DIRECT', STATUS = 'UNKNOWN',
13 &        ACTION = 'READWRITE', RECL=recordLength)
14
15     intData = 1
16     realData = 12.3456
17     charData = 'Pressure'
18
19     DO i = 1, 10
20         WRITE (fileUnit, IOSTAT = ioResult, REC = i)
21 &     intData, realData, charData
22     END DO
23
24     intData = 0
25     realData = 0.0
26     charData = ''
27     WRITE (fileUnit, IOSTAT = ioResult, REC = 5)
28 &     intData, realData, charData
29
30     DO i = 1, 10
31         READ (fileUnit, IOSTAT = ioResult, REC = i)
32 &     intData, realData, charData
33         WRITE (*, '("RECORD #",I2,":")', ADVANCE = 'NO') i
34         WRITE (*, *) intData, realData, charData
35     END DO
36
37     CLOSE (fileUnit)

```

In conclusion, there are various methods to tackle an I/O transfer job among which the best one could be easily recognized by user experience. However, I/O transfer is an extensive topic and this section is merely a brief introduction. One may follow more advanced techniques through reading the references.

# Chapter 2

## Introduction to Marc/Mentat

**Abstract** In the current chapter, the internal structure of MARC/MENTAT is explained. In this chapter, various aspects are considered such as interaction between MARC and MENTAT, program files, the input file structure, table-driven input, sets, user-defined subroutines, predefined-common blocks, debugging tips etc. In addition, a brief introduction to procedure files, using C programming codes in FORTRAN, and PYTHON scripts (PYMENTAT and PYPOST) are provided.

### 2.1 MARC/MENTAT Interactions

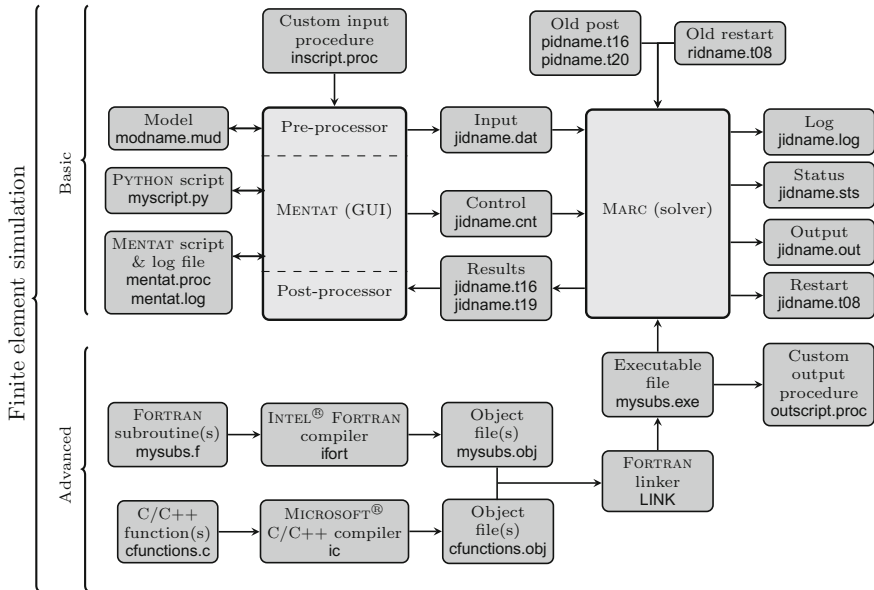
The commercial finite element package MARC/MENTAT provides a wide variety of advanced capabilities to facilitate the finite element modeling of physical phenomena. It consists of two major parts: MARC is the core solver and MENTAT is the GUI<sup>1</sup> responsible of both pre- and post-processing of the finite element model. For an unexperienced user, MENTAT is a good starting point. However, in order to take advantage of the advanced possibilities of this package, it is imperative for an experienced user to have a deeper understanding of how these parts communicate with each other in the background. As several interactions take place between these components, many files are created, opened, read and written. In a nutshell, MENTAT translates the modeling process with the corresponding details into the input file and then passes it to the solver to be read. The solver, MARC, reads this file, interprets it and carries out the solution of the system of equations and the results are passed back to MENTAT for post-processing. A part of these interactions is concisely illustrated in Fig. 2.1.

In this package, MENTAT can handle almost all the steps of generating and manipulating a finite element model before analysis (pre-processing) as well as working on the primary and secondary solutions after obtaining the output results (post-processing). The created finite element model is saved in the *model file* (.mud or .mfd) which is a binary file without any analysis results. This file is used to generate a formatted/unformatted text file, the *input file* (.dat). The input file will be sent to MARC for analysis. Note that MENTAT always starts in a directory, called the *working directory* in which the files mentat.proc and mentat.log will be saved. These two files

---

<sup>1</sup>Graphical User Interface.





**Fig. 2.1** MARC/MENTAT interaction

contain all of the user activities during a working session. Therefore, in the case of a sudden crash of the software the unsaved part of the model can be retrieved. These files are overwritten when a new session starts and thus, it is a good approach to set a separate working directories at the beginning of each session. Such capabilities make MENTAT a great starting point for finite element modeling. The advantages of using MENTAT can be summarized, but are not limited to, as follows:

- the input file can be generated in an interactive way which speeds-up the model generating process,
- extra lines can be added to the input file,
- pre-processing data can be viewed in a graphical environment, e.g. mesh, boundary conditions, loads, element types etc. which makes the debugging of the model easier,
- post-processing can be done to generate graphs, contours, iso-surfaces, cutting-planes, deformed/original shapes, and others, which allows for a quick view of the results,
- error-checking of the input file can be done prior to submitting a job and proper warnings/errors can be displayed,
- input-verification is carried-out to some extent while entering data via keyboard which prevents common mistakes, and
- automatization of the routine jobs is possible by means of the *procedure files* or PYTHON scripts.

Although generally using MENTAT can save a considerable amount of time in the model generation procedure, this is not enough when it comes to dealing with complicated models and advanced questions. In other words, it is a good idea to generate a basic model using MENTAT and apply further manipulations, for example, by directly editing the input file.

As mentioned earlier, after a model is created using MENTAT, upon running the corresponding job, an input file is created and passed to MARC for a *run*, i.e. an analysis. The result of a successful analysis is passed back to MENTAT for post-processing. Along with these, there are several other auxiliary files which are involved in the wider scope of the analysis; for instance:

- the *control file* (.cnt) by which the user can control the analysis process,
- the *status file* (.sts) containing a short summary of the analysis,
- the *log file* (.log) with the progress of the solution of the problem,
- the *output file* (.out) which is a descriptive file containing error and warning messages as well as additionally requested information,
- the *post files* (.t16 and .t19) in binary/text format which are the results of the analysis and can be used for subsequent analyses, and
- the *restart file(s)* (.t08) is created in specific points of an analysis which enables the user to continue the analysis from these points onward.

The most powerful aspect of MARC is providing the *user subroutines* capability which adds great flexibility in solving non-standard problems. A user subroutine can be written in FORTRAN, the C/C++ programming language or a mixture of both. The source files are compiled using the appropriate compilers, linked and run instead of the standard routines of MARC to carry out a user-desired procedure.

Programming features are also made available in MENTAT via PYTHON scripts and *procedure files*. PYTHON scripts can be used either to run commands in the MENTAT environment, called PYMENTAT, or to process the results of MARC, called PYPOST. In contrast, a procedure file is a text file which contains the commands and input values exactly as if the user is running and entering them.<sup>2</sup>

As mentioned in Sect. 1.9, handling a file in FORTRAN requires a unique file unit to handle the data transmission process. Considering the fact that the MARC/MENTAT package is also written in FORTRAN, not only the same concept is still true but also there are additional reserved unit numbers by which the read/write process of the auxiliary files is done.

In other words, when MARC deals with the supplementary files they remain connected until the end of the analysis. Therefore, these specific unit numbers are considered as pre-connected units. The user must avoid connecting such units to other files to prevent any conflicts. Although a unit number between 110 to 119 can be used for a personal file, using a number larger than 200 is recommended. In Table 2.1, a short list of these unit numbers is shown. For a more descriptive table, one may refer to [25]. The aforementioned capabilities will be discussed more descriptively in the following sections of the current chapter.

---

<sup>2</sup>Similar to the concept of a *macro* in other programs.

**Table 2.1** Selected FORTRAN file units used by MARC. Adapted from [25]

| File name      | Unit    | Description                                    | File type |
|----------------|---------|--|-----------|
| jidname.log    | 0       | Analysis sequence log file                     | SF        |
| jidname.t02    | 2       | OOO solver scratch file                        | DF        |
| jidname.t03    | 3       | ELSTO parameter storage for element data       | DU        |
| jidname.dat    | 5       | Data input file                                | SF        |
| jidname.out    | 6       | Output file                                    | SF        |
| jidname.t08    | 8       | Restart file, written out                      | SU        |
| ridname.t08    | 9       | Restart file to be read in from a previous job | SU        |
| jidname.t11    | 11      | OOO solver scratch file                        | SU        |
| jidname.t12    | 12      | OOO solver scratch file                        | SU        |
| jidname.t13    | 13      | OOO solver scratch file                        | SU        |
| jidname.t14    | 14      | OOO solver scratch file                        | DU        |
| jidname.t15    | 15      | OOO solver scratch file                        | SU        |
| jidname.t16    | 16      | Post file, written out                         | SU        |
| ridname.t16    | 17      | Post file to be read in from a previous job    | SU        |
| jidname.t19    | 19      | Post file, written out                         | SF        |
| ridname.t19    | 20      | Post file to be read in from a previous job    | SF        |
| jidname.sts    | 67      | Analysis progress reporting file               | SF        |
| user-specified | 110–119 | Custom files to be included in the input file  | SF        |
| jidname.dump   | N/A     | Scratch file for a failed in-core reallocation | SU        |

*OOO* Out-of-core

*SF* Sequential-access formatted

*SU* Sequential-access unformatted

*DD* Direct-access formatted

*DU* Direct-access unformatted

### 2.1.1 Mentat Commands

To provide a consistent system for specifying commands in MENTAT, the symbols in Table 2.2 are used. For example, the command for importing the test.dat input file to MENTAT is as follows:

③ File> Import> Marc Input> File name:test.dat

The interpretation of this command is to select File from the Main Menu Tabs. Then Import should be selected and after that, the Marc Input menu item must be selected. Finally, the value of the File name field must be set to test.dat.

This notation for specifying a MENTAT command is for the experienced user. A more descriptive notation can be found in [31] which is helpful for a beginner.

**Table 2.2** Command symbols for MENTAT

| Symbol | Description                              |
|--------|--|
| ①      | Dropdown menu                            |
| ②      | Function buttons                         |
| ③      | Main menu tabs                           |
| ④      | Tab sections                             |
| ⑤      | Model navigator                          |
| ⑥      | Graphic interface                        |
| ⑦      | Graphic interface navigation menu        |
| ⑧      | Command line dialog                      |
| ▷      | Command separator                        |
| a:b    | The value of <b>a</b> is set to <b>b</b> |

### 2.1.2 MARC Solver Types

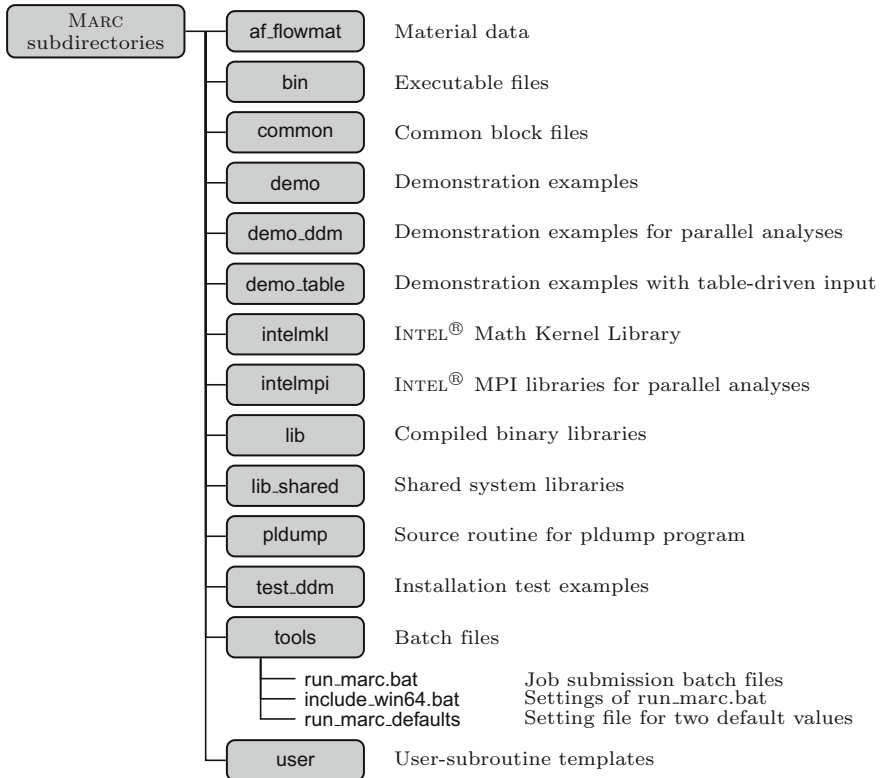
Using the finite element method leads to the solution of a system of simultaneous equations instead of the original differential equation governing the system. The most straightforward approach is using the inverse of the matrix of coefficients, which is not the best approach in every case. In MARC, the two main categories of solvers are *direct* and *iterative* ones. Various solver types are made available through the command-line and the SOLVER option; they are listed in Table 2.3.

### 2.1.3 Structure of the Installation Folder

After completing the installation of the MARC/MENTAT package, separate folders are created for MARC and MENTAT. If the default path is selected for the installation, then a path similar to C:\MSC.Software\Marc\2014.2.0 will be the parent folder of the package. Each part of the package, i.e. MARC and MENTAT, will have separate

**Table 2.3** Solver types for MARC

| Solver code | Solver type                        |
|-------------|------------------------------------|
| 0           | Profile direct solver              |
| 2           | Sparse iterative                   |
| 4           | Sparse direct solver               |
| 8           | Multi-frontal direct sparse solver |
| 9           | CASI iterative solver              |
| 10          | Mixed direct/iterative solver      |
| 11          | Pardiso parallel direct solver     |
| 12          | MUMPS parallel direct solver       |



**Fig. 2.2** Subdirectories of MARC

folders, e.g. `marc2014.2` and `mentat2014.2`, respectively. The subdirectories of each one of these folders contain all the executables and the corresponding setting files. In the advanced level, it is sometimes required to modify some of the installed files. Note that before applying any modifications, it is advised to make a backup copy of the files to be modified as a precaution to any unwanted or incorrect modifications. In Figs. 2.2 and 2.3, the typical subdirectories are shown for MARC and MENTAT, respectively. However, based on the chosen type and/or the version of installation package, the subdirectories might be slightly different than what is shown in the figures.

In Fig. 2.2, the subdirectories for the main folder of MARC are listed and are accompanied with a brief description for each one of them. Among these folders, the following ones are worth mentioning:

- The `common` subdirectory is dedicated to the common blocks of MARC. Each common block is included in a separate file with a `.cmn` file extension with a few lines of comments describing some of the variables.
- The `tools` folder contains the following important files:
  - The `run_marc.bat` is used to run a job from the command prompt.

- The `include_win64.bat` batch file runs prior to the `run_marc.bat` batch file in order to specify the settings for the latter.
- The `run_marc_defaults` file indicates the default solver mode (`MARC_MODE i4/i8`) and the default MPI<sup>3</sup> version. There are two solver modes: the LP64 or i4 mode which uses 4-byte integers and the ILP64 or i8 mode which uses 8-byte integers. The latter has almost no limitations regarding model size but it uses more memory. This setting is stored in the `run_marc_defaults` file in a line such as the following one:

```
MARC_MODE i8
```

In order to change this setting, simply edit the mentioned file and change `i4` to `i8` or the other way around, as preferred. Within the same file, the default MPI can be specified to either one of INTEL® MPI or HP® MPI via the following line:

```
MARC_MPI intelmpi
```

This line can be changed to `MARC_MPI hpmi` to apply the setting for HP® MPI.

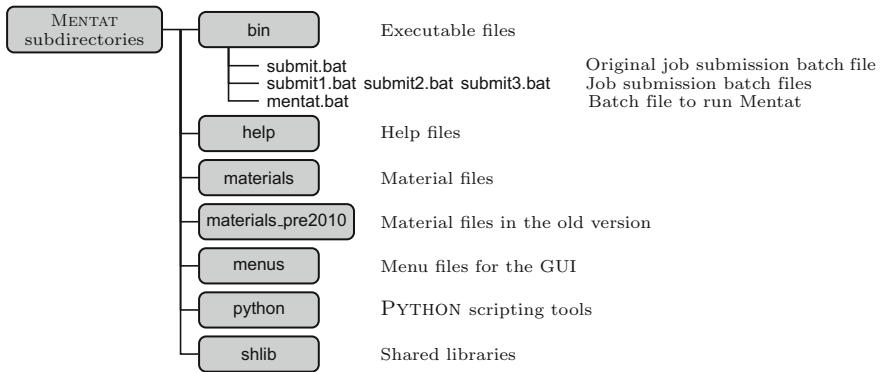
- The user folder contains the templates for the subroutines of MARC. The syntax of these templates is slightly different from that of the listed subroutines in the documentation of MARC. In the templates of the documentations, an implicit type declaration is used, but in these template files no implicit type declarations exist. It is recommended to use these files as the starting point of writing subroutines for MARC.
- Various demonstration examples can be found in `demo`, `demo_ddm` and `demo_table` folders.
- A few installation testing examples are also provided within the folder `test_ddm`.
- The `pidump` folder contains the source files for a utility program with the same name which is used to work on post-files of MARC. The executable version is located in the `bin` folder.
- Various library files are located in the `lib`, `lib_shared`, `intelmk1` and `intelmpi` folders.
- Executable files of MARC are located in the `bin` folder.

As mentioned, MENTAT files are organized in a separate parent folder (Fig. 2.3) in which the following subdirectories are located:

- The `bin` subdirectory contains the executables and the batch files which can be run within MENTAT by specific commands, e.g. a job can be submitted using either `submit1.bat`, `submit2.bat` or `submit3.bat` and these files can be manipulated to arrange a differently-configured run. These files originate from the `submit.bat` file in this subdirectory. Note that the main batch file for running MENTAT is the `mentat.bat` file which is also located in the same subdirectory.
- The `help` subdirectory contains the help files for MENTAT.
- The `materials` and `materials_pre2010` folders contain several material properties saved in files with the `.mat` extension. The latter folder contains the same files as the former but saved in the old format.

---

<sup>3</sup>Message Passing Interface.



**Fig. 2.3** Subdirectories of MENTAT

- The interface menu files for MENTAT are located in the menus subdirectory and are saved with the extension `.ms`.
- The python subdirectory contains the files for the PYTHON programming language which can be used for scripting purposes within MENTAT.
- The shlib subdirectory contains the shared library files.

## 2.2 The Input File

The *input file* or sometimes called the *data-file*, is a text file containing the finite element model with all the corresponding data, i.e. the *parameters* and the *options* which have been selected during the process of modeling.

At the beginner level, this file is usually generated by the GUI of the package, i.e. MENTAT, but at more advanced levels it can completely, or at least in part, be created and edited by the user.

In any computer programming language, interpretable commands and the necessary data are passed to the compiler via a source-file. The compiler is in charge of translating the commands and the related data to machine code or at the most basic level, to binary codes.

Similarly, MARC plays the role of the interpreter of the input file and since MARC is written in FORTRAN, the syntax of the input file is very similar to FORTRAN programming code. Each line of the input file is called a *block* or a *data block*. Commands in the input file are usually organized in several sequential blocks and are recognized by their *keywords*. In other words, a command is identified by its specific keyword followed by its required data. MARC reads each block of the input file as an alphanumeric string and then interprets it as a combination of keywords, integer numbers or floating point numbers. In conclusion, the keywords represent the commands and

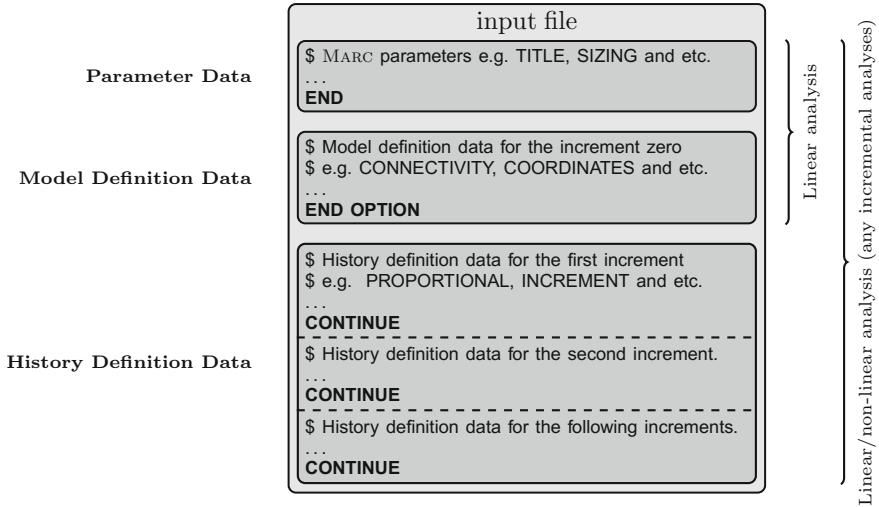


Fig. 2.4 Input file structure in MARC

the data can be passed to MARC as either alphanumeric values, integer numbers or floating point numbers.

### 2.2.1 Grouped Structure

Three distinctive groups of commands are distinguishable in the input file structure: the *parameter data*, the *model definition data* and the *history definition data*, as illustrated in Fig. 2.4. Each one of these groups is composed of specific commands which are organized in several blocks. Specific keywords are allowed in each group. A *parameter* is a keyword which is used in the parameter data section of the input file whereas an *option* can be used in either one of the model definition data section or history definition data section. Using the parameter data group and the model definition group is enough to handle a simple linear analysis if no incremental loading is required. Additional increments will be specified in the history definition group.

It is worth mentioning that in order to reduce the size of the input file, default values are defined for various features. In other words, most of the keywords are optional and they are used only if a value other than the default one is preferred. This will help to maintain the small size and the simple structure of the input file.

The input file starts with the first mandatory group of blocks, i.e. the parameter data group. In this section, several attributes can be set by means of different parameters, among which the following are worth mentioning:

- TITLE, defines the output title,
- ELEMENTS, defines up to 14 element types used in the analysis,



- ALLOCATE, allocates the necessary memory for the job to start,
- SIZING, sets the maximum number of nodes and elements,
- VERSION, selects the version of the input file,
- EXTENDED, allows working with large and/or higher precision models, and
- END parameter which marks the end of the section.

There is a rather long list of parameters that are used to set different preferences in this section such as analysis type, rezoning, input/output control and even for modifying default values. However, the minimum parameters to be used are TITLE, ELEMENTS and END.

The second mandatory section of the input file is the model definition section. This section, in the most concise form, contains the necessary options for obtaining an initial elastic solution, a so-called *zero-increment solution* of the problem in-hand. Thus, it covers a wide variety of information such as the geometry, nodal point data, boundary conditions, material data, contacts, error controlling, post-processing, print-out options etc. Some of the common options of this section can be listed as follows:

- GEOMETRY, specifies the geometrical data,
- COORDINATES, assigns coordinates to nodes,
- CONNECTIVITY, specifies the nodes that compose an element,
- DEFINE, defines a set and associates members to it,
- POST, creates a file for post-processing,
- OPTIMIZE, invokes a bandwidth optimizer for the stiffness matrix,
- INCLUDE, inserts an external file at the exact place of this option,
- PRINT, activates the printout of various useful items for debugging,
- NO PRINT, suppresses element and nodal output,
- PRINT ELEMENT, selected elements and the corresponding selected quantities will appear in the output file,
- PRINT NODE, selected nodes and the corresponding selected quantities will appear in the output file,
- SUMMARY, generates a summarized report of the increment,
- RESTART, used to write restart data to a file or read the data from a file, and
- the END OPTION keyword which marks the end of the section.

The last section of the input file is the optional history definition section. After obtaining the zero-increment solution, controlling the flow of MARC will be done through the options provided in this section. This group of options can apply multiple sequential loadcases or any alterations to the initial model such as a change in the boundary conditions. Each one of the loadcases ends with the keyword CONTINUE and can be applied to a single increment or multiple ones; the latter is usually done automatically. The following is a list of some allowable *options* of the history definition group:

- NEW, used to change the format of the input file from EXTENDED to default or vice versa,

- INCLUDE, POST, NO PRINT, PRINT ELEMENT, PRINT NODE, SUMMARY and RESTART have the same effect as explained in the previous section,
- CONTROL, an important option to set the convergence parameters of the current loadcase,
- PARAMETERS, sets quite a few finite element parameters for advanced users,
- LOADCASE, sets the boundary and initial conditions active in the current loadcase, and
- CONTINUE which marks the end of the information related to the current loadcase, and is the starting point for the data related to the next loadcase.

As it has already been mentioned, there are some common options which can be used in either model definition or history definition group such as the POST option. In addition, there are common keywords in both parameter and history definition groups such as TITLE. Another frequently used keyword is the COMMENT keyword or the Dollar sign (\$) which are used by the user to add informative comments anywhere in the input file.

### 2.2.2 *Format Conventions*

MARC recognizes the parameters and the options of the input file in either lower- or upper-case letters. Each line of the input file is structured as a *data-block* and may consist of one or several *entries*. Each entry is either an alphanumeric keyword or a piece of data which can be an alphanumeric value (A), an integer number (I), or a floating point number (F). Each data-block can be structured in *fixed format* or *free format*; although a mixture of these formats can be used, only one of them is allowed in a single line of text. Similar to the fixed format of FORTRAN programming convention, a fixed number of columns are designated to each entry in the fixed format. The following points are mentionable with regards to the fixed format:

- Each line consists of a total of 80 characters.
- An integer number is specified using five columns and a floating point number is specified using 10 or 15 characters.
- Each number must be right-justified with respect to its own field. Therefore, blank spaces are significant.

In contrast to this rather restrict format, the free format provides more flexibility in terms of field-width considerations. This format can be used by following some simple rules:

- A comma must follow after every single item of a data block (even a keyword).
- If a block consists of just one data item, an optional comma could follow that item.
- Blank spaces are insignificant.
- In order to skip a value in the data block, two consecutive commas shall be used.

In both of these formats, the following points must be considered to introduce a number:

| Format                         |      | Data Entry | Entry  |
|--------------------------------|------|------------|--|
| Fixed                          | Free |            |  |
| <b>1st data block</b>          |      |            |  |
| 1-11                           | 1st  | A          | Enter the word COORDINATES.  |
| <b>2nd data block</b>          |      |            |  |
| 1-5                            | 1st  | I          | Enter the maximum number of coordinate directions to be read in per node; defaults to the number of coordinates per node. Repeated COORDINATES blocks need not have the same value in this field.                    |
| 6-10                           | 2nd  | I          | Enter the number of nodal points read-in in this option; (optional) default to the number of nodes in the mesh.  |
| 11-15                          | 3rd  | I          | Enter the unit number for input of coordinates; defaults to the file number specified on the <i>INPUT TAPE</i> parameter.  |
| 16-20                          | 4th  | I          | Set to 1 to suppress print-out of nodal coordinate list during this option input.  |
| <b>3rd data block</b>          |      |            |  |
| One data line per nodal point. |      |            |  |
| 1-5                            | 1st  | I          | Nodal point number.  |
| 6-15                           | 2nd  | F          | Coordinate 1.  |
| 16-25                          | 3rd  | F          | Coordinate 2.  |
| 26-35                          | 4th  | F          | Coordinate 3.  |
| 36-45                          | 5th  | F          | Coordinate 4.  |
| Etc.                           |      | Etc.       | See library element description in <i>Marc Volume B: Element Library</i> for the definition of coordinates for a particular element.<br>Input 6 coordinates per data line; continuation data lines in format 6E10.5. |

**Fig. 2.5** Format for the COORDINATES option. Adapted from [24]

- Floating point numbers can be specified with or without exponents but in either case, using a decimal point is mandatory.
- The decimal point distinguishes between an integer and a floating point without an exponent. Although MARC tries to compensate for the errors, this is an advised approach.
- The exponent of a floating point follows after the character E, D or a plus sign (+).

The correct format for all of the available options and parameters are explained thoroughly in [24], but as an example, the COORDINATES option of the model definition group will be discussed to further clarify the concept. This option is used to specify the coordinates of the nodes in the model. The format of this option is illustrated in a table in Fig. 2.5. The first column of the table is the number of each fields in the fixed format. The second column indicates the sequence of fields in the free format. The third column is the type of the field which is either an alphanumeric, an integer or a floating point number represented with A, I or F, respectively. The last column is a short explanation of each data field. The fields of the COORDINATES option are categorized in three data blocks: the first one is the keyword, the second one consists of four integers to specify the maximum number of degrees of freedom for each node, set the number of total nodes, specify the unit number to read the node coordinates from and to set the print out capability. The following blocks are repetitive blocks each consisting of the number of the node followed by its coordinates. Based on this format, a sample COORDINATES option can be written in the fixed format as follows:

```

1 $ Lines 2 to 4 are used as a guide for the fixed format
2 $ 5 10 15 20 25 30 35 40
3 $2345678901234567890123456789012345678901234567890
4 $ | | | | | | | |
5 $ 1st data block
6 COORDINATES
7 $ 2nd data block
8 3 40 0 1
9 $ 3rd data block
10 1 1. 20. 300. $ (1.0,20.0,300.0)
11 2 1.0 20.0 300.0 $ (1.0,20.0,300.0)
12 3 1.E0 2.E1 3.E2 $ (1.0,20.0,300.0)
13 4 1.D0 2.D1 3.D2 $ (1.0,20.0,300.0)
14 5 1.00000+0 2.00000+1 3.00000+2 $ (1.0,20.0,300.0)
15 6 1.00000+02.000000+13.000000+2 $ (1.0,20.0,300.0)
16 7 1.00 2.00 3.00 $ (1.0,20.0,300.0)
17 8 1 2 3 $ (1.0,2.0,3.0)
18 9 1. 2.0E1 3.0000+3 $ (1.0,2E10,3E30)
19 101 2 3 $ (1.0,2.0e9,3.0e9)
20 11 1e1 2e2 3e3 $ (1.0,20.0,300.0)
21 12 1d1 2d2 3d3 $ (1.0,20.0,300.0)
22 ...
23 40 0. 0.00000+0 0.E1
24 ...
25 $Indicates the output unit file for the summary is 8
26 SUMMARY 8
27 ...

```

In this listing, different examples of inputs for floating point numbers are demonstrated. The result of each line is shown as a comment in the same line. Note that if the right-justification rule is not considered, MARC fills the empty spaces with zeros and the result might be far away from what is intended for (line 19). In the listing, just to clarify this example, a pipeline character (|) is used as a mark for every five column; informative comments are printed in gray. The first data block is the keyword of the option, i.e. COORDINATES. The second data block indicates that 3 coordinates will be specified in the default input file for each of the 40 nodes and the print-out of the node list is disabled. The following blocks consists of 40 node numbers and the corresponding coordinates. The same listing can be rewritten using the free format:

```

1 $ 1st data block
2 COORDINATES
3 $ 2nd data block
4 3,40,0,1,
5 $ 3rd data block
6 1,1.,20.,300., $ (1.0,20.0,300.0)
7 2,1.0,20.0,300.0, $ (1.0,20.0,300.0)
8 ...
9 $Indicates the output unit file for the summary is 8
10 SUMMARY,8,
11 ...

```

### 2.2.3 Extended Precision Mode

For most of the usual cases, using the default precision of MARC would be sufficient. However, for models with more than 99,999 elements, using extended precision will be required. In order to turn the extended mode on, the EXTENDED keyword is used

in the model definition section. Therefore, the needed space will be doubled for every piece of data in fixed format, i.e. for integer numbers, floating point numbers and characters, namely:

- the total number of characters in each line is doubled, i.e. 160 characters,
- integers will occupy 10 columns instead of 5,
- floating point numbers will occupy 20 or 30 instead of 10 or 15, and
- all character strings will be written using 20 characters instead of 10, e.g. the keywords.

The previous example can be re-written in the extended mode. Obviously, the free format will not be affected by the extended mode but the fixed format will be changed to the following:

```

1  $ 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
2  $ | | | | | | | | | | | | | | |
3  $ 1st data block
4  COORDINATES
5  $ 2nd data block
6  3 40 0 1
7  $ 3rd data block
8  1 1.000000000000000+0 2.000000000000000+1 3.000000000000000+2
9  ...
10 $Indicates the output unit file for the summary is 8
11 SUMMARY 8
12 ...

```

Note that in line 11, the SUMMARY keyword field starts from column 1 to column 20 and the integer number is right-justified in the field which consists of columns 21 to 30.

The extended precision mode can be selected via MENTAT by checking the corresponding option through the following:

③ Jobs > Jobs > Properties > Run > Advanced Job Submission > Extended Precision

## 2.2.4 Modifying the Input File

As previously stated in Sect. 2.2, there are two ways of creating an input file: by MENTAT or manually. For the latter, a simple text file editor could be used to edit the file with the .dat extension. The advised method, at least at the beginner level, is to create a model by MENTAT and to apply further modifications manually. There are several ways to obtain the input file of a model:

1. The current model can be exported to an input file using the following:
  - ① File > Export > Marc Input
2. It can be manually asked to write the input file of the current model by the following:
  - ③ Jobs > Jobs > Properties > Run > Advanced Job Submission > Write Input File

3. Submitting a job within MENTAT will automatically create or overwrite the current input file.

With these methods, one can create the input file from a model and then edit it with a text editor. It is also possible to modify the input file using MENTAT by one of the following methods:

1. After creating the input file using one of the previously stated methods, the editing process of the input file can be done by the following:

③ Jobs ▸ Jobs ▸ Properties ▸ Run ▸ Advanced Job Submission ▸ Edit Input File

After this process, the edited version of the input file will not run by the usual submission of the job because submitting a job overwrites the modified input file. The solution is achieved by *executing* the job instead of submitting it. This will analyse the modified input file without overwriting it and can be done using the following:

③ Jobs ▸ Jobs ▸ Properties ▸ Run ▸ Advanced Job Submission ▸ Execute (1)

2. If extra lines have to be added to the parameter definition or the model definition part of the input file, it can be done using the following:

③ Jobs ▸ Jobs ▸ Properties ▸ Input File Text

Note that it is not possible to remove any lines by this method.

3. A very neat way of adding lines to the input file is including an external text file. This included file will always be considered in a job submission and it can be added in MENTAT using the following:

③ Jobs ▸ Jobs ▸ Properties ▸ Include File

### 2.2.5 Table-Driven Input

Most of the time, especially in non-linear analyses, it is required to specify a quantity such as the value of a boundary condition as a function of some other independent variable, for example, the time or location. Generally, such an association can be specified using several input file options but using the *table* feature of MARC usually reduces the number of required options and saves some time and effort. The style of using tables in the input file is referred to as the *table-driven input*, whereas the earlier style, used in the pre-2003 versions, was called the *non-table-driven input* or the *old style*. The old style is still supported by the current versions of MARC/MENTAT.

In the table-driven style, a dependent quantity can be specified as a function of several independent values in the form of a table. Currently, MARC supports tabular data to be used in defining either a material behavior, a boundary condition or a contact as function of up to four independent variables. However, these variables must have a meaningful association, for instance, requesting the modulus of elasticity as a function of plastic strain is not supported (see [25]).

**Table 2.4** Input file options with tabular input capability

| Boundary condition options |                        |                      |
|----------------------------|------------------------|----------------------|
| ADD RIGID                  | FIXED TEMPERATURE      | POINT CHARGE         |
| CHANGE STATE               | FIXED VOLTAGE          | POINT CURRENT        |
| DIST CHARGES               | FOUNDATION             | POINT CURRENT-CHARGE |
| DIST CURRENT               | HOLD NODES             | POINT FLUX           |
| DIST FLUXES                | INIT CURE              | POINT LOAD           |
| DIST LOADS                 | INIT PRESS             | POINT MASS           |
| DIST MASS                  | INIT STRESS            | POINT SOURCE         |
| DIST SOURCES               | INITIAL DISP           | POINT TEMP           |
| EMISSIVITY                 | INITIAL FICTIVE        | QVECT                |
| FILMS                      | INITIAL PLASTIC STRAIN | RESTRICTOR           |
| FIXED ACCE                 | INITIAL STATE          | SINK POINTS          |
| FIXED DISP                 | INITIAL TEMP           | THICKNESS            |
| FIXED EL-POT               | INITIAL VEL            | VELOCITY             |
| FIXED MG-POT               | LATENT HEAT            | WELD FLUX            |
| FIXED POTENTIAL            | PERMANANET             |                      |
| FIXED PRESSURE             | PIEZOELECTRIC          |                      |
| Contact options            |                        |                      |
| CONTACT                    | CONTACT TABLE          | THERMAL CONTACT      |
| Material behavior options  |                        |                      |
| ANISOTROPIC                | INITIAL PC             | POWDER               |
| ARRUDBOYCE                 | INITIAL PORE           | PRESS FILM           |
| CHANGE PORE                | INITIAL POROSITY       | RELATIVE DENSITY     |
| COHESIVE                   | INITIAL VOID RATIO     | SHAPE MEMORY         |
| CRACK DATA                 | ISOTROPIC              | SOIL                 |
| CREEP                      | MOONEY                 | STRAIN RATE          |
| FAIL DATA                  | OGDEN                  | TEMPERATURE EFFECTS  |
| FOAM                       | ORTHO TEMP             | VISCOCREEP           |
| GENT                       | ORTHOTROPIC            | VOID CHANGE          |
| HYPOELASTIC                | POROSITY CHANGE        |                      |

Many options in MARC are provided in both table-driven and non-table-driven formats. A brief list of these options is available in Table 2.4. However, not all input file options support this capability. Therefore, as an alternative to the table-driven approach, several non-tabular model definition options may be used in conjunction with some user subroutines to produce the same effects. For instance, the easiest way to introduce an elastic-plastic material model with isotropic hardening is defining a table for the yield stress as a function of the equivalent plastic strain and then assigning it to the plastic behavior of the material. If the table style is not preferred then using either the WORK HARD input file option or the WKSLP user subroutine is

suggested to employ the hardening law. Although the table-driven options can cover many usual cases, subroutines support the most general cases.

In MENTAT, it is possible to select or change the style of the input file, if possible, by selecting the proper style in the Run Job dialog. The following commands are used to open this dialog:

③ Jobs > Jobs > Properties > Run

And then, the input file can be re-written in the new style by following these commands:

③ Jobs > Jobs > Properties > Run > Advanced Job Submission > Write Input File

Selecting the table-driven style will add the TABLE parameter to the parameter group of the input file, indicating that the new style will be used in particular options relating to the boundary conditions, material definitions and the contact option of the model. Note that it is possible to use the old style in one area and tabular data in others. However, by switching the input file back to the old style, the TABLE parameter will be removed. For instance, the following line indicates that the table-driven style will be used to define boundary conditions, material definition and contact options:

```
1 table ,0,0,2,1,1,0
```

The last data entry of this option specifies the *table reference value*, which is zero in this case. This value is used in conjunction with the TABVA2 utility subroutine. A reference value for a table or a *multiplication factor* is just a scale factor. The values of a table are multiplied by this factor upon extraction. This value is saved in the table common block by the name jtabmult. In this option, a zero value indicates that a reference value of zero for the subroutine is considered as one. If 1 is used as the last data entry of this option then a zero reference value is treated as zero by the subroutine. The TABVA2 subroutine is used to obtain the value of a table within a subroutine. The following syntax is used for this subroutine:

```
1 CALL TABVA2 (refvalue , evalue , idtable , 0 , 0)
```

The reference value is used as the first argument of this subroutine (refvalue) which acts as a scale factor as mentioned. The value of the function for the selected table (idtable) will be returned via the evalue variable. The value of the independent variables must be specified in the common block ctable. As listed in Table 2.5, the variable of each independent variable is indicated in the Variable column; for example, if the equivalent plastic strain is used in a table then the value of that must be assigned to the eqpl variable. In addition to the TABLE parameter, in order to explicitly define a set of tabular data, the TABLE option is used in the model definition group of the input file. This option enables the user to define a table as some data points or as a function. In other words, the dependent value of the table is defined as a function of up to four independent meaningful variables. An independent variable can be one of the variables listed in Table 2.5. The function is defined by either the *piecewise linear mode* or the *equation mode*.

If some data points are available, e.g. the results of a tensile test, then the piecewise linear mode can be easily used. In this case, a linear interpolation is used to obtain



**Table 2.5** Independent variable types of the table-driven input. Adapted from [26]

| #  | Description   | Variable | #  | Description  | Variable  |
|----|---|----------|----|--|-----------|
| 1  | Time  | timec    | 28 | Contact force  | forbd     |
| 2  | Normalized time                                     | timen    | 29 | Contact body   | torbd     |
| 3  | Increment number                                    | xincc    | 30 | Normal stress  | signormal |
| 4  | Normalized increment                                | xincn    | 31 | Voltage  | volti     |
| 5  | $x$ coordinate                                      | xyz(1)   | 32 | Current  | curri     |
| 6  | $y$ coordinate                                      | xyz(2)   | 33 | $(\frac{\text{current radius}}{\text{radius of throat}})^2$        | rdnrrn    |
| 7  | $z$ coordinate                                      | xyz(3)   | 34 | $\xi_p$ pyrolysis damage   | pyrodam   |
| 8  | $s = \sqrt{x^2 + y^2 + z^2}$                        | su       | 35 | $\phi_w$ water vapor fraction                                      | xdv       |
| 9  | $\theta$ angle                                      | angl     | 36 | Coking damage  | xdc       |
| 10 | Mode number   | xmoden   | 37 | Gasket closure distance  | clou      |
| 11 | Frequency   | frequ    | 38 | Displacement magnitude   | displ     |
| 12 | Temperature   | tempi    | 39 | Stress rate  | strsrte   |
| 13 | Function  | xfun     | 40 | Experimental data  | expdata   |
| 14 | Fourier   | xfur     | 41 | Porosity   | voidrtab  |
| 15 | $\bar{\epsilon}^p$ equivalent plastic strain        | eqpl     | 42 | Void ratio   | voidrtab  |
| 16 | $\dot{\bar{\epsilon}}$ equivalent strain rate       | eqplr    | 43 | Equivalent creep strain rate                                       | eqcrpsx   |
| 17 | $B_g = \dot{m}/\rho m$<br>normalized mass flow rate | amdotnam | 44 | Minor principal total strain                                       | formlmt   |
| 18 | arc length  | arcdist  | 45 | Distance from neutral axis   | xxshell   |
| 19 | Relative density                                    | xdensity | 46 | Normalized distance from neutral axis                              | xnshell   |
| 20 | $\bar{\sigma}$ equivalent stress                    | eqstr    | 47 | Local $x$ -coordinate of layer point for open/closed section beams | xxc       |
| 21 | Magnetic induction                                  | babs     | 48 | Local $y$ -coordinate of layer point for open/closed section beams | yyc       |
| 22 | Velocity  | vel      | 49 | 1st isoparametric coordinate                                       | xiso      |
| 23 | Particle diameter                                   | diam     | 50 | 2nd isoparametric coordinate                                       | yiso      |
| 24 | $x_0$ coordinate                                    | corx     | 51 | Wavelength   | xwaveln   |
| 25 | $y_0$ coordinate                                    | cory     | 52 | Creep strain   | eqcrpz    |
| 26 | $z_0$ coordinate                                    | corz     | 53 | Pressure or primary quantity in diffusion                          | pressrx   |
| 27 | $s_0 = \sqrt{x_0^2 + y_0^2 + z_0^2}$                | cors     | 54 | Equivalent strain rate (nonNewtonian viscosity)                    | eqfstn    |

(continued)

**Table 2.5** (continued)

| #          | Description   | Variable     | #  | Description                  | Variable    |
|------------|---|--------------|----|------------------------------|-------------|
| 55         | Normalized arc distance                               | arcnorm      | 67 | Degree of cure               | degofcure   |
| 56         | Distance to other contact surface (near contact only) | dnear        | 68 | Magnetic field intensity     | habs        |
| 57         | Term of series  | ijkl         | 69 | Equivalent mechanical strain | eqms        |
| 58         | Hydrostatic stress                                    | hydstr       | 70 | 1st strain invariant         | dstnin1     |
| 59         | Hydrostatic strain                                    | hydstrn      | 71 | 2nd strain invariant         | dstnin2     |
| 60         | $B_{g,p} = \dot{m}_g, p/\alpha m$                     | amdotgpnam   | 72 | 3rd strain invariant         | dstnin3     |
| 61         | $B_{g,w} = \dot{m}_g, w/\alpha m$                     | amdotgwnam   | 73 | Any strain component         | elocal      |
| 62         | 2nd state variable                                    | statevars(2) | 74 | Damage                       | damage      |
| 63         | 3rd state variable                                    | statevars(3) | 75 | Accumulated crack growth     | crackgrowth |
| 64         | 4th state variable                                    | statevars(4) | 76 | Relative sliding velocity    | relselvel   |
| 65         | 5th state variable                                    | statevars(5) | 77 | Damping ratio                | dampratio   |
| 66         | Loadcase number                                       | xldcas       | 78 | log frequency (base 10)      | frequ       |
| -1 to -100 | Parametric variable 1-100                             |              |    |                              |             |

the values in between data points. However, for the points after the last data point or before the first one, either an extrapolation or the closest data point is used, i.e. for the points after the last data point, the value of the last one is used.

For instance, it is a common approach to define a linear loading in a static structural analysis. This can be done using one independent variable of type TIME and the following lines in the model definition part of the input file:

```

1 TABLE, RAMP
2 1, 1, 0, 0, 2
3 1, 2, 2, 0, 0, 2, 0, 0, 2, 0, 0, 2
4 0.0E0, 0.0E0
5 1.0E0, 1.0E0
    
```

Although using just data-points would be enough for many cases, it is sometimes more convenient to introduce the table as a function. If a mathematical equation exists for our case then the equation mode might be just what we are looking for. The equation can be defined in terms of four independent variables, i.e. V1, V2, V3 and/or V4 combined together using the operators listed in Table 2.6. For instance, a sinusoidal load as a function of time with a maximum value of 20 can be readily introduced by the function  $20*\sin(2.*\pi*V1)$  in which V1 is the first independent value of type TIME. The following lines in the model definition part will add such a load to the model:

```

1 TABLE, SINLOAD
2 1, 1, 0, 0, 3
3 1, 2, 2, 0, 0, 2, 0, 0, 2, 0, 0, 2
4 20* sin (2.* pi*V1)
    
```

**Table 2.6** Symbols, operators and mathematical functions for declaring mathematical equations of tables in equation mode. Adapted from [25]

| Symbol                 | Description                                      |
|------------------------|--|
| Operators              |  |
| +                      | Addition   |
| −                      | Subtraction                                      |
| *                      | Multiplication                                   |
| /                      | Division   |
| ^                      | Exponential                                      |
| !                      | Factor   |
| %                      | Mod  |
| Constants              |  |
| pi                     | $\pi$  |
| e                      | Exponent   |
| tz                     | Offset temperature                               |
| q                      | Activation energy                                |
| r                      | Universal gas constant                           |
| sb                     | Stefan Boltzman constant                         |
| Mathematical functions |  |
| sin(x)                 | $\sin(x)$ , $x$ in radians                       |
| cos(x)                 | $\cos(x)$ , $x$ in radians                       |
| tan(x)                 | $\tan(x)$ , $x$ in radians                       |
| dsin(x)                | $\sin(x)$ , $x$ in degrees                       |
| dcos(x)                | $\cos(x)$ , $x$ in degrees                       |
| dtan(x)                | $\tan(x)$ , $x$ in degrees                       |
| asin(x)                | $\sin^{-1}(x)$ in radians                        |
| acos(x)                | $\cos^{-1}(x)$ in radians                        |
| atan(x)                | $\tan^{-1}(x)$ in radians                        |
| atan2(x,y)             | Inverse tangent of the point $(x, y)$ in radians |
| dasin(x)               | $\sin^{-1}(x)$ in degrees                        |
| dacos(x)               | $\cos^{-1}(x)$ in degrees                        |
| datan(x)               | $\tan^{-1}(x)$ in degrees                        |
| datan2(x,y)            | inverse tangent of the point $(x, y)$ in degrees |
| log(x)                 | $\log_{10}(x)$                                   |
| ln(x)                  | $\ln(x)$   |
| exp(x)                 | $e^x$  |
| sinh(x)                | $\sinh(x)$                                       |
| cosh(x)                | $\cosh(x)$                                       |
| tanh(x)                | $\tanh(x)$                                       |
| asinh(x)               | $\sinh^{-1}(x)$                                  |
| acosh(x)               | $\cosh^{-1}(x)$                                  |
| atanh(x)               | $\tanh^{-1}(x)$                                  |

(continued)

**Table 2.6** (continued)

| Symbol   | Description  |
|----------|--|
| sqrt(x)  | $\sqrt{x}$   |
| rad(x)   | Converts degrees to radians  |
| deg(x)   | Converts radians to degrees  |
| abs(x)   | $ x $  |
| int(x)   | Truncates the value to whole   |
| frac(x)  | Takes the fractional value   |
| max      | Takes the maximal value  |
| min      | Takes the minimal value  |
| mod(x,y) | The remainder of $x$ based on $y$ , i.e. $\text{mod}(x,y) = x - y * \text{int}(x/y)$ |

Note that the maximum length of a formula is 80 characters which is doubled in the extended mode.

### 2.2.6 Items, Sets and Numbering

There are 12 types of *items* which can be used in conjunction with the capabilities of MARC/MENTAT:

1. element numbers,
2. node numbers,
3. degrees of freedom numbers (DOFs),
4. integration point numbers,
5. layer numbers,
6. increment numbers,
7. points,
8. curves,
9. surfaces,
10. bodies,
11. edges, and
12. faces.

In order to use some functions within MARC/MENTAT, it is required to make a list of these items, e.g. a list of nodes for a boundary condition. For this purpose, a *set* is used which contains either several items or a combination of other sets. The terms *sublist* or *subset* are also used, equivalently. In this section, defining sets and the numbering convention of these items will be discussed.

Sets are defined using the DEFINE input file option followed by the list of its members. The list can be defined by syntaxes declared in Table 2.7 and combined by means of the *subset connectors*. For instance, consider the following lists:

**Table 2.7** Syntax used in MARC sets

| Syntax                     | Description   |
|----------------------------|---|
| Set declaration            |   |
| a [, b [,c]]               | A list of items separated by commas   |
| a:b                        | Element:edge/Element:face pair used to specify edges/faces, a is element ID and b is edge/face ID |
| a[:d] TO b[:d] [BY c]      | A list of items starting from a to b i.e. a, a+1, a+2,..., b-2, b-1, b                            |
| or                         | c indicates the step increase i.e. a, a+c, a+2*c,..., b-2*c, b-c, b                               |
| a[:d] THROUGH b[:d] [BY c] | d indicates the edge number i.e. a:d, (a+1):d,..., (b-2):d, (b-1):d, b:d                          |
| Set connectors             |   |
| set1 AND set2              | Final set is consisted of all items in both set1 and set2   |
| set1 INTERSECT set2        | Result is the common items of set1 and set2   |
| set1 EXCEPT set2           | Result is all items in set1 which are not listed in set2  |

```

1  DEFINE,NODE,SET,list01
2  1 TO 10
3  DEFINE,NODE,SET,list02
4  10 TO 5
5  DEFINE,NODE,SET,list03
6  1 TO 10 BY 2
7  DEFINE,NODE,SET,list04
8  1 TO 10 BY 3
9  DEFINE,NODE,SET,list05
10 list03 AND list04
11 DEFINE,NODE,SET,list06
12 list03 AND list04 INTERSECT list02
13 DEFINE,NODE,SET,list07
14 list03 AND list04 INTERSECT list02 EXCEPT list04
15 DEFINE,EDGE,SET,edgelist
16 1:1 TO 5:1

```

Based on these definitions, the defined lists consist of the following items:

list01 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

list02 = 5, 6, 7, 8, 9, 10

list03 = 1, 3, 5, 7, 9

list04 = 1, 4, 7, 10

list05 = 1, 3, 4, 5, 7, 9, 10

list06 = 5, 7, 9, 10

list07 = 5, 9

edgelist = 1:1, 2:1, 3:1, 4:1, 5:1

The first block, when defining a set, starts with the DEFINE input file option. Next, the keyword NODE indicates that a list of nodes is defined. Then there is the keyword

NODE followed by the name of the set; list01 in the first example. A set name is of type CHARACTER with a maximum length of 32. The second block is the list of the members which can be any combination of the predefined set names and members of the model. Note that only the sets which are already defined can be used in the current definition.

Generally, a list can be either *sorted* or *unsorted*. Most of the sets used in MARC are sorted lists, except for the list of nodes in the TYING and EXCLUDE options (i.e. FNDSQ), the list of degrees of freedom in FIXED DISP option (i.e. DOF) and the unsorted list of elements (i.e. ELSQ). Putting aside these exceptions, MARC sorts the items of a sorted list or any combination of lists incrementally prior to use. Note that except and intersect sublist connectors cannot be used in an unsorted list.

An edge/face ID can be expressed in either MARC or MENTAT convention: the edge/face ID is increased by one when transferred from MARC to MENTAT, e.g. an edge pair such as 10:0 in MARC is equal to a 10:1 representation in MENTAT. The list of edges in the example is defined using the MARC convention. A similar convention is valid for the orientation of the surfaces and curves; for more information on this, refer to the documentation of MARC (see [25]).

When dealing with MARC/MENTAT entities, especially elements and nodes, two types of numbering are used. One is the numbering that is used by the user, called *external* or *user ID*. The other one is the internal numbering which is used by MARC itself, namely a *consecutive numbering system* starting from one.

If the user IDs of elements/nodes start from one and they proceed sequentially, the internal numbering will be the same as the user IDs. This is the consecutive numbering system. In any other cases, the user IDs will not match the internal IDs, e.g. the user numbering starts from any number other than one. It is important to understand which of these two numbering systems are used because an argument indicating the number of an element/node may be required to be specified in either one of the systems.

The non-consecutive numbering system for the elements is flagged by means of the variable `nelids`, provided by the prepro common block. The value of this variable is set to a non-zero value equal to the total number of elements to indicate a non-consecutive numbering system. In this case, the internal numbering system is related to the user numbering via the `ielids_d` array provided by the `spacevec` common block.

In contrast, the consecutive numbering system is used when the flag is set to zero. In this case, the internal IDs are the same as the user IDs and no conversion is required. In the same manner, the flag `nnoids` is used to indicate the numbering system of the nodes and the array `inoids_d` is used to carry out the conversion for the non-consecutive case.

## 2.3 Subroutines

The most powerful aspect of MARC is its support of user-defined subroutines. A user-defined subroutine replaces the corresponding standard subroutine of the program and consequently, unconventional cases of analyses can be covered. A subroutine

in MARC is a FORTRAN subroutine with a predefined header, namely the arguments are declared based on the context. These arguments are used either with an input intent or an output intent; they are called *inputs* and *outputs*, respectively. The input is the data provided for the subroutine as the raw material of the calculations and the output is the data which must be available at the end of the subroutine, namely it is the result of running the subroutine. There are some arguments which act both as an input and an output. In such cases, the input should be updated during the execution of the subroutine.

Although most of the subroutines have inputs and outputs, some of them do not require any inputs/outputs, running at specific points of the analysis for specific purposes. For example, the UBGINC subroutine runs at the beginning of each increment and it can be used to set up some initial values.

The header of the subroutines and the required inputs/outputs are explained in [26]. In addition to these templates provided in the documentation of MARC, there is a template file for each subroutine which is located in the user subdirectory (Sect. 2.1.3). These two templates are slightly different in the sense that the one in the documentation is provided with an implicit type declaration while the other one does not use any implicit type declaring. Although using either works fine, the implicit type declaration makes the debugging process cumbersome and it is not advised. In addition, it is worth noticing that some of these so-called templates contain some default lines of code which usually come with enough comments to be modified appropriately. An example for such a template is the `tensof.f` file which can be referred to for more information.

### 2.3.1 *Activating Subroutines*

The philosophy behind user subroutines is to increase the flexibility of the package by attaching a custom piece of code to the solution procedure. The need for a subroutine in a complicated problem is usually indispensable but in order to develop an efficient subroutine, most of the time, an in-depth knowledge of finite element analysis is required. Around 200 subroutines are supplied by MARC to be engaged with user-defined calculations. These subroutines already exist inside MARC without having any lines of codes inside them, i.e. they do nothing by default. Therefore, they are called *dummy subroutines*. In order to engage them in the process of an analysis, the user must activate them with the appropriate coding. The FORTRAN compiler makes an object file out of this source code and then the FORTRAN linker makes an executable file by linking the code with the appropriate library files of MARC. This file will be incorporated during the solution of the model. The whole process actually is replacing the user's subroutine with the dummy one. Based on the type of the subroutine, one or more of the following methods can be used to activate it:

- Activating is done by selecting a setting in MENTAT. An option must be selected within the interface which writes an option to the input file for activation of the

subroutine. In order to set, for example, the behavior of a nonlinear spring/dashpot by the subroutine USPRNG, the following option must be selected within MENTAT:

③ Links ▸ Springs/Dashpots ▸ New ▸ Fixed DOF ▸ User Subroutine Usprng

- Using the appropriate parameter or option in the input file. Generally, using some keywords alters the sequence of an analysis in MARC. This process is called *setting a flag*. For example, using the USDATA option in the input file invokes the USDATA subroutine to initialize the user variables.
- Finally a group of subroutines are automatically invoked which means just by adding the appropriate subroutine header to the FORTRAN code, MARC will execute them. In the documentation of MARC, this type of subroutine is marked with a note indicating that *'No special flag is required in the input file'*. For example, the UEDINC subroutine is an automatic subroutine which runs at the end of each increment.

At least one single FORTRAN source file or one compiled executable file must be introduced if the job is submitted using MENTAT. The appropriate FORTRAN source file can be selected within MENTAT by the following chain of commands:

③ Jobs ▸ Jobs ▸ Properties ▸ Run ▸ User Subroutine File

MARC looks for the activated subroutines in the selected file upon a job submission. Although just one FORTRAN file can be included in the analysis, it may contain several subroutines. However, instead of using a file with all the subroutines, it is possible to include a text file with references to some other FORTRAN files. Suppose that three subroutines are required and each one of them is in a separate file. By creating a single text file with the following lines and selecting it as the subroutine file, the issue will be addressed:

```
1 #include "myfile1.f"
2 #include "myfile2.f"
3 #include "myfile3.f"
```

Note that the style is similar to that of the C programming language. Therefore, these lines are actually compiler directives and they are not FORTRAN statements. However, using the INCLUDE statement of FORTRAN, the same lines could appear such as the following:

```
1 INCLUDE "myfile1.f"
2 INCLUDE "myfile2.f"
3 INCLUDE "myfile3.f"
```

### 2.3.2 Structure of Subroutines

The structure of subroutines in MARC is the same as every subprogram in FORTRAN. However, while dealing specifically with subroutines of MARC, considering the following notes is recommended:



- Use the templates without any implicit type declaration, i.e. the templates inside the user subdirectory.
- Declare all the floating point numbers as real numbers of kind = 8 for the sake of maintaining compatibility with MARC.
- The general rule of using a template is to keep its original structure, i.e. the name and the type of the arguments, and only add the necessary code. However, sometimes it is necessary to change the name of the argument of the subroutine in order to resolve a name conflict. This conflict usually arises when using multiple common blocks with a variable named exactly as an argument. In other cases, it is easier just to avoid redeclaring any variables already defined as an argument of the subroutines.
- Try to use the concepts of structured programming and use modular programming to maintain the clarity of your code. It also helps to keep a library of your own subroutines for subsequent uses.
- In the documentation of MARC, arguments are called either a *required input* or a *required output*. A required input is an input argument of the subroutine and the required output is an output argument of that. Therefore, after carrying out the calculations the required outputs must be passed back to MARC. Usually, it is not necessary to calculate every required output but only the required ones with respect to the specific problem in-hand. Similarly, it is possible that based on the problem, just a few of required inputs are defined.

For example, consider the template of the FORCDT subroutine:

```

1  ! Part 1: Subroutine header
2      SUBROUTINE FORCDT(u, v, a, dp, du, time, dtime, ndeg, node, ug,
3      &                  xord, ncrd, iacflg, inc, ipass)
4  ! Part 2: Data Environment (Specification part)
5  #ifdef _IMPLICITNONE
6      IMPLICIT NONE
7  #else
8      IMPLICIT LOGICAL (a-z)
9  #endif
10 !
11     ** Start of generated type statements **
12     REAL*8 a, dp, dtime, du
13     INTEGER iacflg, inc, ipass, ncrd, ndeg, node
14     REAL*8 time, u, ug, v, xord
15 !
16     ** End of generated type statements **
17
18     DIMENSION u(ndeg), v(ndeg), a(ndeg), dp(ndeg), du(ndeg), ug(ndeg),
19     &          xord(ncrd)
20 ! Part 3: The executable construct (User code)
21
22     RETURN
23 ! Part 4: The contained area for internal subroutines i.e. CONTAINS statement
24 ! Part 5: The subroutine end statement.
25     END

```

In this listing, the header of the subroutine is followed by the specification part which contains the implicit statement and the attribute-oriented type declaration of the arguments. The executable part contains the user code and ends with the RETURN statement. Internal subprograms are also allowed in the contained area. The user must manage the inputs and outputs by programming a proper code in the executable

part. Other subroutines have a similar structure but different headers. More specific information can be obtained by referring to the documentation of MARC (see [23]).

### 2.3.3 Predefined Common Blocks of Marc

As mentioned earlier, a custom subroutine in MARC is replacing the default dummy subroutine in the background with a user-defined one. In most cases, there is more than one subroutine to be replaced, and most of the time, these incorporated subroutines have to interact with each other in a single analysis, i.e. some data within a subroutine must be shared between other subroutines. As introduced in the previous chapter, there are several ways to do this; using modules, the common blocks or data blocks. Although modules and data blocks share the data in an elegant way, MARC uses its own *pre-defined common blocks* to share the data regarding the model. By including a common block in a programming unit, the data will be accessible within that unit. The common blocks of MARC are located in its common subdirectory in several text files with the .cmn file extension. Each one of these files share pieces of data regarding the model and they can be used in subroutines to obtain additional required information about the model. For instance, in the *dimen* common block, the number of elements and nodes are determined by variables *numel* and *numnp*, respectively. In order to make these variables available in a subroutine, an `INCLUDE` statement is used followed by the name of the common block. For this particular case the `INCLUDE 'dimen'` statement should be used.

Although using predefined common blocks is imposed by MARC, a better approach is to share the data via modules, i.e. use the `INCLUDE` statements in a single module and then share the data by the `USE` statement. This way has the advantage of being concise because it is not required to use multiple `INCLUDE` statements. In addition, other auxiliary variables can be incorporated in the same module. A sample data-sharing module can be done as follows:

```

1  MODULE MarcCommonData
2  IMPLICIT NONE
3
4  INCLUDE 'dimen'
5  ...
6  CONTAINS
7  ...
8  END MODULE MarcCommonData

```

Several quantities have been made available through the predefined common blocks. Some of them are general quantities such as the total number of nodes in a mesh (*numnp*) which is available in all subroutines. A list of such globally-available quantities is presented in, but not limited to, Table 2.8. In this table, some variables are presented from the *creeps*, *dimen*, *concom*, *spaceset* and *iautcr* common blocks. The first two common blocks contain some general information of the model whereas the *concom* common block contains most of the advanced control *flags* of MARC. A flag is an integer variable which can be set to control the flow of the analysis. Usually, setting a flag on is done by assigning the value 1 to it and setting it off is possible

**Table 2.8** Selected globally-available quantities via common blocks. Adapted from [26]

| Variable name       | Description   |
|---------------------|---|
| creeps common block |   |
| cptim               | Total time at the beginning of increment  |
| timinc              | Time increment for this step  |
| time_beg_lcase      | Time at the beginning of the current load case  |
| time_beg_inc        | Time at the beginning of the current increment  |
| mcreep              | Maximum number of iterations for explicit creep   |
| jcreep              | Counter of the number of iterations for explicit creep  |
| dimen common block  |   |
| numel               | Number of elements in mesh  |
| numnp               | Number of nodes in mesh   |
| nintb               | Maximum number of integration points used in integration point calculation  |
| nintbmx             | Maximum number of nintb   |
| ndeg                | Number of degrees of freedom per node   |
| ndegmx              | Maximum number of degrees of freedom per node   |
| nnodmx              | Maximum number of nodes per element in the whole model  |
| neqst               | Maximum number of invariants per integration point  |
| ngens               | Number of strains per integration point   |
| nrcd                | Number of coordinates per node for the current element  |
| nrcdmx              | Maximum number of coordinates per node in the whole model (same for that of the integration point)  |
| nstrmx              | Maximum number of stress components per integration point   |
| ngenmx              | Maximum number of generalized strains   |
| idss                | Size of the element stiffness matrix  |
| nsxx                | Number of non-zero blocks stored in stiffness matrix  |
| maxnp/maxnpr        | Maximum number of connections to a node (for mass/specific heat matrix)   |
| maxqnp              | Maximum number of connections to a node (for optimization processes)  |
| neltyp              | Number of different element types   |
| maxall              | Current size of ints/vars array   |
| nusdat              | Number of user variables to be defined in USDATA  |
| nstrmz              | Maximum number of stresses stored per element   |
| nstrm3              | Maximum number of generalized stress/strain components per element (number of generalized components $\times$ number of integration points) |
| nstrm4              | Maximum number of state variables per element (number of layers $\times$ number of integration points)                                      |
| nstrm5              | Maximum number of words needed to store all stress-strain laws in every stress storage point per element                                    |
| nintps              | Number of integration points used on the post-file  |
| maxall              | Current size of ints/vars array   |
| concom common block |   |
| inc                 | Increment number  |
| incsub              | Sub-increment number  |
| icrpim              | Implicit creep  |
| iradrt              | Radial return   |

(continued)

**Table 2.8** (continued)

| Variable name | Description  |
|---------------|--|
| incext        | Flag indicating if currently a sub-increment is under process  |
| idyn          | Dynamic analysis type based on the DYNAMIC parameter   |
| ilem          | Indicates which part of the element assembly is under process  |
| loadup        | Control flag indicating if nonlinearity occurred during previous increment   |
| iupblg        | Control flag for the follower force option   |
| loaduq        | Control flag indicating if nonlinearity occurred   |
| ncycle        | Cycle number (accumulated in oasemb.f)   |
| iaxisymm      | Flag indicating an axisymmetric analysis   |
| iassum        | Assumed strain flag  |
| iradrtp       | Radial return flag for plastic material  |
| iupdatp       | Updated Lagrange flag for elastic-plastic material   |
| ismall        | Flag indicating a small displacement analysis  |
| jlshell       | Flag indicating if a shell element exists in the model   |
| icompso       | Flag indicating if a composite solid is in the mesh  |
| isetoff       | Flag indicating if beam/shell offset is applied  |
| jel           | Control flag indicating that the total force is applied in the increment   |
| ioffsetm      | Minimum value of the offset flag   |
| idinout       | Flag indicating if inside out elements should be deactivated   |
| lodcor        | Flag indicating if residual load correction is activated   |
| loadup        | Flag indicating if nonlinearity has occurred in the previous increment   |
| loaduq        | Flag indicating if nonlinearity has occurred in the current increment  |
| marmen        | Flag indicator of MARC used:<br>0 : for normal analysis 1 : as reader via MENTAT   |
| istpnx        | 1 if to stop at the end of increment   |
| lovl          | Control flag for determining the analysis phase (overlay name)<br>1 : memory allocation (omarc.f)      2 : model definition input (oaread.f)<br>3 : distribute load (opress.f)      4 : stiffness matrix (oasemb.f)<br>5 : solver (osolty.f)                6 : stress recovery (ogetst.f)<br>7 : output (oscinc.f)                8 : operator assembly (odynam.f)<br>13 : history definition input (oincdt.f) 14 : mass matrix (oasmas.f)<br>17 : vector transformation (oshtra.f) 20 : rezoning (orezon.f)<br>21 : convergence testing (otest.f)    22 : Lanczos (oeigen.f)<br>23 : global adaptive meshing |
| jactch        | Flag for the activation/deactivation of the elements<br>0 : normally/reset to 0 when assembly is done<br>1 or 2 : if elements are activated or deactivated<br>3 : if elements are adaptively remeshed or rezoned   |
| isolv         | Flag for the solver type<br>0 : profile direct solver    2 : sparse iterative<br>4 : sparse direct solver    8 : multifrontal direct sparse solver<br>9 : CASI iterative solver 10 : mixed direct/iterative solver   |

(continued)

**Table 2.8** (continued)

| Variable name   | Description  |
|---|--|
| ipass   | Control flag for the type of current iteration in a coupled analysis<br>-1 : reset to base values 0 : do nothing<br>1 : stress                      2 : heat transfer<br>3 : fluids                      4 : joule heating<br>5 : diffusion                  6 : electrostatic<br>7 : magnetostatic          8 : electromagnetics  |
| newton  | Newton-Raphson flag for the current iteration:<br>1 : full Newton-Raphson<br>2 : modified Newton-Raphson<br>3 : Newton-Raphson with strain correct<br>4 : direct substitution<br>5 : direct substitution followed by Newton-Raphson<br>6 : direct substitution with line search<br>7 : full Newton-Raphson with secant initial stress<br>8 : secant method<br>9 : full Newton-Raphson with line search |
| spaceset common block   |  |
| ndset   | number of total sets   |
| nsetmx  | maximum number of set  |
| nchnam  | maximum set name characters  |
| mxitmset  | maximum number of set items (pre-reader)   |
| setname(i)  | name of set i  |
| isetdat   | array containing information on all sets   |
| isetdat(1,i) type of set:   |  |
| 0 : sorted element list    10 : face list (no longer active)<br>1 : sorted node list      11 : unsorted element list<br>2 : integration point    12 : element:edge list<br>3 : layer list             13 : element:face list<br>4 : DOF list               14 : point list<br>5 : increment list       15 : curve list<br>6 : unsorted node list   16 : surface list<br>7 : frequency list       17 : cavity list<br>8 : entity list            18 : surface:orientation list<br>9 : body list              19 : curve:orientation list |  |
| isetdat(2,i) number of items in set i   |  |
| isetdat(3,i) sort flag for set i  |  |
| isetdat(4,i) boundary conditions flag for set i   |  |
| isetdat(5,i) number of items in set i (full model)  |  |
| isetdat(6,i) flag for remeshing update of set i   |  |
| iatucr common block   |  |
| loadcn  | Control flag for AUTO LOAD option  |
| totinc  | Total increment time for the current load increment  |
| totins  | Running counter of time for the current load increment   |

**Table 2.9** Element-based user-subroutines. Adapted from [26]

| List of subroutines |           |         |         |            |             |        |
|---------------------|-----------|---------|---------|------------|-------------|--------|
| ANELAS              | ANEXP     | ANKOND  | ANPLAS  | ASSOC      | CRPLAW      | CRPVIS |
| CUPFLX              | ELEVAR    | ELEVEC  | FILM    | FLUX       | FORCEM      | GENSTR |
| HOOKLW              | HOOKVI    | HYPELA2 | INITSV  | INTCRD     | NASSOC      | NEWSV  |
| ORIENT              | ORIENT2   | PLOTV   | REBAR   | SINCER     | TENSOE      | TRSFAC |
| UACTIVE             | UACOUS    | UADAP   | UARRBO  | UCOHEISIVE | UCOHEISIVET | UCOMPL |
| UCRACK              | UDAMAG    | UDELAM  | UELDAM  | UELASTOMER | UENERG      | UEPS   |
| UFAIL               | UFILM     | UFOUND  | UGENT   | UELOOP     | UFINITE     | UHTCOE |
| UHTCON              | UINSTR    | UMDCOE  | UMDCON  | UMOONY     | UMU         | UNEWTN |
| UOGDEN              | UPERM     | UPOWDR  | URESTR  | URPFLO     | USELEM      | USHELL |
| USIGMA              | USPCHT    | UVOIDN  | UVOIDRT | UVSCPL     | UVTCOE      | UVTCON |
| VSWELL              | UWELDFLUX | WKSLP   | YIEL    |            |             |        |

by assigning 0 as the value. However, other values are also possible. These flags either provide the user with more detailed information or they make it possible to fine-tune the properties of the analysis and to have more control of the procedures. For instance, the `lowl` flag indicates the analysis phase and the `istopx` flag can be used to stop the analysis at the end of the current increment. The `spaceset` common block holds the information of the sets of the model. The `iautcr` common block provides the user with the information regarding the increments.

In contrast to the quantities which are available in all subroutines, some common blocks can be used only in specific subroutines which are called for each element of the model during the analysis. Such subroutines are called *element-based* subroutines which contain the calculations to be repeated for each element. A brief list of the element-based subroutines is provided in Table 2.9. The quantities of some common blocks are only available within the element loops of the element-based subroutine (see Table 2.10). Note that in a model with various types of elements, some variables of this table vary depending on the type of element. For instance, the `nnode` variable indicates the number of nodes in an element. This is true for a model with only one type of element but for a model with various types of elements, it only holds the respective data for the current type which is the last group of elements by default. The `SETUP_ELGROUPS` utility subroutine is used to change the current group of elements by setting the proper pointers such as the `icrpt` pointer [26]. The number of groups (`iGroup`) is passed to the subroutine and the number of elements for that group (`nEI`) is retrieved back. To do so, the following line of code can be used:

```
CALL SETUP_ELGROUPS (iGroup, nEI, 0, 0, 0)
```

Using this line of code, the current element group is changed to `iGroup` and as mentioned earlier, some variables of the common blocks will be updated, e.g. the `ielgroup_eln` variable of the `elemdata` common block now points to the list of elements of the group `iGroup`. Additionally, it is possible to set the current element to `intEIID` by the following:

**Table 2.10** Selected element-related data entities accessible via common blocks. Adapted from [26]

| Variable name              | Description   |
|----------------------------|---|
| <b>lass common block</b>   |   |
| n                          | elsto element number  |
| nn                         | Integration point number  |
| kcus(1)                    | User layer number   |
| kcus(2)                    | Internal layer number   |
| <b>elmcom common block</b> |   |
| intel                      | Number of integration points per element (for the current element group)  |
| jintel                     | Number of integration points per element (for the current element). It is equal to intel except when CENTROID option is used (jintel=1)   |
| intsf                      | Number of integration points per element for stiffness matrix evaluation  |
| nnode                      | Number of nodes in element  |
| nnodc                      | Number of corner nodes in element   |
| ibeamel                    | Is set to 1 if the element is a beam  |
| lclass                     | element class:<br>0 : pipe element                      8 : 3D solid<br>1 : truss element                      9 : Fourier element<br>2 : shell                                10 : axi with twist<br>3 : none                                 11 : axisymmetric shell<br>4 : plane stress                        12 : open section beam<br>5 : plane strain                        13 : closed section beam<br>6 : generalized plane strain        14 : membrane<br>7 : axisymmetric solid            15 : gap |
| ityp                       | Internal element type   |
| jtype                      | User element type   |
| ioffsnum                   | Number in list of offset elements   |
| ngenel                     | Number of generated stress component(s) per element   |
| ndi                        | Number of direct components of stress   |
| nshear                     | Number of shear components of stress  |
| nstrm1                     | Size of stress-strain law   |
| nstrm2                     | Number of layers per element  |
| ncrdel                     | Number of coordinates   |
| ndegel                     | Number of degrees of freedom  |
| ndi                        | Number of direct componenets  |
| ngens                      | Number of generalized strains   |
| igroup                     | Group number  |
| <b>nzro1 common block</b>  |   |
| neqst                      | Max. number of layers per element   |
| nstres                     | Max. number of integration points per element   |
| nelstr                     | The amount of memory per element in terms of integer words  |
| <b>far common block</b>    |   |
| m                          | Element number  |

(continued)

**Table 2.10** (continued)

| Variable name         | Description  |
|-----------------------|--|
| nnumel                | Number of elements assembled and stored on sequential file for out of core assembly  |
| elemdata common block |  |
| nelgroups             | Total number of element groups   |
| ielgroup_elnum(*)     | List of internal element numbers stored sequentially (for the current element group)   |
| ielgroup(*)           | List of the group of each element (based on the internal element number)   |
| ieltype(*)            | Element internal type (based on internal ID), generic  |
| ieltype_s(*)          | Element internal type (based on internal ID), structural   |
| ielcon(*)             | Element connectivity (based on the internal element number)  |
| ieltab                | Basic element data array   |
| ieltab(1,i)           | number of nodes in the element   |
| ieltab(2,i)           | internal material ID   |
| ieltab(3,i)           | material orientation type (iangtp)   |
| ieltab(4,i)           | number of layers in the element  |
| ieltab(5,i)           | flag for rebar(2), gasket(3), piezo(4,5), pshell(6), pbush(7) and soil(8)  |
| ieltab(6,i)           | geometry ID of the element (idgeom)  |
| ieltab(7,i)           | property ID of the element, < 0 if user element, Nastran only  |
| ieltab(8,i)           | property type of the element, Nastran only   |
| ieltab(9,i)           | set to 1 if orient2 specifies orientation for all plies  |
| ielgroupinfo          | info on element groups   |
| ielgroupinfo(1,i)     | internal element type  |
| ielgroupinfo(2,i)     | material number/composite group etc.   |
| ielgroupinfo(3,i)     | number of elements in group  |
| ielgroupinfo(4,i)     | nelsto for this element group  |
| ielgroupinfo(5,i)     | number of layers   |
| ielgroupinfo(6,i)     | flag indicating fast integrated composites   |
| ielgroupinfo(7,i)     | number of layers for state variables   |
| ielgroupinfo(8,i)     | material type  |
| ielgroupinfo(9,i)     | flag indicating micro1 solids  |
| ielgroupinfo(10,i)    | flag indicating that this group can be used with fefp for non-automatic option flag indicating that this group can be used with fefp, additive or total (for LARGE STRAIN,4) |
| ielgroupinfo(11,i)    | packed large strain formulation flags for group  |
| ielgroupinfo(12,i)    | packed large strain formulation flags for group  |
| ielgroupinfo(13,i)    | packed material/element allocation flags for group   |
| ielgroupinfo(14,i)    | packed material/element allocation flags for group   |
| ielgroupinfo(15,i)    | packed material/element allocation flags for group   |
| ielgroupinfo(16,i)    | jelsto_a for this element group  |
| ielgroupinfo(17,i)    | Nastran element type for this element group  |
| ielgroupinfo(18,i)    | Number of element subgroups in element group   |

(continued)



**Table 2.10** (continued)

| Variable name       | Description  |
|---------------------|--|
| ielgroup_records    | List of elsto records used by this element group             |
| matdat common block |  |
| et(3)               | Young's moduli   |
| xu(3)               | Poisson's ratios   |
| rho                 | Mass density   |
| shrm(3)             | Shear moduli   |
| coed(3)             | Coefficient of thermal expansion                             |
| yield(1)            | Yield stress   |
| prepro common block |  |
| nNoIDs              | Flag if non-zero indicates non-consecutive node numbering    |
| iNoIDs_d            | Contains the external ID of the nodes if nNoIDs is flagged   |
| nEIDs               | Flag if non-zero indicates non-consecutive element numbering |
| iEIDs_d             | Contains the external ID of the elements if nEIDs is flagged |
| maxEIID             | Maximum user ID of the elements                              |
| maxCrdIID           | Maximum user ID of the coordinate system                     |
| maxCmplID           | Maximum ID of composites                                     |
| iQuit               | Flag indicating if the QUIT subroutine is executed           |

```
1 CALL SetEI (intEIID)
```

This line of code updates the number of integration points (jintel) to that of the current internal element (intEIID). This is a straightforward way of obtaining the number of integration points for a specific element.

## 2.4 Debugging

When using any programming language, avoiding possible errors is almost inevitable. Therefore, usually a built-in debugger is provided with the programming package. But unfortunately, when it comes to using the user subroutines in conjunction with MARC, things get more complicated because there are not many direct tools provided for debugging the code and tracing the steps within the program. Moreover, a critical error during the execution of a FORTRAN subroutine leads to the crashing of the program or getting stuck in an infinite loop and it would be rather hard to find out which part of the code is responsible for such cases. The general strategy is using procedural/modular programming which makes it possible to verify these independent pieces of code in a proper debugger outside of MARC (Sect. 1.5). Although, this will ascertain that the modular part is almost innocent, sometimes the erroneous code is not placed within the modular part. With no tools in hand, coping with problems would be laborious.

An indirect approach to find the culprit can be putting a simple output statement in specific places into the code which are more susceptible to errors, e.g. the complex part of an algorithm or newly-added lines of code. By this method, the last successful output indicates that most probably up to that output statement everything works well; the bad line of code is most probably located somewhere after the output statement. By using more output statements, it would be possible to narrow down the code and exactly locate the line.

The same approach can be taken in order to get an insight of the variables of the subroutines and mimic the *watch* tool of a debugger. In other words, the output statements can be used to print out the value of the variables to help detecting the error.

Note that the output to the screen using the default output which is indicated by an asterisk, i.e. a `WRITE (*,*)` or a `PRINT` statement, will not appear on the screen unless the job is run from the command prompt (see Sect. 2.4.5). If the job is submitted using `MENTAT`, the output file can be used as a debugging medium, namely the outputs can be directed to the output file instead of the screen. Since the default file unit for the output file (.out) is 6, the `WRITE (*,*)` statement should be replaced by `WRITE (6,*)`. The result is that the debugging output appears in the output file instead of the screen.

In the following subsections, some of the common mistakes in FORTRAN programming will be discussed briefly. The best remedy to these pitfalls is to avoid them. However, a few methods are introduced to tweak `MARC/MENTAT` in order to obtain extra debugging information of the program and some tools are introduced to increase the control on running jobs.

### 2.4.1 Common Pitfalls

There are a few common mistakes in programming which can add an extra twist to the debugging procedure. The best approach is to get familiar with the following cases and avoid them to ease the debugging process:

1. It is a good practice to initialize the variables to be used and make them defined prior to use. An undefined variable contains random data, often called *garbage*, which can make the detection of the error source troublesome. It is possible to initialize a local variable upon its declaration either using the `DATA` attribute or by assigning an initial value to it. However, in the former method, the initialization of a variable is accompanied by an implicit `SAVE` attribute. Consequently, only in the first run the initial value is assigned to the variable. Consider the following example:

```

1      SUBROUTINE Sample ()
2          INTEGER :: zero = 0
3
4          WRITE (*,*) 'In the first run the value is ', zero
5          zero = 1
6      END SUBROUTINE Sample

```

In the first run of this code, the value 0 is assigned to the integer zero but in the subsequent calls the last assigned value will be retained. Because of the local initialization, in reality the following code will be compiled:

```

1      SUBROUTINE Sample ()
2          INTEGER, SAVE :: zero = 0
3
4          WRITE (*,*) 'In the first run the value is ', zero
5          zero = 1
6      END SUBROUTINE Sample

```

Concisely, this property is a double-edged sword. It can be used to determine the first run of the subprogram. But if the goal is just initialization of the variable, some problems may arise. In any case, a proper initialization can be done with the following code:

```

1      SUBROUTINE Sample ()
2          INTEGER :: zero
3
4          zero = 0
5
6          WRITE (*,*) 'In the first run the value is ', zero
7          zero = 1
8      END SUBROUTINE Sample

```

2. An *access violation* occurs when attempting to access a restricted memory address, namely the address is not available or the user is not authorized to use it. Usually the following message is returned by the FORTRAN compiler for this error:

fortrl: severe (157): Program Exception - access violation

Various reasons may cause such an error among which are the followings:

- a. Accessing elements of an array which are not in the defined range. For instance, accessing myList(3,5) of the array REAL :: myList(3,3) is not possible. Note that extra care should be devoted to the assumed size arrays since the programmer is responsible for the number of elements in such arrays. To check for the bounds, the following switch can be used with the FORTRAN compiler:

/check:bounds

- b. The argument mismatch is another source of error, which can be a subtle one sometimes. Because not only the number and type of arguments must match but also the KIND and LEN parameter must be selected accordingly. Consider the following subroutine header:

```

1      SUBROUTINE Demo(a, b)
2          INTEGER, INTENT(IN) :: a
3          REAL*8, INTENT(OUT) :: b
4          ...
5      END SUBROUTINE Demo

```

A call to this subroutine with the following variables is invalid:

```

1      INTEGER :: anInt
2      REAL :: aReal
3
4      CALL demo(anInt, aReal)

```

Although the compiler could recognize such mismatches, this is not the case for the external subprograms which are in compiled libraries. Therefore, make sure that the number and the type of arguments are always in accordance to the header of the subprogram.

- c. Assigning a value to a constant, either a numerical or a named one, causes a crash. This case usually happens when the constant is passed as an argument to a subprogram in which a value is assigned to the constant. A constant parameter is assumed to be read-only. To allow writing on the constants the following switch option can be used with the FORTRAN compiler:

```
/assume:noprotect_constants
```

- d. Referencing uninitialized variables, unallocated arrays or pointers may cause such a problem as well.

3. Avoid implicit type declarations to make typographical errors easy to discover.
4. Try to avoid using common blocks as much as possible and instead use modules to share data. However, when using common blocks is inevitable, make sure that the ordering of the variable is preserved.
5. Comparing two real numbers using `.EQ.` and `.NE.` may not produce accurate results because there are always approximations when dealing with floating point numbers. It is a good idea to compare the difference of two real variables with an acceptable tolerance. Consider the following code:

```
1 REAL :: aReal, bReal
2
3 IF (aReal .EQ. bReal) THEN
4     ...
```

It is better to convert it to the following:

```
1 REAL :: aReal, bReal, maxErr
2
3 IF (ABS (aReal - bReal) .LT. maxErr) THEN
4     ...
```

Note that the `maxErr` variable is small enough depending on the problem in-hand but also larger than the machine epsilon.

6. When it comes to real-integer numerical errors, floating point approximations can be amplified. Such a case arises with intrinsic functions such as `MOD` to calculate the remainder of the division of two real numbers. Consider the following example:

```
1 REAL :: aReal, bReal, rem
2
3 aReal = 2.0
4 bReal = 0.2
5
6 rem = MOD (aReal / bReal)
7 Print *, rem
```

Although the result is zero as the remainder, the output will show the number 0.2. The `MOD` function actually calculates the following expression:

```
1 MOD = aReal - (bReal * INT(aReal / bReal))
```

During the floating point conversion of  $bReal = 0.2$ , a round-up occurs and thus,  $aReal/bReal$  will be equal to a number slightly less than 10, e.g. 9.999. Next, the INT function will extract the integral part of the number, i.e. 9. Finally, the calculation results in 0.2 as the answer which is incorrect. If instead of a real number, double precision real numbers are used, the result will be a very small number which can be detected as zero with a proper IF statement.

### 2.4.2 Requesting Additional Information

During the finite element analysis, MARC hides the extra information from the user, i.e. not all details of the finite element analysis are provided. Sometimes it is useful to take a look at the information in these hidden steps to obtain an improved judgment on the results. It is possible to ask for the hidden information using either MENTAT, subroutines or input file options. By default, the requested information will appear in the output file.

Within MENTAT, the required details can be selected in the output dialog. This dialog can be reached by the following command:

③ Jobs > Jobs > Properties > Job Results > Output File

It is also possible to request for additional information within a subroutine by setting the ideva flag from the common block concom. This flag is actually an array of flags, each one regarding a specific piece of information as listed in Table 2.11. For instance, it is possible to request extra information on the model via the ideva variable by setting the first flag on, i.e.  $ideva(1) = 1$ . This will print the information regarding the stiffness matrices of the elements to the output file.

Another way is to use some input file options such as PRINT, PRINT NODE, PRINT ELEMENT, NODE SORT, ELEM SORT, PRINT CHOICE, PRINT VMASS and others. Note that the PRINT option uses the same mechanism as the ideva flag which leads to a debug printout and the NODE SORT and ELEM SORT options produce sorted outputs for nodes and elements, respectively. A summary of specific output information can be obtained using the SUMMARY option in conjunction with the PRINT NODSTS option.

There are four subroutines which make various quantities available during the analysis. These quantities can be used for debugging purposes or to pass them to other subroutines for controlling purposes. The list is as follows:

- the IMPD subroutine makes some nodal quantities available at the end of each increment,
- the ELEVAR subroutine makes the elemental quantities in the integration points available at the end of each increment,
- the ELEVEC subroutine makes the elemental quantities in the integration points available at the end of each harmonic sub-increment, and

**Table 2.11** Print-out details of the IDEVA debugging flag. Adapted from [24]

| Variable name  | Description  |
|--|--|
| ideva(60)  | Flag for debugging print-outs  |
|  | (01) Print element stiffness matrices, mass matrix   |
|  | (02) Output matrices used in tying   |
|  | (03) Force the solution of a non-positive definite matrix  |
|  | (04) Print info of connections to each node  |
|  | (05) Info of gap convergence, internal heat, contact touching/separation   |
|  | (06) Nodal value array during rezoning   |
|  | (07) Tying info in CONRAD GAP option, fluid element numbers in CHANNEL option  |
|  | (08) Output incremental displacements in local coordinate system   |
|  | (09) Latent heat output  |
|  | (10) Stress-strain in local coordinate system  |
|  | (11) Additional info on inter-laminar stress   |
|  | (12) Output right hand side and solution vector  |
|  | (13) Info of CPU resources used and memory available on NT   |
|  | (14) Info of mesh adaptation process, 2D outline information info of penetration checking for remeshing save .fem files after afmesh3d meshing |
|  | (15) Surface energy balance flag   |
|  | (16) Print information regarding pyrolysis   |
|  | (17) Print information on streamline topology  |
|  | (18) Print mesh data changes after remeshing   |
|  | (19) Print material flow stress data read in from *.mat file if unit flag is on, print out flow stress after conversion                        |
|  | (20) Print information on table input  |
|  | (21) Print out information regarding kinematic boundary conditions   |
|  | (22) Print out information regarding dist loads, point loads, film and foundations   |
|  | (23) Print out information about automatic domain decomposition  |
|  | (24) Print out iteration information in SuperForm status report file   |
|  | (25) Print out information for ablation  |
|  | (26) Print out information for films - table input   |
|  | (27) Print out the tying forces  |
|  | (28) Print out for CASI solver and convection  |
|  | (29) DDM single file debug printout  |
|  | (30) Print out cavity debug information  |
|  | (31) Print out welding related information   |
|  | (32) Prints categorized DDM memory usage   |
|  | (33) Print out the cutting info regarding machining feature  |
|  | (34) Print out the list of quantities which can be defined via a table and for each quantity the supported independent variables               |
|  | (35) Print out detailed coupling region info   |
|  | (36) Print out solver debug info level 1 (Least Detailed)  |
| (37) Print out solver debug info level 1 (Medium Detailed) |  |

(continued)

**Table 2.11** (continued)

| Variable name | Description   |
|---------------|---|
|               | (38) print out solver debug info level 1 (Very Detailed)  |
|               | (39) print detailed memory allocation information   |
|               | (40) print out MARC-ADAMS debug information   |
|               | (41) output rezone mapping post file for debugging  |
|               | (42) output post file after calling oprofos() for debugging   |
|               | (43) debug printout for VCCT  |
|               | (44) debug printout for progressive failure   |
|               | (45) print out automatically generated midside node coordinates (arecrd)  |
|               | (46) print out message about routine and location, where the ibort is raised (ibort_inc)  |
|               | (47) print out summary message of element variables on a group-basis after all the automatic changes have been made (em_ellibp)   |
|               | (48) Automatically generate check results based on maximum and minimum vals. These vals are stored in the checkr file, which is inserted into the *.dat file by the generate_check_results script from /marc/tools  |
|               | (49) Automatically generate check results based on the real calculated values at the specified check result locations. These vals are stored in the checkr file, which is inserted into the *.dat file by the update_check_results script from tools subdirectory |
|               | (50) generate a file containing the resistance or capacity matrix; this file can be used to compare results with a reference file   |
|               | (51) print out detailed information for segment-to-segment contact  |
|               | (52) print out detailed relative displacement information for uniaxial sliding contact  |
|               | (53) print out detailed sliding direction information for uniaxial sliding contact  |
|               | (54) print out detailed information for edges attached to a curve   |
|               | (55) print information related to viscoelasticity calculations  |
|               | (56) print out detailed information for element coloring for multi-threading  |
|               | (57) print out extra overheads due to multi-threading. These overheads include time and memory. The memory report will be summed over all the children  |
|               | (58) debug output for ELSTO usage   |

- the INTCRD subroutine makes the integration point coordinates available at the end of each increment.

The activation of the first three subroutines can be done using the UDUMP input file option and the last is done automatically. There are also a couple of utility subroutines which can aid the user to obtain additional quantities; NODVAR and ELMVAR to extract the analysis results of elements and nodes, respectively. These two utility subroutines can be called from other subroutines but depending on the stage of the analysis, the returned values may differ. For example, if NODVAR is called from the IMPD subroutine then the acquired values will be for the end of the

current increment. However, if the call is made at other points of the analysis, the results may refer to the end of the previous increment.

### 2.4.3 Activating the Debugging Mode

One of the powerful FORTRAN compilers is the INTEL® FORTRAN which is available for various platforms and is the only compatible compiler for MARC. After introducing the appropriate file containing all the subroutines, say a file named `all_sub.f`, MARC sends this file automatically to the INTEL® FORTRAN compiler, namely `ifort`, with a command such as the following to create the *object file*:

```
1 ifort /fpp /c /DWIN32\_intel -D\_IMPLICITNONE ...
```

The next step will be using the LINK command to link the object file to other library files of MARC and/or user-defined libraries which finally leads to an executable file by means of the following command:

```
1 LINK /nologo /out:"all\_sub.exe" ...
```

Such command prompt switches are used by MARC to compile the source file with the appropriate options and link the object file to the appropriate library files. These switches are set by the `include_win64.bat` batch file. This important batch file contains the definition of the variables required during compilation and it runs upon submitting the job; either from MENTAT or from the command prompt via the `run_marc.bat` batch files.

Similar to other batch files of the command prompt, many environmental variables are defined and used to cover various circumstances. For instance, it is possible to activate the debugging mode for the compiler by setting the `MARCDEBUG` variable to ON or even add stack checking for function calls by setting the `MARCCHECK` variable to ON.

Activating the debugging mode is useful because it forces the compiler to create the *program database file* with the `.pdb` extension. An additional benefit is providing the number of the responsible line(s) in the case of a crash. This temporary environmental variable actually activates the `/traceback` option of the compiler. To activate the debugging mode, modify the following lines of the `include_win64.bat` batch file by means of a text editor:

```
1 ...
2 REM Uncomment the following lines to build MARC in Debuggable mode
3 set MARCDEBUG=OFF
4 REM set MARCDEBUG=ON
5 if "%MARCDEBUG%"=="ON" goto setdebug
6 ...
```

The REM command indicates the remarks of the batch file. To activate the debugging mode, remove the REM from the fourth line and put it in front of the third line. Activating extra checks for debugging purposes is possible by modifying the following line:



```

1 REM Uncomment following lines to build MARC with more run-time checks
2 set MARCCHECK=OFF
3 REM set MARCCHECK=ON
4 if "%MARCCHK%"=="OFF" goto endcheck

```

And setting the MARCCHECK variable to ON:

```

1 REM Uncomment following lines to build MARC with more run-time checks
2 REM set MARCCHECK=OFF
3 set MARCCHECK=ON
4 if "%MARCCHK%"=="OFF" goto endcheck

```

It is also possible to add other switches, for instance:

```

1 SET DEBUG_OPT=%DEBUG_OPT% /debug-parameters:all

```

In addition, there are other variables in this batch file to force more control among which are the following ones:

- the MAXNUM variable which is used to set the maximum number of node/elements of the model with the default value of one million,
- the MEMLIMIT variable which is used to limit the amount of memory used by the matrix solver with the default value equal to the amount of the available physical memory,
- the LIBDIR variable contains the paths for the required libraries for compilation, and
- the MPITYPE variable is used to select a MPI which is used in parallel computing.

## 2.4.4 Compiler Directives

Another capability of the compiler is using the FORTRAN compiler preprocessor which is activated using the /fpp switch. There are times prior to compiling the FORTRAN source file in which some manipulations are required to match the compiler with the circumstances or in a more technical term some *preprocessing* might be needed for conditional compiling. The commands used to direct the compiler in order to carry out the preprocessing are called *compiler directives*. The typical job of a compiler directive is usually one of the following:

- conditional compilation, e.g. #if, #ifdef, #elif, #else and #endif,
- file inclusion, e.g. #include, or
- preprocessing variable or *macro* substitution, e.g. #define.

The syntax of the compiler directives are based on the C/C++ programming language preprocessor and for this fact the directives must start from the first column of each line; they are not FORTRAN statements and the fixed format rules will not apply to them.

One useful directive which is used for conditional compilation is the #ifdef directive: it checks whether a flag is defined either by the #define directive inside the FORTRAN source file or by the -D option upon using the compiler. This facility is

used by MARC and the flag `_IMPLICITNONE` is used upon calling the compiler to impose the necessity of defining every variable explicitly by the following code:

```

1 #ifdef _IMPLICITNONE
2     IMPLICIT none
3 #else
4     IMPLICIT LOGICAL (a-z)
5 #endif

```

This piece of code is used at the beginning of every subroutine template which is located in the user subdirectory of MARC and is a good replacement for the following line:

```

1     IMPLICIT REAL*8 (a-h, o-z)

```

This implicit type declaration is used in the templates listed in the documentation of MARC and it has the same problem as any other implicit type declaration, i.e. it adds more complications to the debugging process. Therefore, this approach is not recommended. It is also worth reminding that every real type declaration in the subroutines must be of `KIND = 8` to ensure the correct transfer of data between MARC and the subroutine.

The `#include` directive is used to include another file in the current source file. For instance, a file named `mymodule.f`, residing in the same directory as the source file, will be included by adding the following line in the code:

```

1 #include "mymodule.f"

```

A macro is a series of commands to which a name is assigned. A macro is also called a *symbol* which is usually a value. The compiler replaces every macro/symbol with the corresponding value during preprocessing. There are some predefined preprocessor symbols such as `_WIN64` which is equal to 1 in WINDOWS® operating systems with a 64 bit architecture. For more information on macros refer to [17].

Another use of the compiler directives is to manage the statements which are related to the debugging of the code. As mentioned earlier, it is not possible to use a debugger while developing the user subroutines for MARC. Therefore, output statements such as `WRITE` are used to print critical variables of the code for debugging purposes. After finishing the debugging of the code, these extra statements, which are not really a part of the original code, are usually turned into comments or removed from the code. In the case of another crash of the code, those lines must be added again or turned back into effective code lines.

There is an elegant way of avoiding this repetitive process which appears to be useful in the case of erroneous code. It is done by using the conditional compilation capability of the FORTRAN compiler, i.e. the `#define`, `#ifdef` and `#endif` compiler directive. A variable/macro is defined using the `#define` directive to indicate that debugging is required for the code. This condition is checked with the other two mentioned directives. If the condition is satisfied the code responsible for printing the debugging information is compiled. Otherwise, the compiler will neglect the code.

It is worth mentioning that to distinguish the name of this variable/macro from the normal FORTRAN programming code, a different naming convention is used. Usually uppercase letters are used with an underscore at the beginning, e.g. `_IMPLICITNON`.

Let us consider the following lines of code:

```

1      DO i = 1, n
2      elmStress (i) = [(IPStress(j), j = 1, m)] / m
3      END DO

```

The program crashes during execution and the compiler points to the second line as the faulty line. It is necessary to investigate the values of the variables during the execution of the loop. Therefore, the code is modified to the following:

```

1      DO i = 1, n
2      WRITE (6,*) 'i = ', i
3      WRITE (6,*) 'IPStressSum(i) =', IPStressSum
4      WRITE (6,*) 'm =', m
5      WRITE (6,*) 'IPStress(1:m) =', [(IPStress(j), j = 1, m)]
6
7      IPStressSum(i) = IPStressSum(i) + [(IPStress(j), j = 1, m)] / m
8      END DO

```

Lines 2 to 5 of this code are dedicated to watching the variables for debugging. After resolving the issue, one may either transform these lines to comments or delete them. But using the compiler directives, one could control this without changing all the code. Using the directives, the previous code is update to the following:

```

1  #define _DEBUGON
2      DO i = 1, n
3
4  #ifdef _DEBUGON
5      WRITE (6,*) 'i = ', i
6      WRITE (6,*) 'IPStressSum(i) =', IPStressSum
7      WRITE (6,*) 'm =', m
8      WRITE (6,*) 'IPStress(1:m) =', [(IPStress(j), j = 1, m)]
9  #endif
10
11     IPStressSum(i) = IPStressSum(i) + [(IPStress(j), j = 1, m)] / m
12     END DO

```

In this listing, a variable named `_DEBUGON` is declared without assigning any values to it (line 1). The value could be a number or an expression but none of these is required in our case. In line 4, the existence of this variable is checked. If it exists, the debugging lines will be compiled. Otherwise, they will be ignored. To turn off the debugging lines, the first line should be simply transformed into a comment line.

The benefit of this method is in controlling all the debugging lines of the code with one single line. In addition, it is possible to define multiple levels of debugging in terms of description, e.g. a very detailed debugging output or a very compact one. This can be done using nested compiler conditions using the `#if`, `#elif` and `#endif` directives. The same code can be modified for a two-level debugging output:

```

1  #define _DEBUG_LEVEL2
2
3      DO i = 1, n
4  #ifdef _DEBUG_LEVEL1
5      WRITE (6,*) 'i = ', i
6      WRITE (6,*) 'IPStressSum(i) =', IPStressSum
7      WRITE (6,*) 'm =', m
8      WRITE (6,*) 'IPStress(1:m) =', [(IPStress(j), j = 1, m)]
9  #elif _DEBUG_LEVEL2
10     WRITE (6,*) 'i = ', i
11     WRITE (6,*) 'IPStressSum(i) =', IPStressSum
12 #endif

```

```

13 IPStressSum(i) = IPStressSum(i) + [(IPStress(j), j = 1, m)] / m
14 END DO

```

In this listing, the compact debugging output will be printed. This debugging level is indicated by defining the `_DEBUG_LEVEL2` variable.

One could go even further and define some macros for the debugging statements. For instance, the `WRITE` statement can be defined as a macro. Using macros, the previous code can be updated to the following:

```

15 #define _W WRITE(6,*)
16 #define _DEBUG_LEVEL2
17
18 DO i = 1, n
19 #ifdef _DEBUG_LEVEL1
20   _W 'i = ', i
21   _W 'IPStressSum(i) =', IPStressSum
22   _W 'm =', m
23   _W 'IPStress(1:m) =', [(IPStress(j), j = 1, m)]
24 #elif _DEBUG_LEVEL2
25   _W 'i = ', i
26   _W 'IPStressSum(i) =', IPStressSum
27 #endif
28 IPStressSum(i) = IPStressSum(i) + [(IPStress(j), j = 1, m)] / m
29 END DO

```

In the first line, a macro is defined for the output statement and the corresponding debugging statements are replaced by the macro (lines 6–9, 11 and 12). This macro enables the user to change the output of all the debugging statements from 6 to any other file units by changing a single line. In addition, the debugging code looks more concise and distinguishable from the normal lines of code.

Macros are quite powerful and they can even be defined to accept parameters and it is possible to use them to replace normal programming lines. They obscure the real code, which is alright for the debugging code, but it reduces the clarity of the normal programming code. Since they change the normal look of the code to a custom one, it is not advised to use them extensively.

### 2.4.5 Controlling the Job Submission

The analysis of a job is usually submitted from `MENTAT` and it is done via one of the `submit1.bat`, `submit2.bat` or `submit3.bat` batch files. Either one of these files will configure the options for the job submission and finally runs the `run_marc.bat` batch file. This batch file is responsible for the running of the job and is located in the `tools` subdirectory of `MARC`. However, it is also possible to submit a job from the command prompt directly to the `run_marc.bat` batch file. The direct submission of a job has the following advantages:

- providing more flexibility over controlling job submission via option switches, and
- releasing the memory occupied by `MENTAT`. Therefore, a larger amount of memory will be available to `MARC` for the analysis.

**Table 2.12** Switches for direct running of Marc. Adapted from [25]

| Switch syntax                         | Description   |
|---------------------------------------|---|
| -alloc                                | Forces MARC to perform memory allocation test without carrying out the analysis   |
| -autorst yes/no or<br>-au y/n         | With the yes switch, enables the auto-restart feature in which the analysis is stopped, a mesher/remesher runs and then the analysis restarts. The default value is no for which the analysis runs normally |
| -back yes/no or -b<br>y/n             | With the yes switch, MARC runs in the background. The default value is no for which the analysis runs normally  |
| -bg yes/no                            | With the yes switch, all the messages will be printed to the screen, i.e. the log file and the output file. In addition, the output file will be created without the log file                               |
| -def or -de<br>default-filename       | Specifies the name of the default-file  |
| -dir working-path                     | Specifies the current working directory to run the job. The default value is the current path   |
| -jid or -j input<br>filename.dat      | Specifies the name of the input file name. The .dat file-extension is optional  |
| -list yes or -list y                  | Lists the options used in the input file in the output file. No analysis is done  |
| -ml memory-limit                      | Specifies the memory limit in Megabytes. This will override the default values specified in the file run_marc_defaults  |
| -mo i4/i8                             | Specifies one of the following kinds for the integers: i4 and i8 which is available only on 64-bit systems. This will override the default values specified in the file run_marc_defaults                   |
| -mpi mpi-type                         | Specifies one of the following types for the MPI: intempi, msmpi, hpmpi or hardware   |
| -nprocs or -nps<br>domain-count       | Specifies the number of domains for parallel processing using a single input files  |
| -nprocd or -np<br>domain-count        | Specifies the number of domains for parallel processing   |
| -nsolver<br>task-count                | Specifies the number of tasks for solver number 12. These distributed tasks operate via MPI   |
| -nthread_elem<br>thread-count         | Specifies the number of threads for element assembly and recovery. A shared-memory thread which does not use MPI  |
| -nthread or -nts<br>thread-count      | Specifies the number of threads for the parallel matrix solvers (solver number 8, 9 and 10)   |
| -obj object-<br>filename.obj          | Specifies the user object-file/libraries (if any) used in the subroutines. Note that the .obj file-extension must be used   |
| -ou 1                                 | Forces MARC to do the out-of-core storage of the element data   |
| -pid or -pi post-<br>filename.t16/t19 | Specifies the previously-created post-file  |
| -prog or -pr<br>compiled-filename     | Specifies the compiled executable ready to be used by MARC. Note that the .exe file-extension must not be used  |
| -rid or -r restart-<br>filename.t08   | Specifies the previously-created restart-file   |
| -save yes/no or<br>-sav y/n           | yes: The compiled user subroutine file will be saved. The default value is no for which the compiled file will be deleted   |

(continued)

**Table 2.12** (continued)

| Switch syntax              | Description  |
|----------------------------|--|
| -sdir scratch-path         | Specifies the path for scratch files during the analysis. The default value is set to the working path |
| -sid substructure-filename | Specifies the substructure-file  |
| -user or -u user-sub.s.f   | Specifies the name of the user subroutine file. The .f file-extension is optional                      |

In Table 2.12, a list of switches for run\_marc.bat is provided. Note that most of the time they come in pairs, e.g. the *command-line argument* -jid is paired with its *value* input filename.dat. These switches can be used to control the behavior of MARC. For instance, the following command will run an input file named truss01.dat and compiles the corresponding subroutine source file mysub.f:

```
1 run_marc -jid truss01.dat -user mysub.f
```

It is possible to save the compiled subroutine as an executable with the following line:

```
1 run_marc -jid truss01.dat -user mysub.f -save yes
```

Now, it is possible to run the saved executable rather than compiling the FORTRAN code at each run:

```
1 run_marc -jid truss01.dat -prog mysub
```

As another example, an object file mylib.obj can be linked to the main FORTRAN file with the following line:

```
1 run_marc -jid truss01.dat -user mysub.f -obj mylib.obj
```

It is also possible to save the compiled file of such a link to the external object file by the following:

```
1 run_marc -jid truss01.dat -user mysub.f -obj mylib.obj -save yes
```

And run the executable file in the next run by the following line:

```
1 run_marc -jid truss01.dat -user mysub.f -prog mysub
```

It is also possible to use these options from MENTAT provided that the proper modifications are done to the files submit1.bat and mentat.bat. For instance, passing a file name to -obj switch as an object file can be done by means of environmental variables, e.g. an environmental variable named my\_object is created containing the full path to the object file. Next, some modifications will be done to the file submit1.bat to check if such a variable exists or not and to act accordingly. The following lines of the file submit1.bat are responsible of calling run\_marc.bat to run a job:

```
1 :do_run
2
3 if exist %job%.cnt del %job%.cnt
4
```

```

5 call "%MARCKDIR%\tools\run_marc" %slv% -j %job% %nprocs% %nproc% -autorst %autorst%
   %srcfile% %restart% %postfile% %viewfact% %hostfile% %compat% %copy_datfile%
   %copy_postfile% %dcom_host% %scr_dir% %decoup% %assem_recov_nthread% %nthread% %nsolver%
   %mode% %gpu% -b y
6
7 goto done
8 :error
9 echo ERROR... jobname is required
10 :done

```

These are the last lines of the submit1.bat batch file and can be found easily by searching for the :do\_run label. In this listing, line 6 is responsible of running MARC without the -obj switch. This listing should be modified to the following listing:

```

1 :do_run
2
3 if exist %job%.cnt del %job%.cnt
4
5 IF "%my_object%"==" " GOTO no_object_run
6
7 :object_run
8 CALL "%MARCKDIR%\tools\run_marc" %slv% -j %job% %nprocs% %nproc% -autorst
   %autorst% %srcfile% %restart% %postfile% %viewfact% %hostfile% %compat%
   %copy_datfile% %copy_postfile% %dcom_host% %scr_dir% %decoup%
   %assem_recov_nthread% %nthread% %nsolver% %mode% %gpu% -b y -obj
   %my_object%
9 GOTO done
10
11 :no_object_run
12 call "%MARCKDIR%\tools\run_marc" %slv% -j %job% %nprocs% %nproc% -autorst
   %autorst% %srcfile% %restart% %postfile% %viewfact% %hostfile% %compat%
   %copy_datfile% %copy_postfile% %dcom_host% %scr_dir% %decoup%
   %assem_recov_nthread% %nthread% %nsolver% %mode% %gpu% -b y
14
15 goto done
16 :error
17 echo ERROR... jobname is required
18 :done

```

In this modified batch file, line 5 checks if an environmental variables named my\_object exists; if yes, it jumps to the label object\_run and calls the run\_marc.bat batch file with the switch -obj %my\_object%. Otherwise, it jumps to the label no\_object\_run and runs the same batch file without the -obj switch.

Alternatively, to carry out the same task, a simple modification can be done in the following lines of the run\_marc.bat file:

```

1 ...
2 set userdir=
3 set objs=
4 set qid=y
5 ...

```

And add the environmental variable %my\_object% as the object of the job submission. The same lines will finally look as the following:

```

1 ...
2 set userdir=
3 set objs=%my_object%
4 ...

```

### 2.4.6 Using the Visual Studio IDE

MICROSOFT® VISUAL STUDIO (VS)<sup>4</sup> is a developer tool for computer programming, which provides the user with a powerful IDE.<sup>5</sup> The IDE offers an environment equipped with various tools for code editing, compiling and debugging. Normally, all of these features can be used for typical FORTRAN programming code. However, regarding MARC subroutines, the IDE is merely used as a text editor and other processes, i.e. compiling and debugging, are done using the methods presented in the previous subsections.

It is possible to configure the MICROSOFT® VISUAL STUDIO IDE and take advantage of its capabilities to the fullest extent. The correct settings are made available in this subsection. They are tested for MARC/MENTAT 2014.2 equipped with INTEL® FORTRAN 2013 XE-Update 5 and MICROSOFT® VISUAL STUDIO 2012 under a 64bit WINDOWS® operating system. A similar configuration can be applied to other cases.

It is assumed that MARC/MENTAT is installed in its default directory C:/MSC.Software/ and that the working directory is C:/Mentat\_Dir/. The working directory should contain the model input file and the source codes for the subroutines.

In order to compile a dependent MARC subroutine in MICROSOFT® VISUAL STUDIO, these steps must be followed:

1. A new empty project must be created as an INTEL® Visual FORTRAN console application:  
File > New > Project
2. The source file of the subroutine must be added to this project:  
Project > Add Existing Item...
3. It is required to set the debugging options of the project. For a project, e.g. Console1, one must execute the following:  
Debug > Console1 Properties...

In the appearing dialog box, the following configurations must be set for the compiler:

- For 64-bit systems the active platform must be changed by executing the following:  
Configuration Manager... > Platform: x64
- Suppress the startup banner:  
Fortran > General > Suppress Startup Banner: Yes (/nologo)

---

<sup>4</sup><http://www.visualstudio.com/>.

<sup>5</sup>Integrated development environment.



- Include your working directory, e.g. c:/mentat\_dir, along with the required directories of MARC as follows:

Fortran ▸ General ▸ Additional Include Directories: <Edit...>

```
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/common"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/bcsgpusolver/common"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/mumpsolver/include"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmpi/win64/include"
"C:/mentat_dir"
```

- Activate the generation of full debugging information:

Fortran ▸ General ▸ Debug Information Format ▸ Full (/debug:full)

- Disable optimization:

Fortran ▸ Optimization ▸ Optimization: Disable (/Od)

- Activate the interprocedural optimization:

Fortran ▸ Optimization ▸ Interprocedural optimization: Single-file (/Qip)

- Enable preprocessing of the source file:

Fortran ▸ Preprocessor ▸ Preprocess source file: Yes(/fpp)

- Use the following preprocessor definitions:

Fortran ▸ Preprocessor ▸ Preprocess Definitions: <Edit...>

```
WIN32_intel
_IMPLICITNONE
I64
MKL
OPENMP
OMP_COMPAT
_MSCMARC
WIN64
CASI
PARDISO
MUMPS
BCSGPU
CUDA
DDM
```

- Activate the generation of parallel codes:

Fortran ▸ Language ▸ Process OpenMP Directives: Generate Parallel Code (/Qopenmps)

- The following property must be changed only if the 8-byte integer mode is used:

Fortran ▸ Data ▸ Default Integer KIND: 8 (integer\_size:64)

- Deactivate sharing memory between CRAY pointers and other variables:

Fortran ▸ Data ▸ Assume CRAY Pointers Do Not Share Memory Locations: Yes (/Qsafe\_cray\_ptr)

- Activate the generation of automatic codes after function calls to check the stack:

Fortran ▸ Floating-Point Stack ▸ Yes (/Qfp-stack-check)

- Set the output directory, e.g. c:/mentat\_dir, for the compiled module file(s), compiled object file(s), and information file(s):

Fortran ▸ Output Files ▸ Module path: c:/mentat\_dir/

Fortran ▸ Output Files ▸ Object File Name: c:/mentat\_dir/

Fortran ▸ Output Files ▸ Profile Directory: c:/mentat\_dir/

- Specify the output directory, e.g. c:/mentat\_dir, for the program database file:

Fortran ▸ Output Files ▸ Program Database File Name: c:/mentat\_dir/vc110.pdb

- Activate the generation of traceback information for runtime errors:

Fortran ▸ Run-time ▸ Generate Traceback Information: Yes (/traceback)

- Activate full runtime error checking:

Fortran ▸ Run-time ▸ Runtime Error Checking: All (/check:all)

- Specify the runtime library for linking:

Fortran ▸ Libraries ▸ Runtime Library: Debug Multithread (/Libs:static /threads /dbglibs)

- Add the following miscellaneous switches as additional options:

Fortran ▸ Command Line ▸ Additional Options:

/c /Qvec- /switch:fe\_old\_modvar /W0 /Zi /fpe:0 /Qopenmp /Qopenmp-threadprivate: compat /MD

The following configurations must be applied to the linker:

- The name of the output executable file, e.g. subs.exe, must be added along with its full path, e.g. c:/mentat\_dir/. This file is the final compiled one which will be used by MARC:

Linker ▸ General ▸ c:/mentat\_dir/subs.exe

- Disable incremental linking:

Linker ▸ General ▸ Enable Incremental Linking: No (/INCREMENTAL:NO)

- Suppress the startup banner:

Linker ▸ General ▸ Suppress Startup Banner: Yes

- Add the additional library directory:

Linker ▸ General ▸ Additional Include Directories: <Edit...>

"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmpi/win64/lib"

- Enter the additional object and library files for linking:

Linker > Input > Additional Dependencies:

```
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/main.obj"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/blkdta.obj"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/comm1.obj"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/comm2.obj"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/comm3.obj"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/srclib.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/mcvfit.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/mnflib.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/md_user.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/mdsrc.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/bcsgplib.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/marccuda.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_intel_ilp64.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_intel_thread
.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_core.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/libiomp5md.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/blas_src.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/casilib.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_solver_ilp
64.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/ACSI_Marc.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/mumps.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_scalapack_ilp
64.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_lapack95_ilp
64.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/intelmkl/win64i8/mkl_blacs
_intelmpi_ilp64.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/stubs.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/clib.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/clibp.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/lib/win64i8/metislib.lib"
"C:/MSC.Software/Marc/2014.2.0/marc2014.2/xdr_lib/win64/xdr_irc.lib"
```

- Prevent the linker from using the following default libraries:

Linker > Input > Ignore Specific Library: <Edit...>

```
libc.lib
libcmtd.lib
libifcoremt.lib
MSVCRTD.lib
```

- Deactivate the generation of the manifest file:

Linker > Manifest File > Generate Manifest: No (/MANIFEST:NO)

- Use the same output file name with the .pdb extension, e.g. subs.pdb, at the same path, e.g. c:/mentat\_dir/, to generate the program database file:

Linker▷ Debugging▷ Generate Program Database File: c:/mentat\_dir/subs.pdb

- Specify the subsystem for the linker:

Linker▷ System▷ Subsystem: Console

- Enable the checksum of the executable file:

Linker▷ Advanced▷ Set Checksum: Yes

- Add the following additional libraries:

Linker▷ Command Line▷ Additional Options:

libmmd.lib libifcoremd.lib impi.lib ws2\_32.lib kernel32.lib user32.lib netapi32.lib advapi32.lib comdlg32.lib comctl32.lib

4. Now rebuild the solution using the following:

Build▷ Rebuild Solution

The result of these steps is an executable file which can be used in either MARC or MENTAT. However, the analysis should run from the command prompt to enable step-by-step debugging. The following command should be executed for a job named test\_job1:

```
1 run_marc -j test_job1 -prg subs -bg n
```

In addition, a Read statement must be used at the beginning of the program to pause the execution of the subroutine. This will provide time to attach MICROSOFT® VISUAL STUDIO to the running subroutine. The UEDINC subroutine can handle this task. Consider the following code as an example:

```
1 SUBROUTINE UEDINC (uInc, ulncsub)
2
3 IMPLICIT NONE
4
5 c ** Start of generated type statements **
6 INTEGER uInc, ulncsub
7 c ** END of generated type statements **
8
9 IF (uInc .EQ. 0) READ (*,*)
10
11 END
```

This code waits for the user to type an integer as an input in increment zero. Before typing the integer, MICROSOFT® VISUAL STUDIO must be attached to the running subroutine. This can be done in the IDE using the following:

Tools▷ Attach to Process...

In the appearing dialog box, the name of the running subroutine must be selected which is subs.exe in our case. Now, it is possible to toggle breakpoints by pressing F9 when the cursor is at the intended line. In addition, it is possible to use QuickWatch to view the value of a variable. This can be done by executing the following in the IDE:

Debug▷ QuickWatch

Other facilities are also available in the Debug menu of the IDE. After defining the breakpoints, the trivial integer number can be entered and the IDE will stop at the breakpoints of the code. This is a powerful debugging method for complicated programming code.

## 2.5 Miscellaneous Tools

Although subroutines are the strong side of MARC/MENTAT, sometimes the tasks at hand are not that complicated and/or the user is not that acquainted with programming skills. For such cases, MARC/MENTAT enables the user to simply use either some recorded commands via procedure files or some PYTHON scripts for slightly more complex tasks. In this section, these two tools are briefly introduced. In addition, the method of using compiled C libraries is demonstrated by an example. Any of these miscellaneous tools may come in handy for appropriate cases.

### 2.5.1 Procedure Files

Every interaction with MENTAT will finally lead to executing a *command*. Selecting a menu, clicking on a button, entering a value in a field of a dialog, among others, will issue several commands. In the interface, these commands and the user interaction appear in the command area. A command is just a line of text which has a specific meaning for MENTAT. A *procedure file* is nothing but a text file with .proc extension which contains an ordered chain of commands. As mentioned previously, all these commands in every running session of the MENTAT software will be saved as the mentat.proc file in the working directory of MENTAT. Similar to this concept, the user can save a customized procedure file to tackle some obstacles. In this subsection, the potential of procedure files is briefly introduced.

The procedure capability can have various applications, such as the following:

- Procedure files can be used to handle the typical parts of the modeling process. Repetitive operations such as entering the coordinates of a geometry can easily be handled by procedures whereas the more specific steps can be followed up by the user.
- Since modifying a procedure file can readily be done by a simple text editor, it makes generating the corresponding models for a parametric study an easy process.
- It can serve as an educational tool by which it is made possible to investigate the commands of MENTAT by modifying the model creation process. This can enlighten the user about the intricate chain of commands in a modeling process.
- The default procedure file of MENTAT, i.e mentat.proc, is actually the most up-to-date backup of the current session which can play a critical protecting role in the case of a software crash.

- Sometimes, it is more convenient to organize the results of the analysis in the form of a procedure file by which the table/graph of the result can be created in MENTAT.
- Any possible chain of commands which reveals some errors or bugs of the software can be recorded using a procedure file for reporting purposes.

In Fig. 2.1, three procedure files are illustrated: a custom input file as an input for MENTAT to run a set of commands (`inscript.proc`), a custom output file as an output of an analysis (`outscript.proc`) and the default session script of MENTAT (`mentat.proc`) as a backup. A subroutine can be used to generate the procedure for the output file of the results, e.g. a table for the history plot.

The next question is how to make a procedure file. In the beginning, the most direct and recommended way of creating a procedure file is via MENTAT itself and then modify it by a text editor to suit the user's case. As the experience of the user increases, one will be able to create the procedure file by the text editor from the very beginning. This is most likely possible for some simple tasks such as creating a table or simple geometries (see [31] for such examples).

It is advised to observe the command area while working with MENTAT to get acquainted with the syntax of commands. At more advanced levels, the process of creating the procedure file can be automatized by means of a programming language; more specifically by a user subroutine coded in FORTRAN. This approach can liberate the user of the tiring process of organizing the results of analyses.

It is even possible to use other programming languages to create procedure files which generate models for parametric studies. For instance, a very common table in most analyses is a Ramp table which helps applying a boundary condition incrementally. A simple procedure file can create such a table with the following lines:

```

1 *new_md_table 1 1
2 *table_name Ramp
3 *set_md_table_type 1
4 increment_number
5 *table_add
6 0 0
7 1 1

```

Note that each command starts with an asterisk; the other lines without an asterisk are the inputs. The data points, i.e. next two lines after the command `*table_add`, can be replaced by the results of an analysis to create a ready-to-use procedure file.

Procedure files are quite useful when dealing with the aforementioned tasks which are mainly a linear chain of commands. However, they do not provide the user with the abilities of a programming language such as performing calculations and using control structures. For example, conditional statements and loops cannot be incorporated in procedure files. MENTAT also provides the solution for such cases, i.e. PYMENTAT which combines the ability of the PYTHON programming language with the typical commands of MENTAT. In the next subsection, this topic is discussed briefly.

## 2.5.2 Python and Mentat

Although PYTHON is a general-purpose programming language, it is mostly famous for being an object-oriented scripting language. Readability, coherence, portability, development productivity, extensive support libraries, component integration and many other fine characteristics make PYTHON a favorable scripting tool. In our context, PYTHON serves as a product customization/extension tool which can invoke library functions and interact with the MENTAT interface. This gives the advantage of end-user product customization without manipulation of the source code. On the other hand, PYTHON is a very high-level programming language which makes it rather slow in particular cases. In such occasions, a lower level compiled code can be accessed to reach a better performance.

PYTHON interacts with MENTAT via a module named PYMENTAT which sends a sequence of commands to MENTAT. This is similar to the previously introduced procedure concept, but additionally the capabilities of a programming language are also provided. In other words, it is possible to convert a procedure file to a PYMENTAT script to benefit from the general potential of the programming language; a script is provided for this purpose in [27]. When the procedure file is converted, additional modifications can be applied to the file, e.g. adding loops and calculations. While this is the recommended approach for a beginner, a more advanced user can start readily from the PYMENTAT script itself.

There is another module which acts independently from MENTAT on the post files of MARC, i.e. PYPOST. By this module, the output results which are already recorded in the post file, can be read, processed and plotted.

As an example consider the following PYMENTAT code which generates a grid of nodes using parameters from MENTAT:

```

1 from py_mentat import *
2 def make_grid(x, y, z, dx, dy, dz, xs, ys, zs):
3     x_dist = dx / xs
4     y_dist = dy / ys
5     z_dist = dz / zs
6     xt=x
7     yt=y
8     zt=z
9     for i in range(0, zs):
10        for j in range(0, ys):
11            for k in range(0, xs):
12                cmd = "*add_nodes_%f_%f_%f" % (xt, yt, zt)
13                py_send(cmd)
14                xt = xt + x_dist
15                yt = yt + y_dist
16                xt = x
17                zt = zt + z_dist
18                yt = y
19            return
20
21 def main():
22     nx = py_get_int("nx")
23     ny = py_get_int("ny")
24     nz = py_get_int("nz")
25     x_length = py_get_float("x_length")
26     y_length = py_get_float("y_length")
27     z_length = py_get_float("z_length")

```

```

28  make_grid (0,0,0,x_length,y_length,z_length,nx,ny,nz)
29  return
30
31  if __name__ == '__main__':
32      main()

```

In this listing, *nx*, *ny* and *nz* are the number of nodes and *x\_length*, *y\_length* and *z\_length* are the total length of the grid in the *X*-, *Y*- and *Z*-direction, respectively. These are the names of the parameters which can be defined using the MENTAT parameters dialog which appears using the following command:

① Tools > Parameters

After defining the parameters, the PYMENTAT script can be selected and run by the following command:

① Tools > Python > Run

Using PYMENTAT and PYPOST scripts will help the user to automate many tasks without a great deal of programming expertise. However, the downside of the deal may be the slower performance especially if using heavy PYMENTAT scripts which interact directly with MENTAT. The other option in this case may be using some compiled FORTRAN subroutines to boost the speed of the process.

### 2.5.3 C Programming Language

The two previous subsections introduced options for rather unexperienced programmers. Here, a brief introduction for mixed-language programming, especially for the C family language, is provided and it is concluded by an example.

In general, approaching mixed-language programming is due to the availability of some existing code or the implementation problems in a particular language [18]. A mixed-language approach is beneficial when the required code is available but not in the main language and it is easier to just prepare a way of communication between two languages. For the case of MARC, the *interoperability* between FORTRAN and C programming language is supported. Namely, it is possible to use an entity, e.g. a function, a derived type, a variable etc. of FORTRAN in C or vice versa. Because the subroutines of MARC are in FORTRAN, any C code library can be used to facilitate the calculations. In addition, even C++ codes, if adapted carefully, can be used in a FORTRAN code.

The interoperability between the FORTRAN and C programming language is a standard part of FORTRAN. There are some delicate matters which can be addressed by means of appropriate references. However, as a simple example, the bubble sort algorithm is coded in a C file named *cfuctions.c* containing the following lines:

```

1  void bubblesort (double x[], int len)
2  {
3      int i, j;
4      double temp;
5      for (i = len - 1; i > 0; i--)
6          for (j = 0; j < i; j++){

```



```

7     if (x[j] > x[j+1]){
8         temp = x[j+1];
9         x[j+1] = x[j];
10        x[j] = temp;
11    }
12    }
13 }

```

In this code, an array of double precision numbers (x[]) with a number of elements (len) is sorted by means of the subroutine bubblesort. Using a module such as the following will set up a neat connection between this code and FORTRAN:

```

1  MODULE CTools
2      USE iso_c_binding
3      IMPLICIT NONE
4
5      INTERFACE
6          SUBROUTINE bubble (a,b) BIND(c)
7              IMPORT
8                  INTEGER (KIND = c_int), VALUE, INTENT(IN) :: b
9                  REAL (KIND = c_double), INTENT(INOUT), DIMENSION(*) :: a
10             END SUBROUTINE bubble
11        END INTERFACE
12
13    CONTAINS
14
15        SUBROUTINE BubbleSort (aList , itemCount)
16
17            INTEGER, INTENT(IN) :: itemCount
18            REAL*8, DIMENSION(*), INTENT(INOUT) :: aList
19
20            CALL bubble (aList , itemCount)
21
22        END SUBROUTINE BubbleSort
23
24    END MODULE CTools

```

The module CTools contains a subroutine in FORTRAN which simply executes the C function. Since the call is an external one, the interface is explicitly declared. This module can be used in any FORTRAN program to sort one-dimensional arrays, such as the following:

```

1  INCLUDE 'ctools.for'
2  PROGRAM FortranC
3
4      USE ctools
5      IMPLICIT NONE
6
7      INTEGER (KIND = c_int) :: itemCount
8      REAL (KIND = c_double), DIMENSION(10) :: theList
9
10     itemCount = 10
11     theList(1) = 100.55D0
12     theList(2) = -12.45D0
13     theList(3) = 0.23D0
14     theList(4) = 0.23D0
15     theList(5) = 40.23D0
16     theList(6) = 0.23D0
17     theList(7) = -20.23D0
18     theList(8) = 0.23D0
19     theList(9) = 300.D0
20     theList(10) = 20.3D0
21
22     CALL BubbleSort (theList , itemCount)

```

```
23     print *, theList
24
25     END PROGRAM FortranC
```

To obtain an executable for an individual FORTRAN program, such as this example, the C and FORTRAN codes must be compiled separately to their corresponding object files and then using the FORTRAN compiler, i.e. `ifort`, all the objects can be linked together. For example, to compile the C file, i.e. `cfunctions.c`, use the following command in the command-prompt to generate the object file:

```
1  cl -c cfunctions.c
```

Using the FORTRAN compiler, compile the module file, i.e. `cTools.for`, and the main program file, i.e. `FortranC.for`, with the following commands:

```
1  ifort -c fortanc.for
```

Note that because the module is already included in the main program, the compiler will automatically generate the corresponding object and module files. Finally, all the object files must be linked together by the following command:

```
1  ifort -o final fortanc.obj cfunctions.obj
```

The executable file, i.e. `final.exe`, will be the result of this command.

Alternatively, it is possible to use the batch file mentioned in Sect. 2.4.5 and set the environmental variable `%my_object%` to the compiled object file of the C program. For our case, a command such as the following will do the trick:

```
1  SET %my_object%='cfunctions.obj'
```

# Chapter 3

## Basic Examples

**Abstract** In this chapter, a few simple problems are stated and solved with the help of subroutines. These examples will help the reader to review the programming concepts introduced in the earlier chapters and practice the concepts behind MARC/MENTAT. The main focus will be on structural problems. The subroutine structure is kept simple and the interaction between subroutines is minimized. This chapter is suitable for the intermediate programmer and provides a good foundation for more complex coding problems presented in the following chapters.

### 3.1 Overview

In the previous chapter, the fundamentals of MARC/MENTAT were investigated with a special focus on the subroutine facility. This knowledge combined with the programming skills, equips the FE user with powerful skills to deal with complex and unconventional problems. In this chapter, this capability is illuminated by simple examples, namely some practical problems are with the help of some selected subroutines. The main area of concern will be simple structural problems which require basic programming skills and simple interactions between subroutines. The purpose of this chapter is to familiarize the reader with employing subroutines in FE problems.

### 3.2 Examples

When approaching these examples, all default values will be used for any options unless otherwise stated. Therefore, it is always assumed that creating a new model is started by resetting MENTAT which can be done by executing the following:

- ① File ▷ New
- ⑧ \*reset

This will ensure that all the default values of the dialog boxes are restored and that MENTAT is back to the default state. Numerical values are given without units and can be understood as consistent units.

### 3.2.1 FORCDT

*Example 3.1* This example illustrates the use of the FORCDT subroutine in a static analysis in which a moving point load is applied on a cantilever beam. The beam is modeled by means of 11 nodes and 10 elements of type 52, i.e. a two-node Euler-Bernoulli straight beam element ( $E = 200 \times 10^3$ ). The beam has a square cross section with sides equal to 40. A vertical load ( $F = -10$ ) is applied on each node in 10 increments, namely in increment 1 the load is applied on node 1, in increment 2 the load is applied only on node 2, and so forth. The structure is shown in Fig. 3.1 with the force acting on the first node in the first increment. The placement of the force in the succeeding increments is distinguishable by dashed arrows.

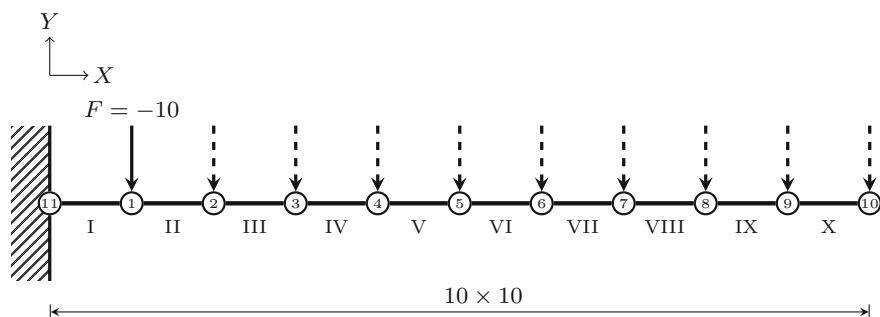
The output of this subroutine in a static structural analysis is the displacement and/or the point load of the nodes. For a table-driven format, the displacement can be either incremental or total which is indicated by the `iaclflag` flag. However, the point load is the total point load at the end of the increment. In the table-driven format, this subroutine is activated within the corresponding kinematic boundary condition or point load by changing the method field from Entered values to User Sub. `Forcdt`. Note that at least one of the degrees of freedom in the boundary condition and/or point load must be selected to make the subroutine active. Since the real values are assigned within the subroutine, the entered value and the selected degree of freedom are arbitrary. Then the subroutine is executed for each of the selected nodes.

The following code is used for the subroutine to apply the constant moving load:

```

1  SUBROUTINE FORCDT(u, v, a, dp, du, time, dtime, ndeg, node,
2  & ug, xord, ncrd, iaclflg, inc, ipass)
3
4  IMPLICIT NONE

```



**Fig. 3.1** Application of a moving force using the FORCDT subroutine

```

5  !      ** Start of generated type statements **
6      REAL*8 a, dp, dtime, du
7      INTEGER iacflg, inc, ipass, ncrd, ndeg, node
8      REAL*8 time, u, ug, v, xord
9  !      ** End of generated type statements **
10     DIMENSION u(ndeg), v(ndeg), a(ndeg), dp(ndeg), du(ndeg), ug(ndeg),
11     *          xord(ncrd)
12
13     IF (inc .EQ. node) THEN
14         dp = [0.0, -10.0, 0.0]
15     END IF
16
17     RETURN
18     END
    
```

In this listing, the node number (node) is compared to the increment number (inc). If the equality condition is true, the value  $-10$  is assigned to the second component of the point load array (dp). The result of this value assignment is a moving load which starts on the first node at the first increment and moves node-by-node to the last node at the last increment.

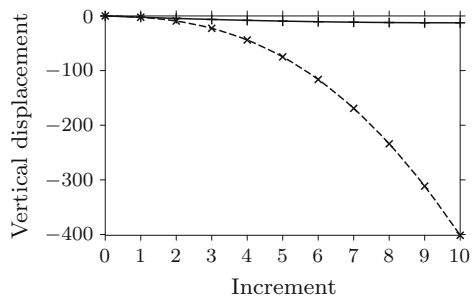
If the non-table-driven format is used, the FORCDT input file option must be used to introduce the list of the nodes for which the subroutine must be executed. For our case, the following lines must be added to the input file:

```

1  FORCDT, 1
2  1 TO 10
    
```

Note that for the non-table-driven format, the point load is applied incrementally. Therefore, by only changing the format to the non-table-driven, the forces of the previous increments will be preserved, i.e. in the last increment the applied force of the first node will be equal to 100. The results of both table-driven and non-table-driven input files are illustrated in Fig. 3.2. In this figure, the displacement of node 10 along the Y-axis is plotted against the increment number. In the example of the table-driven format, the maximum vertical displacement of the node 10 is equal to  $-12.583$  whereas the same quantity for the non-table-driven format is equal to  $-401.396$ .

**Fig. 3.2** Vertical displacement of node 10 Versus increment number



⋆ Table-driven format    ⋆-x Non-table-driven format

### 3.2.2 FORCEM

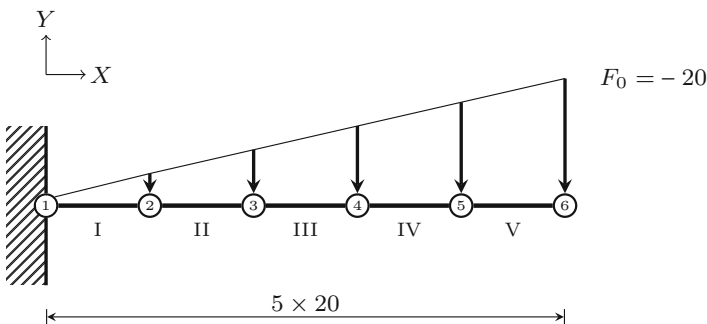
*Example 3.2* This example illustrates the use of the FORCEM subroutine in a static analysis in which a cantilever beam is loaded with a linearly distributed load. The beam is modeled by means of 6 nodes and 5 elements of type 52, i.e. a two-node Euler-Bernoulli straight beam ( $E = 200 \times 10^3$ ) element integrated by three point Gauss integration for numerical calculations. The beam cross-section is a square with edges equal to 5. The discretized structure is shown in Fig. 3.3 along with the triangular load. The load is assumed to be a function of the  $X$ -coordinate and time. In addition, the linear load is increased incrementally with time. The total equivalent load is  $F = -1000$  which is distributed by the following equation:

$$F(x, t) = -2 \times \frac{1000}{100} \times \frac{X}{100} \times t = -0.2Xt. \quad (3.1)$$

In most mechanical problems, a distributed load is required to be specified as a function of time, space coordinates, or both. The FORCEM subroutine enables using nonuniform distributed loads applied on the elements as a function of some independent variables. This subroutine is used to specify either volumetric body forces (e.g. the gravity load) or surface loads (e.g. shear forces).

The value of the arbitrary distributed load is specified in the integration points by which the equivalent nodal loads will be calculated by MARC. Therefore, this subroutine will be called for each integration point of the assigned elements. In our case, the structure consists of 5 elements each with three integration points, and the load is applied using the AUTO LOAD option in 5 equal increments. Thus, the subroutine is going to be executed for  $5 \times 3 \times 5 = 75$  times during the analysis.

In MENTAT, triggering this subroutine is done by selecting User Sub. Formem as the evaluation method of the distributed load for a specific degree of freedom. Note that any value for the degree of freedom can be entered and will not be used as a scale factor for the subroutine. The distributed load can be either an edge load, a



**Fig. 3.3** Application of a non-uniform linear load using the FORCEM subroutine

face load or a global load. Note that the subroutine will run only for the selected degree of freedom. Alternatively, the DIST LOADS input option can be used to flag the subroutine. This option can be used in both table- and non-table-driven formats. Note that in the table-driven format, the value of the load at the integration points, i.e. `press`, is the total value whereas for the non-table-driven style, it is an incremental value.

Another point of difference is that for the table-driven style, the direction of the load is already selected in MENTAT but in the non-table-driven style it is necessary to specify `IBODY` for the DIST LOADS input file option. Depending on the element type and the edge/face on which the load is applied and whether a subroutine is used or not, the value of the `IBODY` differs. For our example, which is a nonuniform load applied on a beam, `IBODY` is equal to 111. For more information on other loading options, refer to [23].

For a time-dependent loading, the increment number or the time increment can be obtained from the `concom` and the `creeps` common blocks, respectively. It is important for an incremental load to be specified as a function of time. Therefore, during cut-backs the correct load can be evaluated. In the current example, the `timinc` variable of the `concom` common block is used to determine the current time of each increment out of the total time of 1.

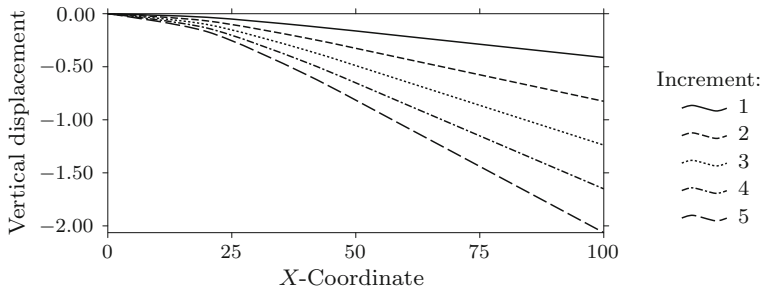
The following code is used for the subroutine:

```

1  SUBROUTINE FORCEM( press , th1 , th2 , nn , n )
2
3      IMPLICIT NONE
4      INCLUDE 'creeps'
5
6      !    ** Start of generated type statements **
7      REAL*8 :: prnorm(3)
8      CHARACTER*32 :: cdum, bcname
9      COMMON /marc_lpres3/prnorm
10     COMMON /marc_bclabel/cdum, bcname
11     INTEGER n, nn
12     REAL*8 press, th1(*), th2(*)
13     !    ** End of generated type statements **
14     DIMENSION n(10)
15
16     REAL*8, PARAMETER :: TFORCE = -1000.0D0, TLEN = 100.0D0
17     REAL*8 :: maxForcePerLen, curTime, totTime
18
19     curTime = timinc + cptim
20     totTime = 1.0D0
21
22     maxForcePerLen = (2.0*TFORCE/TLEN)*curTime/totTime
23     press = (th1(1)/TLEN)*maxForcePerLen
24
25     RETURN
26     END

```

In this listing, the common block is included in line 4 and two double precision constants are introduced, i.e. `TFORCE` and `TLEN`, which are the total applied force on the beam and the total length of the beam, respectively. Three auxiliary double precision variables are used to hold the current time at the end of the current increment (`curTime`), the total time of the loading (`totTime`) and the maximum force per length of the beam (`maxForcePerLen`). Note that the coordinate system is set at the first node. Therefore, the value of the distributed load at the current integration point (`press`)



**Fig. 3.4** Vertical displacement of the beam in every increment

can readily be calculated using its first coordinate (`th1(1)`). In addition, the total force is applied linearly as a function of time (`curTime/totTime`). It is worth mentioning that there is no need to specify the cosine direction of the load since the table-driven style is used in our example. The vertical displacement of the beam for every increment is illustrated in Fig. 3.4.

Although the most general way of nonuniform loading is provided by the subroutine, this example can be carried out using only a table as function of two independent variables, i.e. time and  $X$ -coordinates.

### 3.2.3 WKSLP

*Example 3.3* This example illustrates the use of the WKSLP subroutine in a nonlinear static analysis in which a rod is stretched by an axial displacement ( $u_0$ ). The rod is modeled by means of 2 nodes and one element of type 9, i.e. a two-node truss element ( $E = 200 \times 10^3$ ,  $L = 100$  and  $A = 400$ ). The load is applied linearly in 10 fixed steps using a displacement boundary condition resulting in the final  $u_0$  displacement in the  $X$ -direction. However, the material is defined by an elastic-plastic behavior with isotropic hardening. The yield criterion is according to von Mises. In Fig. 3.5(a), the rod is shown along with the displacement boundary condition applied on the second node and in Fig. 3.5(b), the stress versus the plastic strain of the material is illustrated. The data points of the hardening curve are approximated by the following function:

$$\sigma(\epsilon_p) = -\frac{66}{0.0225}\epsilon_p^2 + \frac{132}{0.15}\epsilon_p + 190. \quad (3.2)$$

The WKSLP subroutine enables the user to specify the current yield stress by introducing the hardening curve as a function of the equivalent plastic strain and/or temperature. The yield function itself or the slope(s) of the hardening curve can be specified within the subroutine. The subroutine is executed for each of the integration points which undergoes a plastic deformation.



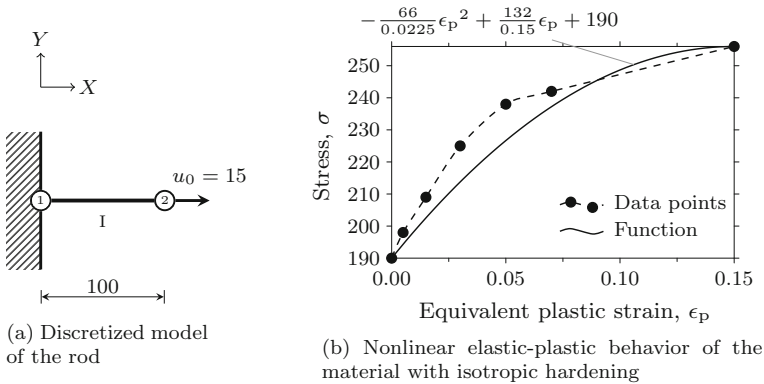


Fig. 3.5 Axial loading of a nonlinear rod defined by the WKSPL subroutine

In MENTAT, the subroutine activation can be done for a material by selecting User Sub. Wkslp as the method in the plasticity properties. The plasticity dialog can be accessed by executing the following command:

③ Material Properties > Properties > Plasticity

The listing for the subroutine is as follows:

```

1  SUBROUTINE WKSPL(m,nn ,kcus ,matus ,slope , ebarp , eqrate , stryt , dt ,
2  * ifirst)
3  IMPLICIT NONE
4  ! ** Start of generated type statements **
5  REAL*8 dt, ebarp, eqrate
6  INTEGER ifirst, kcus, m, matus, nn
7  REAL*8 slope, stryt
8  ! ** End of generated type statements **
9  DIMENSION matus(2), kcus(2)
10
11  stryt = (-66.D0/0.0225.D0)*ebarp**2 + (132.D0/0.15D0)*ebarp + 190.D0
12  slope = (-2.D0*66.D0/0.0225.D0)*ebarp + (132.D0/0.15D0)
13  RETURN
14  END
    
```

In this listing, the yield stress (stryt) is specified in terms of the plastic strain (ebarp) and in line 11, the slope of the hardening curve (slope) is specified.

Now let us consider the case that an explicit function is not provided. If the table-driven input is chosen, it is required to create a table for the data points. It is possible to create a table in MENTAT, manually, by a procedure file or directly by using the TABLE parameter. A procedure such as the following can be used to define the table in MENTAT:

```

1  *new_md_table 1 1
2  *table_name data_points
3  *set_md_table_type 1
4  eq_plastic_strain
5  *table_add
6  0 190
7  0.005 198
8  0.015 209
    
```

```

9 0.03 225
10 0.05 238
11 0.07 242
12 0.15 256
13
14 *table_fit
15 *set_md_table_extrap 1 off
16 *edit_table table 1

```

Note that by adding the command in line 15, no extrapolation is done for the out-of-range independent variables (Sect. 2.2.5). The created table can be used to directly define the material without using the WKSPLP subroutine by choosing Table as the method for plasticity properties and selecting the data\_points table as the table for the yield stress. Note that by selecting a table, the Yield Stress field will be considered as a scale coefficient for the values of the table; in our case a value of 1 is advised for simplicity.

If it is required that the data points are used in a subroutine, then the previous method will not work. This is because MENTAT does not transfer the unused tables to the input file and therefore, the first two methods of creating tables cannot be used for this case. As a result, the TABLE input file option must be used. The following lines must be added to the model definition section of the input file:

```

1 TABLE, data_points
2 2,1,,2
3 15,7,1,
4 0.000D0, 190.D0
5 0.005D0, 198.D0
6 0.015D0, 209.D0
7 0.030D0, 225.D0
8 0.050D0, 238.D0
9 0.070D0, 242.D0
10 0.150D0, 256.D0

```

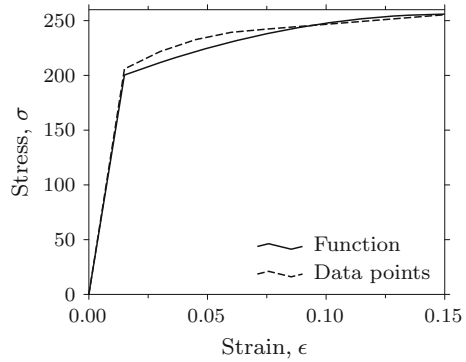
Note that in line 3, the number 15 indicates the equivalent plastic strain as the independent variable and the number 7 is the number of data points. In addition, the number 1 indicates that no extrapolation is done for the out-of-range independent variable and instead, the value of the first and last point is used. To access the tabular values within the subroutine, the TABVA2 utility subroutine is used as the following:

```

1 SUBROUTINE WKSPLP(m,nn,kcus,matus,slope,ebarp,eqrate,stryt,dt,
2 & ifirst)
3 IMPLICIT NONE
4 INCLUDE 'ctable'
5 ! ** Start of generated type statements **
6 REAL*8 dt,ebarp,eqrate
7 INTEGER ifirst,kcus,m,matus,nn
8 REAL*8 slope,stryt
9 ! ** End of generated type statements **
10 DIMENSION matus(2),kcus(2)
11
12 INTEGER, PARAMETER :: TABLEID = 2
13 REAL*8, PARAMETER :: REFVALUE = 1.D0
14
15 eqpl = ebarp
16 CALL tabva2(REFVALUE,stryt, TABLEID, 0, 1)
17 RETURN
18 END

```

**Fig. 3.6** Results of the WKSLEP subroutine using the explicit function versus interpolated data points



In this listing, the equivalent plastic strain argument (e $\bar{\epsilon}$ ) is passed to the independent variable eqpl in the common block ctable. The scale factor is set to 1 (REFVALUE = 1.D0) to extract the original values of the table.

The results of either one of the methods, i.e. the approximate function or the data-points, are quite identical (see Fig. 3.6). Either of the mentioned methods incorporate the table-driven style with the subroutine support of MARC/MENTAT. However, the same task can be tackled by means of the non-table-driven style using the WORK HARD model definition option. The following lines must be entered in the input file:

```

1  WORK HARD, DATA
2  7, 0, 1, ,
3  190.D0, 0.000D0
4  198.D0, 0.005D0
5  209.D0, 0.015D0
6  225.D0, 0.030D0
7  238.D0, 0.050D0
8  242.D0, 0.070D0
9  256.D0, 0.150D0
    
```

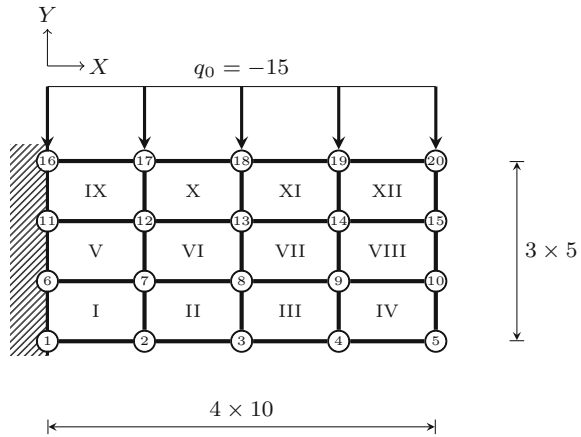
Alternatively, instead of the data points, the slopes of the hardening curve can be specified.

### 3.2.4 PLOTV

*Example 3.4* In this example, the PLOTV subroutine is used to provide two scalar user defined elemental variables at the post processing stage. A cantilever beam is modeled by means of 4-node quadrilateral plane stress elements, i.e. elements of type 3 ( $E = 200 \times 10^3$ ,  $\nu = 0.2$  and  $t = 1$ ). A uniformly distributed load ( $q_0 = -15$ ) is applied per length on the top edge of the beam. The discretized model is illustrated in Fig. 3.7 which is composed of 20 nodes and 12 elements lying in the  $X$ - $Y$  plane.

This subroutine is executed once for each one of the user defined variables which can be up to 200 variables. To calculate the scalars, the subroutine runs for each integration point of the selected elements. Although all the stresses, strains and

**Fig. 3.7** Discretized model for the PLOTV subroutine example



state variables are provided within the subroutine, sometimes accessing the material properties will be required which can be done by including the matdat common block.

In order to flag the subroutine, select any number of user defined variables, to appear in the post file, under Available Element Scalars in the Results dialog:

③ Jobs > Jobs > Properties > Job Results

For the current example, check the boxes in front of User Defined Var #1, # 2 and # 3. This will introduce three user defined variables in the post file which will be handled by the PLOTV subroutine. The number of user defined variables is provided by the jpltd variable. The first and second variables are considered to be the octahedral normal stress ( $\sigma_{oct}$ ) and octahedral shear stress ( $\tau_{oct}$ ), respectively. The first invariant of stress ( $I_1$ ) and second invariant of stress ( $I_2$ ) will be used to carry out the calculation using the following formulas:

$$\sigma_{oct} = \frac{1}{3}I_1, \tag{3.3}$$

$$\tau_{oct} = \frac{1}{3}\sqrt{2I_1^2 - 6I_2}. \tag{3.4}$$

The invariants of the stress tensor are calculated using the following formulas:

$$I_1 = \sigma_{xx} + \sigma_{yy} + \sigma_{zz}, \tag{3.5}$$

$$I_2 = \begin{vmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{vmatrix} + \begin{vmatrix} \sigma_{yy} & \sigma_{yz} \\ \sigma_{zy} & \sigma_{zz} \end{vmatrix} + \begin{vmatrix} \sigma_{xx} & \sigma_{xz} \\ \sigma_{zx} & \sigma_{zz} \end{vmatrix}. \tag{3.6}$$

Element 3 only provides three stresses ( $\sigma_{xx}$ ,  $\sigma_{yy}$  and  $\sigma_{xy}$ ) and three strains ( $\epsilon_{xx}$ ,  $\epsilon_{yy}$  and  $\epsilon_{xy}$ ). Therefore, the formulas can be simplified to the following for the plane stress case:

$$I_1 = \sigma_{xx} + \sigma_{yy}, \tag{3.7}$$

$$I_2 = \sigma_{xx}\sigma_{yy} - \sigma_{xy}^2. \tag{3.8}$$

The third user defined variable is used to calculate the strain along the z-axis, i.e. the strain normal to the plane of the elements. Note that by default, MARC does not provide this quantity for isotropic materials in a plane stress analysis. The elastic modulus ( $E$ ) and the Poisson’s ratio ( $\nu$ ) are used to calculate the strain by the following formula:

$$\epsilon_{zz} = \frac{-\nu}{E}(\sigma_{xx} + \sigma_{yy}). \tag{3.9}$$

The following listing is used for the subroutine:

```

1  SUBROUTINE PLOTV(v,s,sp,etot,eplas,ecreep,t,m,nn,kcus,ndi,
2  & nshear,jplbcd)
3
4  IMPLICIT NONE
5  INCLUDE 'matdat'
6
7  ! ** Start of generated type statements **
8  REAL*8 ecreep, eplas, etot
9  INTEGER jplbcd, kcus, m, ndi, nn, nshear
10 REAL*8 s, sp, t, v
11 ! ** End of generated type statements **
12 DIMENSION s(*), etot(*), eplas(*), ecreep(*), sp(*), m(4), kcus(2),
13 & t(*)
14
15 REAL*8 :: inv1, inv2
16
17 IF (jplbcd .EQ. 1) THEN
18   inv1 = s(1) + s(2)
19   v = inv1 / 3.D0
20 ELSE IF (jplbcd .EQ. 2) THEN
21   inv1 = s(1) + s(2)
22   inv2 = s(1)*s(2)-s(3)**2
23   v = SQRT(2.D0*(inv1**2)-6.D0*inv2) / 3.D0
24 ELSE IF (jplbcd .EQ. 3) THEN
25   v = -xu(3)*(s(1) + s(2))/et(3)
26 END IF
27 RETURN
28 END

```

Line 5 is used to gain access to the material properties, i.e. elastic modulus (et(3)) and the Poisson’s ratio (xu(3)). The invariants are stored in the inv1 and inv2 variables. The stresses are provided by the array s(\*) and the sequence of their storage for the element 3 is  $\sigma_{xx}$ ,  $\sigma_{yy}$  and  $\sigma_{xy}$ .

The result of the analysis in terms of extrapolated nodal values of element 8 is summarized in Table 3.1.

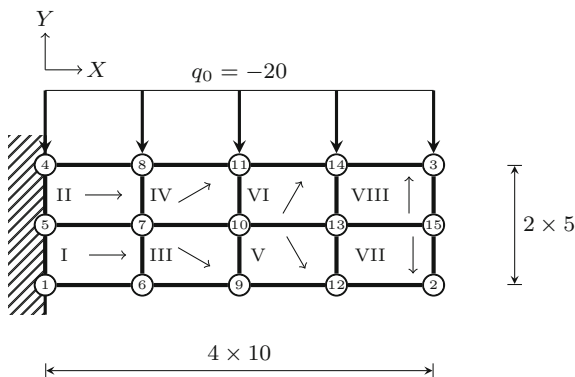
**Table 3.1** Analysis result for nodes of element 8 (linearly interpolated)

| Node | Quantity       |              |                           |
|------|----------------|--------------|---------------------------|
|      | $\sigma_{oct}$ | $\tau_{oct}$ | $\epsilon_{zz}$           |
| 9    | -5.36189       | 9.59335      | $+1.60857 \times 10^{-5}$ |
| 10   | -3.69377       | 5.20001      | $+1.10813 \times 10^{-5}$ |
| 15   | -1.35280       | 8.13492      | $+4.05840 \times 10^{-6}$ |
| 14   | +0.38942       | 12.3154      | $-1.16828 \times 10^{-6}$ |

### 3.2.5 HOOKLW and ORIENT2

*Example 3.5* This example illustrates the use of the HOOKLW subroutine in defining an anisotropic material which degrades as a function of time. A cantilever beam is modeled using 4-node plane stress elements of type 3 ( $t = 1$ ). A constant distributed load per length of the beam ( $q_0 = -20$ ) is applied to its upper edge in ten equal increments. In Fig. 3.8, the discretized structure is illustrated in the  $X$ - $Y$  plane. The material is assumed to be orthotropic with hypothetical orientations which are illustrated schematically in the figure by using the arrows and specified more in detail in Table 3.2.

The HOOKLW subroutine is used to specify a general stress-strain relationship by the user. In other words, in this subroutine the elasticity matrix or the compliance matrix of an anisotropic material can be specified which will cover more general cases. It is a simpler alternative when compared to the ANELAS subroutine. These



**Fig. 3.8** Discretized model for the HOOKLW subroutine example

**Table 3.2** Material orientation indicating the  $X$ -axis of the elements

| Element | 1 | 2 | 3   | 4   | 5   | 6   | 7   | 8   |
|---------|---|---|-----|-----|-----|-----|-----|-----|
| Angle   | 0 | 0 | -30 | +30 | -60 | +60 | -90 | +90 |

subroutines can be activated in MENTAT by checking the User Subs. Hooklw/Anelas check-box in the Stress-Strain Law dialog box. This dialog can be reached by executing the following commands:

① Material Properties ▷ Properties ▷ Type: Elastic-Plastic Anisotropic ▷ Stress-Strain Law

For a non-isotropic material, the specification of the material axes is required. The orientation of the material is specified by means of the *element material coordinate system*. More accurately, its relationship with respect to the global coordinate system must be specified. By default, the elemental coordinates coincide with the global ones. Acquiring a different orientation can be done in several ways among which the focus will be on using the subroutines. Alternatively, the orientation can be defined using the Orientations group of commands in the Material Properties tab in MENTAT or more directly the ORIENTATION input file option.

For more general cases, the ORIENT subroutine can be used which incorporates a transformation matrix to specify the orientation. Instead of this obsolete subroutine, the ORIENT2 subroutine is preferred to define the material orientation using a couple of vectors. This subroutine runs once at the beginning of the analysis for every integration point of the model. The material orientation can be specified either in the global coordinate system or the element coordinate system; for the latter, the ilocal flag argument must be set to 1. The element coordinate system is accessible within the subroutine by referring to dircos, a 3 × 3 array, which holds the direction cosines, e.g. dircos(1:3,1) is the local X-axis in the global coordinate system.

The two vectors used to define the orientation are vec1 and vec2. The first one is the first material direction. The third material direction is perpendicular to the plane of these two vectors, i.e. equal to the cross product of the first and second vectors. Finally, the second direction is perpendicular to the plane of the third and first material directions, i.e. their cross products. This method ensures that an orthogonal coordinate system is defined.

Activating the ORIENT2 subroutine is simply done by executing the following and then selecting the elements in the appearing dialog:

① Material Properties ▷ New Orientations ▷ Usub Orient2

The HOOKLW subroutine is called for each integration point of the selected elements. For the case of our example, the stress-strain relationship using the compliance matrix is as follows:

$$\begin{bmatrix} \epsilon_{XX} \\ \epsilon_{YY} \\ \epsilon_{XY} \end{bmatrix} = \begin{bmatrix} \frac{1}{E_{XX}} & \frac{-\nu_{XY}}{E_{YY}} & 0 \\ \frac{-\nu_{XY}}{E_{YY}} & \frac{1}{E_{YY}} & 0 \\ 0 & 0 & \frac{1}{2G_{XY}} \end{bmatrix} \begin{bmatrix} \sigma_{XX} \\ \sigma_{YY} \\ \sigma_{XY} \end{bmatrix}. \tag{3.10}$$

The following values are considered for the orthotropic material and the orientation of the material changes in 30 degree steps as previously illustrated:

$$E_{XX} = 6.91 \times 10^3, \tag{3.11}$$

$$E_{YY} = 8.51 \times 10^3, \quad (3.12)$$

$$\nu_{XY} = 0.32, \quad (3.13)$$

$$G_{XY} = 2.41 \times 10^3. \quad (3.14)$$

$E_{XX}$  and  $E_{YY}$  are the elastic moduli along the  $X$ - and  $Y$ -axis, respectively;  $\nu_{XY}$  is the Poisson's ratio and  $G_{XY}$  is the shear modulus of the material in the global coordinates.

The first and the most general solution is using the two required subroutines, i.e. the HOOKLW and the ORIENT2 subroutines. The listing which contains both of the subroutines is as follows:

```

1  SUBROUTINE HOOKLW(m,nn,kcus,b,ngens,dt,dtdl,e,pr,ndi,nshear,
2  & imod,rprops,iprops)
3  IMPLICIT NONE
4  ! ** Start of generated type statements **
5  REAL*8 b, dt, dtdl, e
6  INTEGER imod, iprops, kcus, m, ndi, ngens, nn, nshear
7  REAL*8 pr, rprops
8  ! ** End of generated type statements **
9
10 DIMENSION b(ngens,ngens), dt(*), dtdl(*), rprops(*), iprops(*),
11 & kcus(2),m(2),e(*),pr(*)
12
13 REAL*8 :: exx, eyy, vyx, gxy
14
15 exx = 6.91D3
16 eyy = 8.51D3
17 vyx = 0.32D0
18 gxy = 2.41D3
19 ! Compliance matrix
20 imod = 2
21
22 b(1,1) = 1.D0/exx
23 b(1,2) = -vyx/eyy
24 b(1,3) = 0.D0
25 b(2,1) = b(1,2)
26 b(2,2) = 1.D0/eyy
27 b(2,3) = 0.D0
28 b(3,1) = 0.D0
29 b(3,2) = 0.D0
30 b(3,3) = 1.D0/(2.D0*gxy)
31
32 RETURN
33 END
34
35 SUBROUTINE ORIENT2(n,nn,kcus,material,matname,icomp,nodes,nnodes,
36 & coord,coordinat,ncoord,dircos,icall,iply,ilocal,ifast,
37 & vec1,vec2,integer_data,real_data)
38 IMPLICIT NONE
39 INTEGER kcus,n,nn,material,nodes,nnodes,ncoord,icall,iply,ilocal
40 INTEGER icomp,ifast,integer_data
41 REAL*8 coord,dircos,vec1,vec2,coordinat,real_data
42 CHARACTER*24 matname
43 DIMENSION n(2),kcus(2),material(2),nodes(*)
44 DIMENSION coord(ncoord,*),dircos(3,3),vec1(3),vec2(3)
45 DIMENSION coordinat(*),integer_data(*),real_data(*)
46
47 SELECT CASE (n(1))
48 CASE (1:2)
49 vec1 = [1.D0, 0.D0, 0.D0]

```



```

50      vec2 = [0.D0, 1.D0, 0.D0]
51      CASE (3)
52          vec1 = [+0.866D0, -0.500D0, 0.D0]
53          vec2 = [-0.500D0, -0.866D0, 0.D0]
54      CASE (4)
55          vec1 = [+0.866D0, 0.500D0, 0.D0]
56          vec2 = [-0.500D0, 0.866D0, 0.D0]
57      CASE (5)
58          vec1 = [+0.500D0, -0.866D0, 0.D0]
59          vec2 = [-0.866D0, -0.500D0, 0.D0]
60      CASE (6)
61          vec1 = [+0.500D0, 0.866D0, 0.D0]
62          vec2 = [-0.866D0, 0.500D0, 0.D0]
63      CASE (7)
64          vec1 = [0.D0, 1.D0, 0.D0]
65          vec2 = [1.D0, 0.D0, 0.D0]
66      CASE (8)
67          vec1 = [0.D0, -1.D0, 0.D0]
68          vec2 = [-1.D0, 0.D0, 0.D0]
69      END SELECT
70      ilocal = 0
71
72      RETURN
73      END

```

In this listing, the compliance matrix is used to define the elastic matrix. This is done by assigning `imod` equal to 2. It is not necessary to define separate variables for the material properties such as `exx` and `gxy` but it adds to the readability of the code. In the `orient2` subroutine different values are assigned to the direction vectors based on the element number; a `SELECT CASE` structure covers all the possible cases. Note that `ilocal = 0` indicates that all the vectors are defined in the global coordinate system.

It would be helpful to check the orientations visually. However, visualizing the element coordinate system is possible only at the post-processing stage. It can be done by selecting the orientations check box:

① View ▷ Plot Control ▷ Orientations Settings

Additionally, the components of the direction vectors can be obtained by selecting the element post codes 691 and 694 which refer to the first and second elemental orientation vectors. In `MENTAT`, it can be done by selecting the 1st Element Orientation Vector and 2nd Element Orientation Vector in the Available element Scalars of the Job Results dialog. Note that it is possible to specify different orientations for the integration points of a single element. However, it is not possible to visualize them separately. Nevertheless, an average representation for each element will be available in the post-processing visualization.

It is also possible to carry out the same example without using the subroutines and instead, simply modify the input file. In this case, the anisotropic material properties must be entered using the `ORTHOTROPIC` or `ANISOTRIOPIC` option; the former is used and the same material properties are entered via `MENTAT`. This can be done by selecting the Elastic-Plastic Orthotropic material type in the material properties dialog.

To define the material orientations via `MENTAT`, it is necessary to use seven coordinate systems. Each can be defined by entering the orientation angles  $-30$ ,  $30$ ,  $-60$ ,  $60$ ,  $-90$  and  $90$  degrees for the elements III, IV, V, VI, VII and VIII, respectively. Note that elements I and II have the default orientation angle of zero.

**Table 3.3** Analysis results for nodes of element VIII

| Node | Quantity      |         |          |
|------|---------------|---------|----------|
|      | $\sigma_{eq}$ | $u_X$   | $u_Y$    |
| 13   | +3.13989      | 0.00053 | -0.68225 |
| 15   | +1.65111      | 0.00081 | -1.00752 |
| 3    | +3.93205      | 0.16413 | -1.00848 |
| 14   | +10.1924      | 0.15977 | -0.68441 |

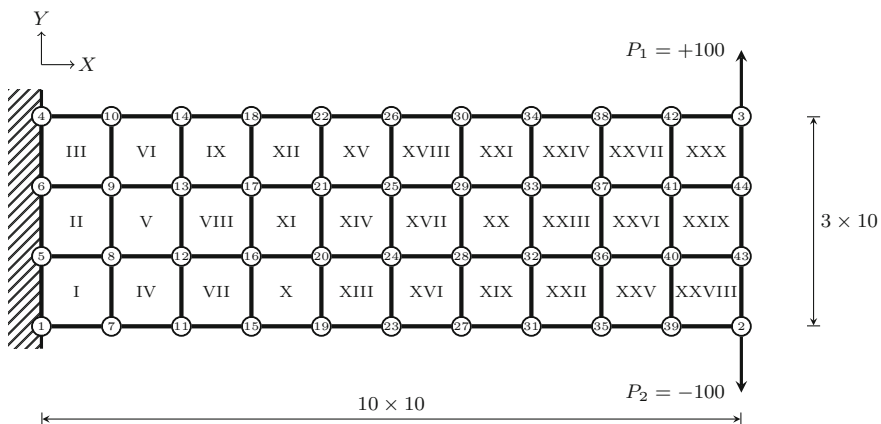
A coordinate system for the material orientation can be defined by executing the following commands:

③ Material Properties▷ New Orientations▷ Coordinate system

The analysis results for the nodes of element VIII are summarized in Table 3.3 and described in terms of the equivalent stress ( $\sigma_{eq}$ ), horizontal displacement ( $u_X$ ) and vertical displacement ( $u_Y$ ).

### 3.2.6 USDATA and UACTIVE

*Example 3.6* This example illustrates the use of the UACTIVE and USDATA subroutine in a static analysis in which a cantilever beam is loaded with two point loads. The beam is modeled by means of 44 nodes and 30 elements of type 3, i.e. 4-node quadrilateral plane stress elements ( $E = 200 \times 10^3$ ,  $\nu = 0.2$  and  $t = 1$ ). The point loads ( $|P_1| = |P_2| = 1000$ ) are assumed to increase gradually in 10 equal increments. In Fig. 3.9, the discretized structure is shown along with the parallel normal loads.



**Fig. 3.9** Discretized model for the USDATA and UACTIVE subroutine example

The nine elements in the second row, i.e. elements XXIX to V, are deactivated one-by-one in increments 2 to 10, respectively. Finally, at the end of the analysis, the beam will look like a double cantilever beam.

The UACTIVE subroutine is used to activate/deactivate the elements during the analysis. It executes both at the beginning and at the end of each increment for every element. This subroutine is automatically activated.

The list of the elements to be deactivated is passed using the USDATA subroutine. The USDATA subroutine is generally used as a variable initializer which is activated by means of the USDATA input file option. This subroutine uses the marc\_usdacm common block to make the variables available among subroutines. By means of this facility, a neat approach will be acquired namely, instead of modifying the code and recompiling it every time, it is possible to edit the input file.

The USDATA subroutine runs twice in each analysis. The first run is executed during the memory allocation of MARC which is called the *pre-reader* phase. The second run, the *real reader* phase, is carried out by MARC while reading the input file; to be more precise, it is done upon encountering the USDATA input file option. At this moment, the reading process of the input file is interrupted and passed to the subroutine. In this way, the subsequent lines can be read by the subroutine as a raw data reader. The raw data can be supplied in multiple text lines. However, to prevent any input file errors, they all must be read by the subroutine, even if they are not used.

The following lines should be added to the model definition section of the input file to activate the USDATA subroutine:

```
1 USDATA9
2 29,26,23,29,17,14,11,8,5
```

The first line indicates the number of variables being passed to the subroutine, i.e. 9 variables. This number is accessible via the nusdat variable of the dimen common block. It is used to allocate the required memory for the data being passed using the marc\_usdacm common block. The second line contains the list of the elements to be deactivated.

The FORTRAN file containing both of the subroutines has the following lines:

```
1 SUBROUTINE USDATA(kin , kou , ic)
2 IMPLICIT NONE
3 INCLUDE 'dimen'
4 ! ** Start of generated type statements **
5 INTEGER ic , kin , kou
6 ! ** End of generated type statements **
7
8 INTEGER , DIMENSION(20) :: elementList
9 INTEGER :: i
10 COMMON /marc_usdacm/ elementList
11
12 IF (ic .EQ. 2) THEN
13 READ (kin ,*) (elementList(i) , i = 1 , nusdat)
14 WRITE (kou ,*) (elementList(i) , i = 1 , nusdat)
15 END IF
16
17 RETURN
18 END
19
20 SUBROUTINE UACTIVE(m,n,mode,irststr , irststn , inc , time , timinc)
```

```

21      IMPLICIT NONE
22      INCLUDE 'dimen'
23      !      ** Start of generated type statements **
24      INTEGER inc, irststn, irststr, m, mode, n
25      REAL*8 time, timinc
26      !      ** End of generated type statements **
27      DIMENSION m(3), mode(3)
28
29      INTEGER, DIMENSION(20) :: elementList
30      INTEGER :: i
31      COMMON /marc_usdadm/ elementList
32
33      mode(2) = 2
34      IF (inc .GT. 1) THEN
35          IF (m(1) .EQ. elementList(inc-1)) THEN
36              mode(1) = -1
37              mode(2) = 0
38              mode(3) = 1
39              irststr = 1
40              irststn = 1
41          END IF
42      END IF
43      RETURN
44      END

```

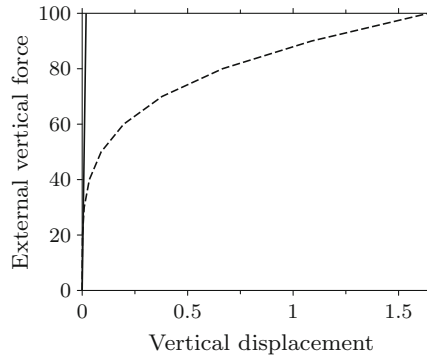
In the USDATA subroutine, the *ic* flag is used to indicate in which phase the subroutine is executed. It is equal to 1 for the pre-reader and 2 for the real reader. The *kin* and *kou* variables are the default input and output file units, respectively. In line 13, an implied-DO structure is used to read the element list from the input file and just to confirm a correct reading process, the same list is printed in the output file.

In the UACTIVE subroutine, the default mode is to keep the status of the element as before ( $mode(2) = 2$ ). Starting from increment 2, the first element of the list is deactivated, i.e. element 29, and in the next increment the next item of the list is deactivated and so on. The deactivated elements are removed from the post-file ( $mode(1) = -1$ ) and their stresses and strains are set to zero upon deactivation ( $irststr = 1$  and  $irststn = 1$ ). The deactivation is done at the beginning of each increment ( $mode(3) = 1$ ).

The same example can be repeated without deactivating the elements, i.e. a typical cantilever beam with two point loads. The results of this analysis along with the one with element deactivation are illustrated in Fig. 3.10. For both cases, the external vertical force, applied to node 3, is plotted against its vertical displacement. The graphs indicate a nonlinear behavior for the case of element deactivation which is accompanied by an extreme displacement whereas a linear behavior is observed for the typical case.

### 3.2.7 SEPFOR and MOTION

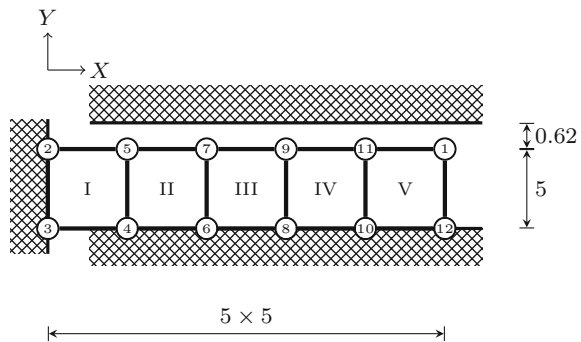
*Example 3.7* This example incorporates both the SEPFOR and the MOTION subroutines in a static contact analysis. In this problem, a cantilever beam is modeled by means of 12 nodes and 5 elements of type 3, i.e. 4-node quadrilateral plane stress



— Without element deactivation    - - - With element deactivation

**Fig. 3.10** Result of incorporating the UACTIVE subroutine in a double cantilever beam

**Fig. 3.11** Discretized model for the SEPFOR and MOTION subroutine example



elements ( $E = 200 \times 10^3$ ,  $\nu = 0.2$  and  $t = 1$ ). The node-to-segment method is used to model the contact problem of the deformable elements and the three rigid boundaries illustrated in Fig. 3.11. The left-hand rigid curve is glued to node 2 and 3 whereas the other two rigid curves are in a touching contact with the elements. An initial touching contact between nodes 4, 6, 8, 10 and 12 and the bottom rigid curve is set at the beginning of the analysis whereas the upper rigid curve has a 0.62 gap with the elements. Also the possibility of a touching interaction is defined between the elements and the upper rigid curve. The elements are defined as a meshed deformable body. Note that no boundary conditions are used to model this problem but instead, sinusoidal rigid body movement is applied to nodes 2 and 3 via the glued contact.

The SEPFOR subroutine is made available only to node-to-segment contact models and it is used to define the normal and tangential separation forces. This subroutine runs for all of the nodes in contact with a body in each increment. Thus, this makes it possible to specify various separation forces per node.

The MOTION subroutine is used to specify the velocity of the rigid bodies during an analysis. It can be used for both 2D and 3D analyses with a slightly different syntax. This subroutine runs at the beginning of each increment and returns the

velocity of the rigid curve/surface along the axes as well as the angular velocity. The 2D version of this subroutine is used in this example to specify the velocity with only a  $Y$ -direction component.

The activation of these two subroutines can be done by checking the corresponding checkboxes in the following dialog box:

③ Jobs▷ Jobs▷ Properties▷ Contact Control▷ Advanced Contact Control

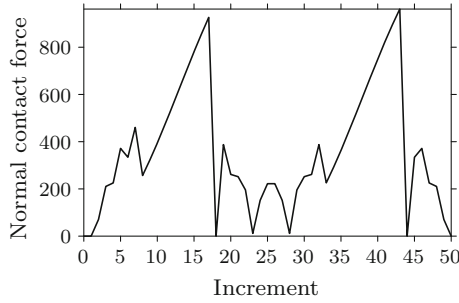
Both of the subroutines can be placed in a single FORTRAN file with the following listing:

```

1  SUBROUTINE MOTION(x , f , v , time , dtime , nsurf , inc )
2  IMPLICIT NONE
3  ! ** Start of generated type statements **
4  REAL*8 dtime , f
5  INTEGER inc , nsurf
6  REAL*8 time , v , x
7  ! ** End of generated type statements **
8  DIMENSION x(*) , v(*) , f(*)
9
10 REAL*8 , PARAMETER :: PI = 3.1415927D0
11
12 IF ( nsurf .EQ. 2 ) THEN
13     v(1) = 0.D0
14     v(2) = 2.D0*SIN(2.0D0*PI*time)
15     v(3) = 0.D0
16 END IF
17
18 RETURN
19 END
20
21 SUBROUTINE SEPFOR(fnorm , ftang , ibody , nnode , inc )
22 IMPLICIT NONE
23 ! ** Start of generated type statements **
24 REAL*8 fnorm , ftang
25 INTEGER ibody , inc , nnode
26 ! ** End of generated type statements **
27
28 IF ( ibody .EQ. 3 ) THEN
29     SELECT CASE ( nnode )
30     CASE ( 6 )
31         fnorm = 300.D0
32         ftang = 0.D0
33     CASE ( 8 )
34         fnorm = 450.D0
35         ftang = 0.D0
36     CASE(10)
37         fnorm = 550.D0
38         ftang = 0.D0
39     CASE(12)
40         fnorm = 240.D0
41         ftang = 0.D0
42     CASE DEFAULT
43         fnorm = 500.D0
44         ftang = 0.D0
45     END SELECT
46 END IF
47
48 RETURN
49 END

```

In line 12, the code determines if the subroutine is running for surface 2, i.e. the left-hand rigid curve. In such a case, a sinusoidal velocity is assigned to the second



**Fig. 3.12** Result of the contact analysis using the SEPFOR and the MOTION subroutines

component of the velocity vector. In line 33, a multiple selection condition is examined and for the in-contact lower nodes of the elements, i.e. nodes 6, 8, 10 and 12, normal separation forces 300, 450, 550 and 240 are assigned, respectively. A normal separation force ( $f_{norm} = 500$ ) is assigned for all other nodes, namely the upper nodes of the elements which will be in contact with the upper rigid curve.

In this example, the initial contact state is selected for the lower rigid curve and 50 increments are used to apply the velocity. Note that for contact problems, it is advised to increase the number of increments for higher accuracy which sometimes helps avoiding penetration of bodies.

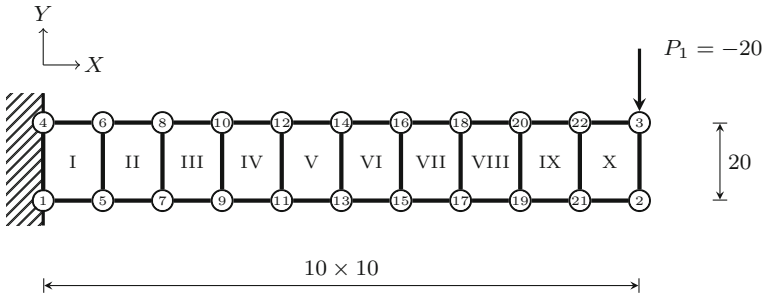
As a result of the analysis, the normal contact force of node 2 versus the number of increments is illustrated in Fig. 3.12.

### 3.2.8 UINSTR

*Example 3.8* This example illustrates the use of the UINSTR subroutine to specify the initial condition of the elements in a static analysis. In this problem, a cantilever beam is modeled by means of 22 nodes and 10 elements of type 3, i.e. 4-node quadrilateral plane stress elements ( $E = 200 \times 10^3$ ,  $\nu = 0.3$  and  $t = 1$ ). A point load ( $P = -20$ ) is applied on node 3 in 10 increments (see Fig. 3.13). The subroutine is used to define the stress state of each element by specifying the stress tensor of the integration points.

The UINSTR subroutine is used to specify the initial stress state of the elements. This subroutine is executed twice for every integration point of each element. In the first run, the stress vector is used to calculate the nodal forces and in the second run the initial stress is calculated. Except in the rigid-plastic analyses, this subroutine runs only in increment zero. The activation of the subroutine is simply done by selecting the proper method for the initial conditions:

- ③ Initial Conditions > Initial Conditions > Properties > Method:User Sub.Uinstr



**Fig. 3.13** Discretized model for the UINSTR subroutine example

The listing of the subroutine is as follows:

```

1  SUBROUTINE UINSTR(s, ndi, nshear, n, nn, kcus, xintp, ncrd, inc, time,
2  &                timeinc)
3
4  IMPLICIT NONE
5
6  ! ** Start of generated type statements **
7  INTEGER inc, kcus, n, ncrd, ndi, nn, nshear
8  REAL*8 s, time, timeinc, xintp
9  ! ** End of generated type statements **
10 DIMENSION s(*), xintp(ncrd), n(2), kcus(2)
11
12 REAL*8, PARAMETER :: TLEN = 100.D0, MAXSTRESS = 50.D0
13
14 s(1) = xintp(1)*MAXSTRESS/TLEN
15 s(2) = -s(1)
16 s(3) = 0.D0
17
18 RETURN
19 END

```

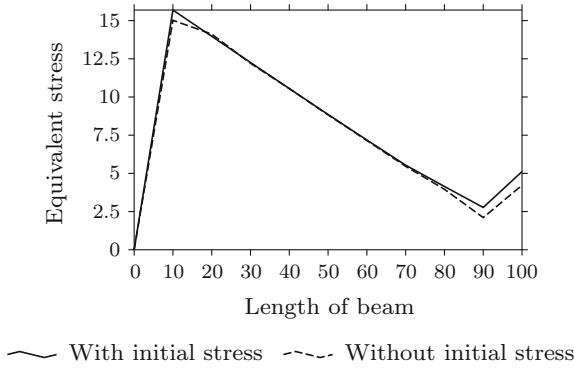
In line 14 of this code, the total length of the beam (TLEN=100) and the maximum stress value (MAXSTRESS) are used to calculate a linear distribution of stress with respect to the X-coordinate of each integration point (xintp(1)). The first and second components of the stress have the same absolute value (s(2)=-s(1)) but the third stress component is zero (s(3)=0.D0). Note that for the element type 3, the first, second and third stress components correspond to  $\sigma_{xx}$ ,  $\sigma_{yy}$  and  $\sigma_{xy}$ , respectively.

The result of the analysis in the last increment is shown in Fig. 3.14 for the length of the beam. The equivalent stress of the upper nodes of the beam are selected for this path plot. In addition, the result of the analysis without the initial stress state is plotted for comparison.

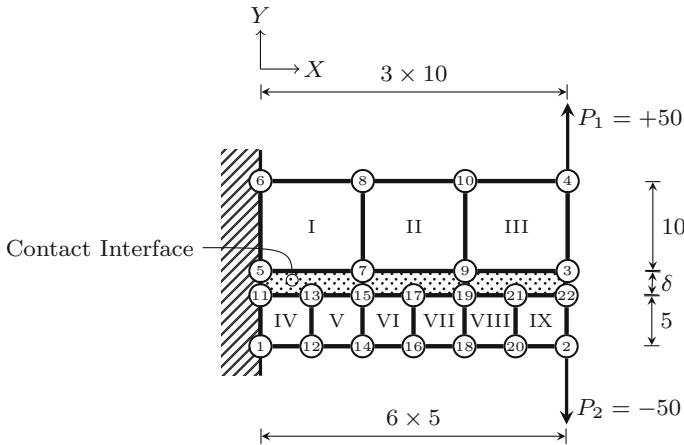
### 3.2.9 UBREAKGLUE

*Example 3.9* This example illustrates the use of the UBREAKGLUE subroutine in a static contact analysis in which a cantilever beam is loaded with a linear load. The beam is modeled by means of 22 nodes and 9 elements of type 52, i.e. 4-node





**Fig. 3.14** Result of incorporating the UINSTR subroutine in the cantilever beam problem (calculated for the upper nodes)



**Fig. 3.15** Discretized model for the UBREAKGLUE subroutine example

quadrilateral plane stress elements ( $E = 200 \times 10^3$ ,  $\nu = 0.3$  and  $t = 1$ ). The point loads ( $|P_1| = |P_2| = 50$ ) are assumed to increase gradually in 20 equal increments. The node-to-segment method is used to model the frictionless contact problem of the two deformable meshed bodies, i.e. the top and the bottom portion of the beam.

The cantilever beam is meshed with incompatible elements, namely the top portion of the beam is discretized by 3 elements of  $10 \times 10$  dimension whereas in the lower portion,  $5 \times 5$  elements are used. In Fig. 3.15, these two portions are illustrated with a contact interface between them. Note that there is no gap between these two parts, and that the contact interface has no dimension ( $\delta = 0$ ). In other words, the contact interface is merely a line starting from node 5, passing through nodes 7 and 9, and finally ending at node 3. With this description, there are four overlapping nodes (double nodes), i.e. nodes 5 and 11, 7 and 15, 9 and 19, and finally 3 and 22.

The node-to-segment contact type is used to carry out the analysis which introduces the concept of *master* and *slave* contact bodies. A body with the slave property has active exterior nodes which are ready to make contact with the master body. On the other hand, the nodes on the master body are not active and instead, its segments are ready to make contact with the nodes of the slave body. Although this is a simplified approach, it contains several disadvantages such as the discontinuity of the stress field along the interface. However, the most evident drawback is the dependency of the solution on the master/slave body selection. Since only the nodes on the slave body can touch the segments of the master body, at the same time nodes on the master body can penetrate the slave body. Therefore, the selection of the master and slave bodies will dramatically change the results of the analysis. In this context, it is advised that the body with a finer mesh to be selected as the slave. Therefore, the lower portion of the beam will be the slave and the upper part will be assigned the master role.

This cantilever beam consists of two meshed (deformable) bodies. Because the node-to-segment contact model determines the master/slave bodies based upon the body number, the selection sequence of the contact bodies is important. MARC considers the body with the lower number to be the slave. Therefore, the lower portion of the beam must be selected first to assign the finely-meshed body as the slave. To define a new deformable contact body, execute the following:

③ Contact ▸ Contact Bodies ▸ New ▸ Meshed (Deformable)

Now, select the 6 elements of the lower portion as cbody1 then define another deformable contact body and assign the remaining elements to cbody2. As a result, the lower portion acts as a slave to the upper master elements. This approach is true for both single-sided and double-sided contacts. However, to ask MARC to look for the body with a finer mesh as the slave and neglect the numbering priority, the Optimize Contact Constraint Equation check-box must be checked in the following dialog box:

③ Jobs ▸ Jobs ▸ Properties ▸ Contact Control ▸ Advanced Contact Control

In this example, the only required interaction is between the two deformable bodies which is a breaking glued contact. To add this interaction to the model, execute the following:

③ Contact ▸ Contact Interactions ▸ New ▸ Meshed (Deformable) vs. Meshed (Deformable)

In the appearing dialog box, select Glued from the Contact Type drop-list. Then click on the Advanced Glue Settings and choose the following items from the drop-lists in the appearing dialog box:

Glue Type: Breaking  
Mode: Normal only

In the same dialog box, enter the values 25 and 1 for the Breaking Normal Stress and the Breaking Normal Exponent, respectively. Now a frictionless breaking glued interaction is defined between two deformable bodies. In this type of contact, nodes

can be released from the glued condition and act as an ordinary contact node which is capable of separation and friction.

In a contact analysis, MARC considers every possible combination of contacts between the defined bodies including self-contact for each one of them. However, in most cases the direction of the contact and the engaged bodies are known. Therefore, it is advised to use a *contact table* to specify only our point of interest and reduce the computational costs. One contact table is enough to define all possible interactions between bodies in this analysis. To do so, first add a new contact table to the model and select the matrix view:

③ Contact > Contact Tables > New > View Mode: Entry Matrix

A matrix view of the possible interactions between two bodies in the model will appear in the right-hand side of the dialog box. No self-contact is required in this example and thus, click on the second column of the first row to assign the interaction between the first and the second body. In the Contact Table Entry Properties dialog box, check the Active check-box and then click on the Contact Interaction button. In the appearing Contact Interaction dialog box, select your previously-defined interaction (interact1).

Note that another benefit of a contact table is overriding the numbering priority of the bodies without changing the numbering of the bodies. In other words, if it is required for the body with the higher number to be the slave, this can be done via the Contact Table Entry Properties dialog box. The default value for the Contact Detection Method is selected which can be changed either to First->Second or Second->First. The former assigns the slave role to the first body which is the default and the latter does the opposite. In our case, the default value and the First->Second will produce the same results.

Note that when applying the glued contact, the initial contact option must be checked, otherwise the two bodies will be separated at the beginning of the analysis. To prevent this, execute the following:

③ Jobs > Jobs > Properties > Contact Control > Initial Contact > Contact Table:table 1

The UBREAKGLUE subroutine is used to redefine the breaking criterion of a breaking glue contact. It is flagged automatically and executed for every node on the slave body acting in a breaking glue interaction. The breaking criterion is the following equation:

$$\left(\frac{\sigma_n}{S_n}\right)^m + \left(\frac{\sigma_t}{S_t}\right)^n > 1. \quad (3.15)$$

This equation considers the interaction between the current normal stress ( $\sigma_n$ ) and the current tangential stress ( $\sigma_t$ ). The normal stress is normalized with respect to the breaking normal stress ( $S_n$ ) and the tangential stress is normalized with respect to the breaking tangential stress ( $S_t$ ). Two exponents are used to give emphasis to the effect of the normal and tangential stress;  $m$  and  $n$ , respectively. In our case, the tangential stress is neglected and the breaking normal exponent is assumed to be 1. The criterion is calculated per node and thus, it is available as a nodal post-processing value. The

normal breaking index ( $t_1 = (\frac{\sigma_n}{S_n})^m$ ), tangential breaking index ( $t_2 = (\frac{\sigma_t}{S_t})^n$ ) and the breaking index ( $t = t_1 + t_2$ ) can be selected to show up in the post file.

In the subroutine, both of the normal and tangential breaking indices will be calculated based on the values already entered via MENTAT. If any modification is necessary, it can be done within the subroutine namely, both of these values act as the input as well as the output of the subroutine. In our example, various normal breaking stresses are considered for the slave node as stated in Table 3.4.

The listing of the subroutine is as follows:

```

1  SUBROUTINE UBREAKGLUE(t1 , t2 , signorm , sigtan , sn , st ,
2  &      expn , expt , info , ibodc , ibodt , inc , time , timeinc )
3
4  IMPLICIT NONE
5
6  REAL*8 t1 , t2 , signorm , sigtan , sn , st , expn , expt , time , timeinc
7  INTEGER info , inc , ibodc , ibodt
8
9  DIMENSION info (*)
10
11 IF (SN .GT. 0.D0 .AND. signorm .GT. 0.D0) THEN
12   SELECT CASE (info (1))
13     CASE (22)
14       sn = 3.D0
15     CASE (21)
16       sn = 5.D0
17     CASE (19)
18       sn = 15.D0
19     CASE (17)
20       sn = 30.D0
21     CASE (15)
22       sn = 35.D0
23     CASE (13)
24       sn = 40.D0
25   END SELECT
26   t1 = (signorm / sn)**expn
27 ELSE
28   t1 = 0.D0
29 END IF
30 RETURN
31 END

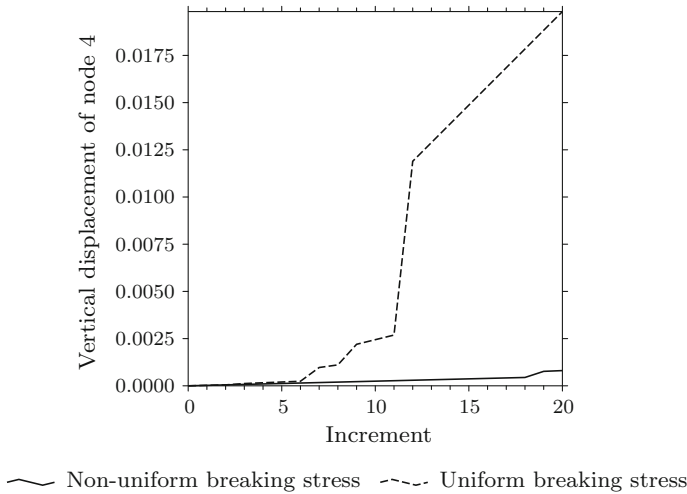
```

In line 11, two conditions are checked: if a normal breaking condition is entered (SN .GT. 0.D0) and if the stress is tensile (signorm .GT. 0.D0). If any of the conditions is not satisfied, the normal breaking index is set to zero (t1 = 0.D0). Otherwise, a SELECT CASE will assign the proper normal breaking stress to each node and the normal breaking index is recalculated in line 26.

The vertical displacement of node 4 versus the increment number is printed for two analyses in Fig. 3.16. The first analysis is conducted with a uniform normal breaking stress and in the second one with the aforementioned non-uniform values. In the former, only one separation has occurred in node 22 and in the latter the separation

**Table 3.4** Assumed non-uniform normal breaking stresses ( $S_n$ ) for nodes

| Node  | 22 | 21 | 19 | 17 | 15 | 13 |
|-------|----|----|----|----|----|----|
| $S_n$ | 3  | 5  | 15 | 30 | 35 | 40 |



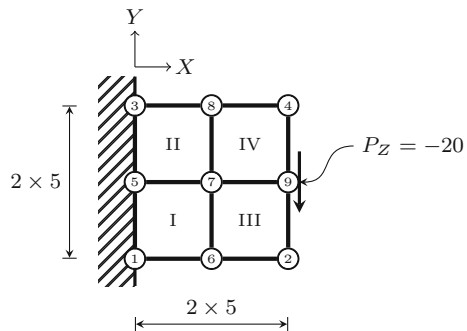
**Fig. 3.16** Result of incorporating the UBREAKGLUE subroutine in the cantilever beam

continues up to node 13. The separations can be detected as the steps in the diagrams e.g. for the uniform case, the break occurs in increment 18.

### 3.2.10 USHELL

*Example 3.10* This example illustrates the use of the USHELL subroutine in a static analysis in which a cantilever plate is loaded with a linear point load. The plate is modeled by means of 9 nodes and 4 shell elements of type 75, i.e. four-node bilinear thick-shell elements ( $E = 200 \times 10^3$ ,  $\nu = 0.3$  and  $t = 0.1$ ). The point load  $P_Z = -1$  is applied along the Z-axis incrementally in 20 fixed steps. The discretized structure is shown in Fig. 3.17 in the XY-plane. However, the analysis is conducted three-dimensionally to investigate the effect of thickness change in shell elements for two

**Fig. 3.17** Discretized model for the USHELL subroutine example (from the top view)



cases: the first one is for the elements with the uniform thickness of 0.1 and the second one is for a thickness starting from 0.1 at the support and linearly increasing to 0.2 at the free end under the following equation:

$$t_{(X)} = 0.1 \times (0.1X + 1). \tag{3.16}$$

The fixed displacement boundary conditions of the support are applied to nodes 1, 3 and 5 to fix the translational degrees of freedom along all three axes and one rotational degree of freedom about the *Y*-axis.

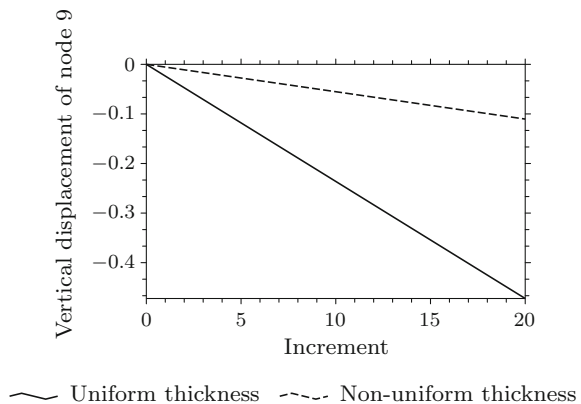
The USHELL subroutine is used to specify the thickness of shell elements at their integration points. The initial uniform thickness of the elements are already specified as geometric property. This value is made available both as an input and an output argument within the subroutine. It is recommended that the non-uniformity of the thickness is kept constant during the analysis. This subroutine is flagged automatically for the shell elements. The following listing is used for the non-uniform thickness case:

```

1      SUBROUTINE USHELL(thick , xintp , ncrd , m , nn)
2          IMPLICIT NONE
3
4      !      ** Start of generated type statements **
5          INTEGER m , ncrd , nn
6          REAL*8 thick , xintp
7      !      ** End of generated type statements **
8          DIMENSION xintp (* ) , m(2)
9
10         REAL*8 , PARAMETER :: TLEN = 5.D0 , t0 = 0.1D0
11
12         thick = (( xintp (1) / TLEN) + 1.D0)*t0
13
14         RETURN
15     END
    
```

In line 12, a linear distribution of the thickness is defined which relates the thickness at the support (*thick* = 0.1D0) to the thickness at the free end (*thick* = 0.2).

**Fig. 3.18** Result of incorporating the USHELL subroutine in a cantilever shell



In this example, two runs were executed: one for the uniform thickness (thick = 0.1D0) and the other one for the linear distribution of the thickness. The results of the analyses are illustrated in Fig. 3.18 which prints the vertical displacement of node 9 versus the loading increments. The maximum displacement for the non-uniform thickness case is less than that of the uniform thickness analysis.

# Chapter 4

## Advanced Examples

**Abstract** This chapter covers more advanced examples which require the incorporation of several Fortran subroutines in Marc/Mentat. Furthermore, many examples apply a multitude of customized subprograms and require advanced finite element knowledge.

### 4.1 Overview

The simulation of a physical phenomena can be done at different levels of complexity. As an increasing number of factors are considered, the more complicated the modeling and the analysis process will be. Obviously, the increased computational cost should be justified by more accurate results provided by the improved model. Such a sophisticated model is not created instantly, but instead, it is developed gradually rising from a very simple model.

In programming, the same concept applies. It is advised to initiate the coding process with the purpose of managing a simple task for a simple model, e.g. obtaining the edges of just one quadrilateral plane element. No matter how trivial the initial task may look, it is best practice to start the programming process from the simplest task for the simplest possible case. Following this, one should build on the example and try to cover more general cases, e.g. develop a subroutine to extract the edges of the elements in a model consisting of several quadrilateral elements. By developing the code further, a quite comprehensive package of subprograms will result which should be able to cover the more general cases, e.g. to extract the edges of various types of elements existing in a model. This step-by-step approach will make the implementation of the most intricate algorithms possible and the testing process manageable.

Following this idea, simple examples were introduced in Chap. 3, and in the current chapter, more complicated cases will be discussed. The complexity of the examples increases with the number of engaged subroutines, the amount of interaction among them and the incorporated knowledge of the finite element method. Dealing with advanced examples provides the user with the opportunity of practicing the fundamentals of the finite element method along with his/her programming skills.

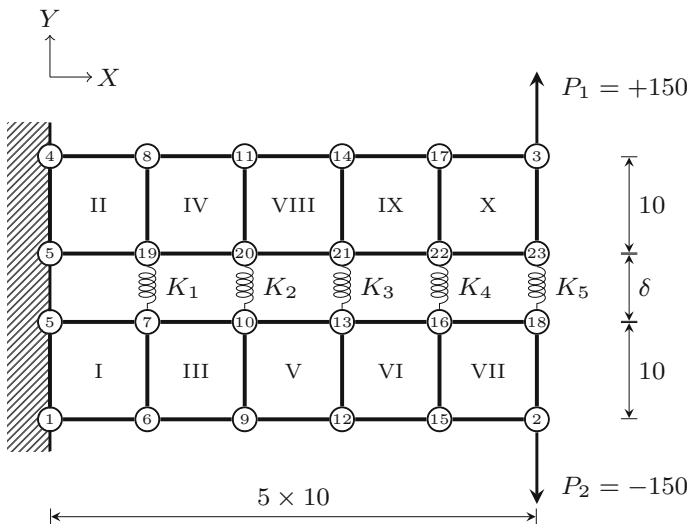


To facilitate some of the common tasks which are usually encountered during this process, some customized subprograms are prepared. These customized subprograms are categorized in three separate modules, namely the MarcTools, FileTools and MiscTools modules. For a detailed description of the included subroutines, one may refer to Chap. 5.

## 4.2 Examples

### 4.2.1 USPRNG and UEDINC

*Example 4.1* This example illustrates the use of the USPRNG and UEDINC subroutine in a static analysis in which a double cantilever beam is loaded with two point loads. The beam is modeled by means of 23 nodes and 10 elements of type 3, i.e. 4-node quadrilateral plane stress elements ( $E = 200 \times 10^3$ ,  $\nu = 0.3$  and  $t = 1$ ). The point loads ( $|P_1| = |P_2| = 150$ ) are assumed to increase gradually in 10 fixed-step increments. In Fig. 4.1, the discretized structure is shown along with the five springs which are linking nodes together. The top half of the cantilever beam is separated from the bottom half. Note that the gap between these two halves is zero ( $\delta = 0$ ), namely all the nodes on the gap are overlapping. Among these nodes is only one common node, i.e. node 5. The boundary conditions are applied as fixed displacements (Displacement  $X = 0$  and Displacement  $Y = 0$ ) on the three leftmost nodes of the model, i.e. nodes 1, 5 and 4. Two analyses are done for this model: one with linear behavior (constant



**Fig. 4.1** Discretized model for the USPRNG subroutine example

stiffness of  $K = 50$ ) and the second one assuming nonlinear behavior of the springs. The nonlinear behavior of the springs is defined by the following equations:

$$F(u) = 22.5u - 2.5u^2, \quad (4.1)$$

$$K(u) = 22.5 - 5u. \quad (4.2)$$

In these equations the force of the spring ( $F(u)$ ) and its stiffness ( $K(u)$ ) are defined as functions of the relative displacement ( $u$ ). The results will be collected by the UEDINC subroutine at each increment.

There are five nonlinear springs ( $K_1$  to  $K_5$ ) which link the second DOF of node couples 7 to 19, 10 to 20, 13 to 21, 16 to 22, and 18 to 23. Linking these nodes can be done by creating a spring via the following command:

③ Links▷ Springs/Dashpots▷ New▷ Fixed DOF

In the appearing dialog box, the DOFs are set to 2 and the start and end nodes are set to the mentioned node couples. Additionally, checking the User Subroutine Usprng check-box will flag the subroutine.

The USPRNG subroutine is used to define nonlinear-elastic foundations (the FOUNDATION option) and springs (SPRINGS option) in both static and dynamic analyses, i.e. the stiffness and the damping effect of springs, respectively. For the case of the springs, it will run once for each spring in every increment. The nonlinear behavior is introduced by using a table and/or the subroutine. However, the latter provides more versatility to the process. In either case, the behavior can be specified in terms of force or stiffness which can be a function of the relative displacement, time, normalized time, increment number or normalized increment number.

If only a table is used to specify the stiffness/force of the spring, the assignment of the table to an existing spring can be done in the following dialog box:

③ Links▷ Springs/Dashpots▷ Properties▷ Properties

In the same dialog box, a value can be entered for either stiffness/force of the spring which acts as a reference value, i.e. a scale factor. In other words, all the values of the table will be calculated automatically and then multiplied by this scale factor.

If only a subroutine is chosen to carry out the task, both the stiffness and the spring force must be calculated in every increment by the user. Calculating only one of the quantities will result in slightly deviated results. The procedure is the same if a table is used in conjunction with the subroutine. However, the scaled tabular values will also be available within the subroutine (via datak variable).

The second subroutine used in this example is UEDINC. This is an automatic subroutine which is executed at the end of each increment and subincrement. It is usually used for collecting the results of the current increment or modifying some common data variables. The only input variable of this subroutine is the number of the current increment and subincrement; no output variables are required. The idea behind using this subroutine is to collect the forces induced in the springs in every

increment and to print them into an external text file. This approach automates the result gathering process, prevents us from having to use History Plot in every analysis we run and is time-efficient, especially in the case of numerous analyses.

In addition to the subroutines, the FileTools user-defined module is used to provide the user with the supplementary subroutines FindFreeUnit and DeleteFile.

The following FORTRAN file is used as the subroutine listing:

```

1  #include 'FileTools.f'
2      MODULE CommonData
3          IMPLICIT NONE
4
5          INTEGER, PARAMETER :: SPRINGS = 5
6          REAL*8, DIMENSION (SPRINGS) :: SpringForces
7      END MODULE CommonData
8
9      SUBROUTINE usprng (ratk , f , datak , u , time , n , nn , nsprng)
10         USE CommonData
11
12         IMPLICIT NONE
13
14         !      ** Start of generated type statements **
15         REAL*8 datak , f
16         INTEGER n , nn , nsprng
17         REAL*8 ratk , time , u
18         !      ** End of generated type statements **
19         DIMENSION ratk (*), datak (*), u (*), time (*), n (*), f (2),
20         * nsprng (*)
21
22         ratk (1) = (-5.D0*u(1)) + 22.5D0
23         f (1) = -2.5D0*(u(1)**2.D0) + 22.5D0*u(1)
24
25         SpringForces (nsprng (1)) = f (1)
26
27         RETURN
28     END
29
30     SUBROUTINE uedinc (inc , incsub)
31         USE CommonData
32         Use FileTools
33
34         IMPLICIT NONE
35         !      ** Start of generated type statements **
36         INTEGER inc , incsub
37         !      ** End of generated type statements **
38         INTEGER :: fileUnit
39         INTEGER :: i
40         CHARACTER (LEN=250), PARAMETER :: fileName = 'Results.txt'
41         LOGICAL :: fileExist , fileEnded , unitConnected
42
43         IF (inc .EQ. 0) THEN
44             CALL FindFreeUnit (fileUnit)
45             CALL DeleteFile (fileName)
46
47             OPEN (UNIT = fileUnit , File = fileName , ACCESS = 'SEQUENTIAL' ,
48             & STATUS = 'NEW' , ACTION = 'READWRITE' , FORM = 'FORMATTED')
49             WRITE (fileUnit , 102)
50             WRITE (fileUnit , 100)
51             WRITE (fileUnit , 102)
52         ELSE
53             WRITE (fileUnit , 101) inc , (SpringForces (i) , i = 1 , SPRINGS)
54             IF (inc .EQ. 10) THEN
55                 WRITE (fileUnit , 102)
56                 CLOSE (UNIT = fileUnit)
57             END IF
58         END IF

```

59  
60  
61  
62  
63

```

RETURN
100 FORMAT ('INC Spring1 Spring2 Spring3 Spring4 Spring5 ')
101 FORMAT (13,1X,5(F7.4,1X))
102 FORMAT (44(' - '))
END
    
```

This listing contains a module (CommonData) and two subroutines (usprng and ued-inc). The module is used to transfer the data between the other two subroutines. It contains the total number of springs (SPRINGS = 5) and an array for the spring forces (SpringForces). In lines 22 and 23, the stiffness of the spring (ratk(1)) and its force (f(1)) are defined using Eqs. (4.1) and (4.2), respectively. In line 25, the calculated forces are assigned to the corresponding element of the SpringForces array. In line 44 a free file unit is selected using the FindFreeUnit subroutine. In the following line, the file Results.txt is deleted using the DeleteFile subroutine (Sect. 5.5). In line 47, the external text file (Results.txt) is prepared for writing (see Sects. 1.9 and 1.10). In lines 49 to 51, the formatted header of the text file is printed which is followed by data lines printed using the code in line 53. The increment number (inc) is followed by an implied-DO to print the spring forces. Lines 54 to 57 print the last line of the file and close its file unit.

The result of the analysis for the nonlinear spring is summarized in the text file with the following content:

| INC | Spring1 | Spring2 | Spring3 | Spring4 | Spring5 |
|-----|---------|---------|---------|---------|---------|
| 1   | 0.0609  | 0.2173  | 0.4459  | 0.7238  | 1.0229  |
| 2   | 0.1218  | 0.4344  | 0.8905  | 1.4436  | 2.0368  |
| 3   | 0.1827  | 0.6513  | 1.3336  | 2.1588  | 3.0413  |
| 4   | 0.2437  | 0.8680  | 1.7752  | 2.8695  | 4.0364  |
| 5   | 0.3047  | 1.0844  | 2.2152  | 3.5757  | 5.0218  |
| 6   | 0.3658  | 1.3006  | 2.6537  | 4.2773  | 5.9978  |
| 7   | 0.4268  | 1.5166  | 3.0907  | 4.9744  | 6.9642  |
| 8   | 0.4879  | 1.7323  | 3.5262  | 5.6669  | 7.9209  |
| 9   | 0.5491  | 1.9477  | 3.9600  | 6.3549  | 8.8681  |
| 10  | 0.6102  | 2.1629  | 4.3924  | 7.0382  | 9.8055  |

The same example can be carried out for the springs with a constant stiffness. The only part requiring modification, is the USPRNG subroutine for which the following listing is used:

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16

```

subroutine usprng ( ratk , f , datak , u , time , n , nn , nsprng )
USE CommonData
IMPLICIT NONE
!
** Start of generated type statements **
REAL*8 datak , f
INTEGER n , nn , nsprng
REAL*8 ratk , time , u
!
** End of generated type statements **
DIMENSION ratk (* ) , datak (* ) , u (* ) , time (* ) , n (* ) , f ( 2 ) ,
* nsprng (* )
REAL*8 :: k = 50.D0

ratk ( 1 ) = k
f ( 1 ) = k*u(1)
    
```

17  
18  
19

```

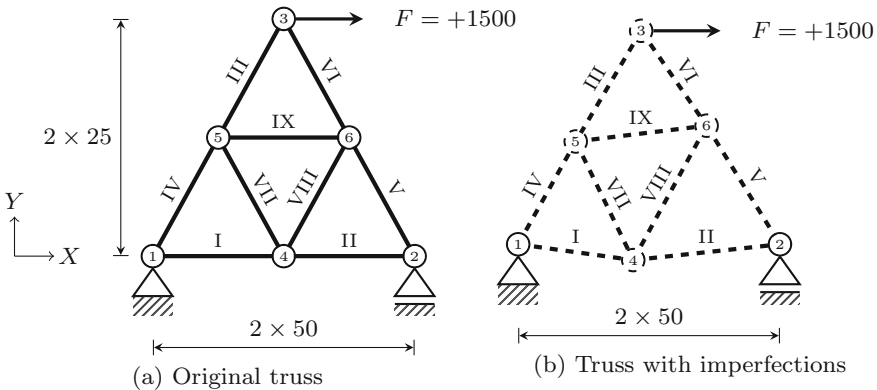
SpringForces(nsprng(1)) = f(1)
RETURN
END
    
```

In line 12, a constant stiffness ( $k = 50.D0$ ) is assigned to every spring and the force is calculated in line 15. The transfer of the force values is done using the same array (SpringForces) which finally results in the following text file as the output:

| INC | Spring1 | Spring2 | Spring3 | Spring4 | Spring5 |
|-----|---------|---------|---------|---------|---------|
| 1   | 0.1159  | 0.4166  | 0.8601  | 1.4048  | 1.9945  |
| 2   | 0.2319  | 0.8332  | 1.7203  | 2.8095  | 3.9891  |
| 3   | 0.3478  | 1.2499  | 2.5804  | 4.2143  | 5.9836  |
| 4   | 0.4637  | 1.6665  | 3.4405  | 5.6191  | 7.9781  |
| 5   | 0.5797  | 2.0831  | 4.3007  | 7.0239  | 9.9727  |
| 6   | 0.6956  | 2.4997  | 5.1608  | 8.4286  | 11.9672 |
| 7   | 0.8115  | 2.9164  | 6.0209  | 9.8334  | 13.9617 |
| 8   | 0.9275  | 3.3330  | 6.8811  | 11.2382 | 15.9563 |
| 9   | 1.0434  | 3.7496  | 7.7412  | 12.6430 | 17.9508 |
| 10  | 1.1593  | 4.1662  | 8.6013  | 14.0477 | 19.9453 |

### 4.2.2 UFXORD, UEDINC and UBGINC

*Example 4.2* This example illustrates the use of the UFXORD, UEDINC and UBGINC subroutines in a static analysis of a 2D truss structure which is modeled by means of 6 nodes and 9 elements of type 9, i.e. a two-node truss element ( $E = 200 \times 10^3$  and  $A = 20$ ). In Fig. 4.2b, the truss is shown with a horizontal point load ( $F = +1500$ ) which is gradually applied on the top node in 10 fixed increments. A pinned support on the left and a roller support on the right-hand side provide the truss with the required constraints. Considering manufacturing imperfections, a slightly-modified version of the structure will be analysed. The required modifications are applied by



**Fig. 4.2** Truss structure in the original and modified configurations

randomly relocating the nodes in a given range. In addition, the whole time of the analysis is measured starting from increment zero to the end of increment 10.

The UFXORD subroutine is used to generate, modify or expand the nodal coordinates of the mesh. The COORDINATES input file option creates the list of nodes followed by their coordinates. Therefore, any calls to the subroutine must be made after the COORDINATES option. The activation of the subroutine is done as soon as encountering the UFXORD input file option. It is also possible to invoke multiple calls to the subroutine by using this option several times. The subroutine is called for all the nodes which are introduced to the UFXORD option. To do so, the list of the nodes must follow this option in the input file. For this example, the subroutine must be executed for all nodes except for the support ones (i.e. nodes 1 and 2). The following lines must be added to the model definition section of the input file:

```
1 UFXORD
2 3 to 6
```

Note that for more sophisticated geometries, the UFXORD subroutine can be used in conjunction with the FOXRD input file option (see [24, 25]).

With a similar structure to the UEDINC subroutine which runs at the end of each increment (Sect. 4.2.1), the UBGINC subroutine is called automatically at the beginning of each increment. It is usually used to set up the initial values of the common variables or to connect a file to a file unit in the beginning of the analysis.

In addition to these three subroutines, the MiscTools module (Sect. 5.6) and the FileTools module (Sect. 5.5) are included in the FORTRAN file to carry out the additional tasks, i.e. generating the random numbers (GetRandNum function), calculating the duration of the analysis (the PrintElapsedTime subroutine), and handling routine file-related tasks (the FindFreeUnit and the AutoFilename subroutines). The full listing is as follows:

```
1 #include 'MiscTools.f'
2 #include 'FileTools.f'
3
4 SUBROUTINE UFXORD(xord,ncrd,n)
5 USE MiscTools
6 USE FileTools
7
8 IMPLICIT NONE
9
10 ! ** Start of generated type statements **
11 INTEGER n, ncrd
12 REAL*8 xord
13 ! ** End of generated type statements **
14
15 DIMENSION xord(ncrd)
16
17 INTEGER :: fileUnit
18 COMMON /CommonData/ fileUnit
19 SAVE /CommonData/
20
21 CHARACTER (LEN=20) :: fileName = 'MyResults'
22 REAL*8, PARAMETER :: MAXTOLERANCE = 1.D0
23 REAL*8 :: randNum
24 INTEGER :: i
25 LOGICAL :: firstRun = .TRUE.
26
27 100 FORMAT ('NODE X-COORD Y-COORD')
```

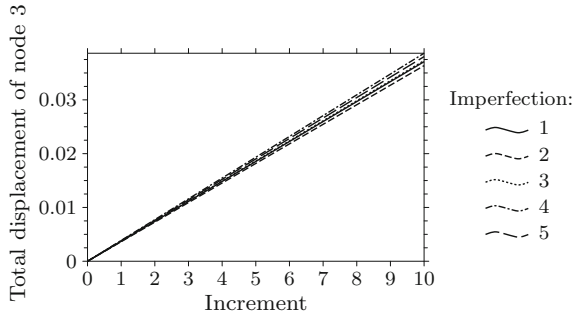
```

28 101 FORMAT (14,1X,2(F7.4,1X))
29 102 FORMAT (21('- '))
30
31     IF (firstRun .EQV. .TRUE.) THEN
32         firstRun = .FALSE.
33         CALL FindFreeUnit (fileUnit)
34         CALL AutoFilename (fileName)
35
36         OPEN (UNIT = fileUnit , File = fileName , ACCESS = 'SEQUENTIAL' ,
37 &           STATUS = 'NEW' , ACTION = 'READWRITE' , FORM = 'FORMATTED')
38         WRITE (fileUnit , 102)
39         WRITE (fileUnit , 100)
40         WRITE (fileUnit , 102)
41     END IF
42     WRITE (fileUnit ,101) n , (xord(i) , i = 1 , 2)
43     DO i = 1 , 2
44         randNum = GetRandNum()
45         xord(i) = xord(i) + (MAXTOLERANCE * randNum)
46     END DO
47     WRITE (fileUnit ,101) n , (xord(i) , i = 1 , 2)
48     RETURN
49 END
50
51 SUBROUTINE UBGINC (inc ,incsub)
52     USE MiscTools
53
54     IMPLICIT NONE
55     ! ** Start of generated type statements **
56     INTEGER inc , incsub
57     ! ** End of generated type statements **
58
59     IF (inc .EQ. 0) CALL PrintElapsedTime ()
60
61     RETURN
62 END
63
64 SUBROUTINE uedinc(inc ,incsub)
65     USE MiscTools
66
67     IMPLICIT NONE
68     ! ** Start of generated type statements **
69     INTEGER inc , incsub , i
70     ! ** End of generated type statements **
71
72     INTEGER :: fileUnit
73     COMMON /CommonData/ fileUnit
74     SAVE /CommonData/
75
76     IF (inc .EQ. 10) THEN
77         CALL PrintElapsedTime ()
78         CLOSE (UNIT = fileUnit)
79     END IF
80     RETURN
81 END

```

In line 17 to 19 of this listing, a common block is defined containing the file unit which will be required at the end of the analysis in lines 72 to 74 of the UEDINC subroutine (Sect. 1.8.3). In lines 21 to 25, the required variables in the program are declared. Lines 31 to 41 prepare a new file to be written upon the first run of the UFXORD subroutine. Note that because the AutoFilename subroutine is used, the previously-created result files will be preserved. Line 42 writes the original coordinates to the custom file and after updating the coordinates by random numbers, the modified coordinates are written in line 47. The UBGINC subroutine is used to start

**Fig. 4.3** Result of incorporating UFXORD subroutine in a cantilever shell



the stopwatch at increment zero (line 59) whereas the UEDINC subroutine is used to stop the stopwatch (line 77) and to close the file unit (line 78). A sample of the generated result file is as follows:

```

-----
NODE X-COORD Y-COORD
-----
 3 50.0000 50.0000
 3 50.3702 49.7619
 4 50.0000 0.0000
 4 49.6244 -0.2325
 5 25.0000 25.0000
 5 25.7875 24.4215
 6 75.0000 25.0000
 6 74.8862 25.5726
-----
Elapsed time is 0.031 second(s)
    
```

The results of the analyses for the ideal truss (no imperfections) along with four random imperfections are illustrated in Fig. 4.3. In this graph, the total displacement of node 3 is printed at each increment. It is notable that the displacement of the imperfect structure fluctuates about the values of the perfect structure.

### 4.2.3 USPLIT\_MESH

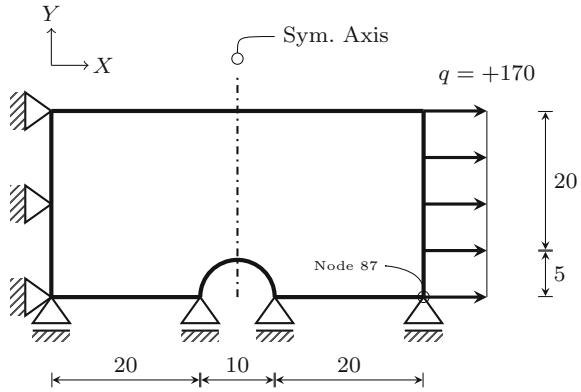
*Example 4.3* This example demonstrates the use of the USPLIT\_MESH subroutine, along with the UBGINC and UEDINC subroutines to model the behavior of a plate (50 × 25) with a hole (r = 5) which is illustrated in Fig. 4.4. The Automesh capability of MENTAT is used to discretize the model by quadrilateral plane elements. To do this, the following commands are executed to result in 96 elements and 115 nodes:

- ③ Geometry & Automesh > Planer > Divisions
- ⑧ 10 10
- Quadrilaterals (Overlay) > Quad Mesh!

The elements are 4-node quadrilateral plane stress elements ( $E = 200 \times 10^3$ ,  $\nu = 0.3$  and  $t = 5$ ). A distributed load ( $q = +170$ ) is applied to the right-hand edge



**Fig. 4.4** Geometry of the plane stress plate with a hole for the USPLIT\_MESH subroutine example



of the plate and is increased gradually in 10 fixed-step increments. The material behaves linearly but as the splitting criterion, the von Mises equivalent stress will be compared to the yield stress ( $\sigma_y = 210$ ).

The USPLIT\_MESH subroutine is used to split a mesh along an edge or a face. The split results in additional edges and nodes which are added automatically by MARC. For a 2D case, a list of nodes and/or a list of edges should be defined as an output of the subroutine. If only a list of nodes is introduced, the corresponding edge list will be created automatically. If only a list of edges is used, all the nodes of the edges are eligible of splitting. If only some of the nodes of an edge should be split, both the node and edge list must be filled. Note that this subroutine is flagged automatically.

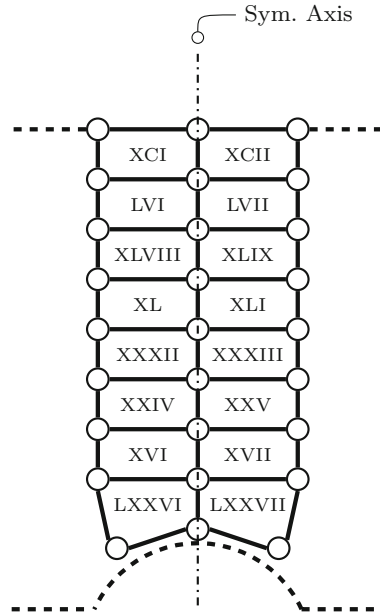
This subroutine can be executed multiple times during an analysis and thus, the icall variable is used to determine the stage of the analysis. The very first run is in increment zero (icall=1). It also runs during the recycling of the current increment (icall=2) and at the end of each increment (icall=3). In the current example, only runs at the end of increments are handled by the code. As mentioned earlier, the USPLIT\_MESH subroutine requires a list of edges to be split. In most cases, it is not possible to introduce an exact path for the split. Therefore, in this example a set is defined, named ElementList, consisting of nominated elements which have potential edges to split. This can be done by creating a new set of elements and selecting the two column of elements which are located above the hole (15 elements in total). The selecting process of the members of the set can be done by executing the following:

- ① Select > Set Control > Elements > New Set
- ⑧ ElementList

Then, the two central columns of elements above the hole must be selected (see Fig. 4.5).

In addition to the three mentioned subroutines, the MarcTools module (Sect. 5.4) is included in the FORTRAN file to carry out the additional tasks, i.e. extracting the members of a set (ExtractSetItems subroutine), extracting the edges of the elements (the ExtractElementEdges subroutine), removing multiple items of an array (the DelRepeated and PutSmallFirst subroutines), and removing the exterior edges

**Fig. 4.5** Selected elements located above the hole of the plate (ElementList set)



(the RemoveExterior subroutine). In addition, the stress on the edges is calculated using the CalcStressEdge subroutine which itself uses other subroutines of the module. The CommonData module is also used to share the common data between the subroutines. The full FORTRAN listing is as follows:

```

1 #INCLUDE 'MarcTools.f'
2
3     MODULE CommonData
4     IMPLICIT NONE
5
6     CHARACTER*32, PARAMETER :: SETNAME = 'ElementList'
7     REAL*8, PARAMETER :: yStress = 210.D0
8
9     INTEGER, ALLOCATABLE :: edgeList(:, :)
10    INTEGER :: edgeCount
11
12    REAL*8, ALLOCATABLE :: edgeStress(:)
13    LOGICAL, ALLOCATABLE :: yMask(:)
14 END MODULE CommonData
15
16 SUBROUTINE ubginc(ubInc, ubIncsub)
17
18     USE CommonData
19     USE MarcTools
20
21     IMPLICIT NONE
22
23     ! ** Start of generated type statements **
24     INTEGER ubInc, ubIncsub
25     ! ** End of generated type statements **
26
27     INTEGER, PARAMETER :: MAXEDGE = 8
28

```

```

29 INTEGER, ALLOCATABLE, DIMENSION(:) :: e1Lst
30 INTEGER :: e1Num
31
32 INTEGER, ALLOCATABLE, DIMENSION(:,:) :: curEdLst
33 INTEGER :: curEl, curEdNum
34
35 INTEGER, ALLOCATABLE, DIMENSION(:,:) :: orgEdLst
36 INTEGER :: orgEdNum
37
38 INTEGER, ALLOCATABLE, DIMENSION(:,:) :: refEdLst
39 INTEGER :: nRefEdLst
40
41 INTEGER :: i, j, k
42
43 IF (ub1nc .EQ. 0) THEN
44
45     CALL ExtractSetItemLst (SETNAME, e1Lst, e1Num)
46
47     IF (e1Num .GT. 0) THEN
48         ALLOCATE (orgEdLst(2, e1Num*MAXEDGE))
49         orgEdNum = 0
50
51         DO i = 1, e1Num
52             curEl = e1Lst(i)
53             CALL ExtractElmEdgeLst (curEl, curEdLst, curEdNum)
54             DO j = 1, curEdNum
55                 DO k = 1, 2
56                     orgEdLst(k, orgEdNum + j) = curEdLst(k, j)
57                 END DO
58             END DO
59             orgEdNum = orgEdNum + curEdNum
60         END DO
61
62         CALL PutSmallFirst (orgEdLst, orgEdNum)
63         CALL DelRepeated2D (orgEdLst, orgEdNum, refEdLst, nRefEdLst)
64
65         CALL DelElmFreeEdge (refEdLst, nRefEdLst, edgeList)
66         edgeCount = size(edgeList, 2)
67
68         Allocate (edgeStress(edgeCount))
69         Allocate (yMask(edgeCount))
70         yMask = .FALSE.
71         edgeStress = 0.D0
72     ELSE
73         CALL QUIT(1234)
74     END IF
75 END IF
76 RETURN
77 END
78
79 SUBROUTINE usplit_mesh (icall, nodelist, nlist, iedgelist, nedgelist,
80 $   ifacelist, nfacelist, inc, time, timeinc)
81
82     USE CommonData
83     IMPLICIT NONE
84
85     ! ** Start of generated type statements **
86     INTEGER nodelist, nlist, iedgelist, nedgelist, ifacelist, nfacelist
87     INTEGER icall, inc
88     REAL*8 time, timeinc
89     DIMENSION nodelist(*), iedgelist(2,*), ifacelist(4,*)
90     ! ** End of generated type statements **
91
92     INTEGER :: i, yEdNum
93     INTEGER, ALLOCATABLE, DIMENSION(:) :: yIndex
94
95     IF (icall .EQ. 3) THEN

```

```

96      yMask = [(edgeStress(i) .GT. yStress , i=1,edgeCount)]
97      yEdNum = COUNT(yMask)
98      ALLOCATE (yIndex ,SOURCE=PACK([(i , i=1,edgeCount)] ,yMask))
99
100     nEdgeList = yEdNum
101     iEdgeList(:,1:yEdNum) = edgeList(:,yIndex)
102     END IF
103     RETURN
104     END
105
106     SUBROUTINE uedinc(inc ,incsub)
107
108     USE CommonData
109     USE MarcTools , ONLY: GetElmEdgeVal
110     IMPLICIT NONE
111     ! ** Start of generated type statements **
112     INTEGER inc , incsub
113     ! ** End of generated type statements **
114
115     INTEGER :: i
116
117     IF (inc .GT. 0) THEN
118     DO i = 1, edgeCount
119         edgeStress(i) =
120     &      GetElmEdgeVal (edgeList(1,i) ,edgeList(2,i) ,17,2)
121         WRITE(6,*) edgeList(1,i) ,'- ' ,edgeList(2,i)
122         WRITE(6,*) 'stress ' , edgeStress(i)
123     END DO
124     END IF
125     RETURN
126     END

```

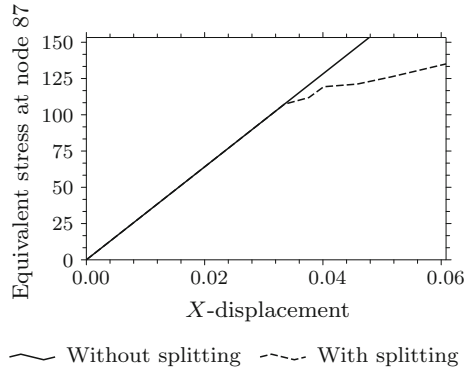
In lines 6 and 7, the constant parameters SETNAME and YSTRESS are used to specify the name of the set and the yield stress, respectively. In lines 9 and 10, the allocatable 2D array edgeList and the integer edgeCount are declared to hold the number of nodes which make up the edges and the number of total edges, respectively. In the next two lines, the edgeStress array holds the calculated stress of the edges and the yMask array holds the yield state of each edge.

In the UBGINC subroutine, the code only runs in increment zero (line 43) to extract the edgeList array and initialize other common variables. The pseudo-code can be stated as the following steps:

- extract the element numbers using the set name (line 45),
- extract all the edges of each element (line 48 to 59),
- remove the recurring edges (line 62 and 63), and
- remove the exterior edges (line 65).

The result of these steps is a list of only the interior edges. In line 66, the intrinsic function SIZE is used to extract the second dimension of the edgeList array, i.e. the final number of edges. This value is used in lines 55 and 56 to allocate the dynamic arrays edgeStress and yMask in the common module which is followed by a couple of lines to initialize them with default values zero and .FALSE., respectively. Note that the ExtractSetItemLst returns zero if no items could be found in the specified set and for this case, the QUIT utility subroutine is used to return the exit code 1234 and to terminate the job.

The UEDINC subroutine is used to calculate the stress on the edges at the end of each increment (lines 117 to 125). The GetElmEdgeVal function is used to calculate



**Fig. 4.6** Result of using the USPLIT\_MESH subroutine in the plate

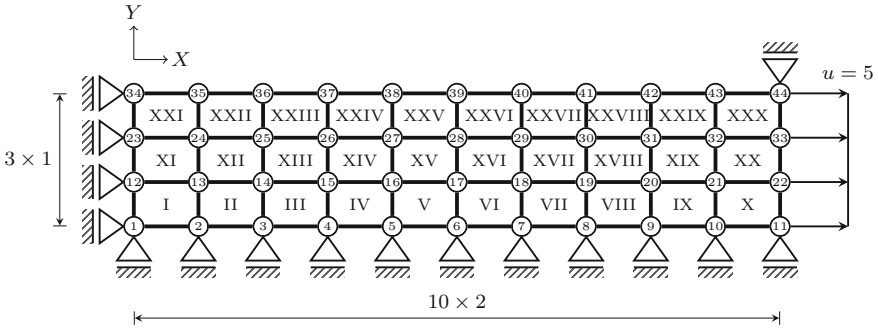
the von Mises stress on an edge which is assumed to be equal to the mean stress of its connecting nodes. The stress on the node is the average stress of the neighboring integration points (Sect. 5.4).

The analysis is done twice; with the subroutine to model the split of the mesh and without the subroutine for a linear material. The results of both analyses are demonstrated in Fig. 4.6 for the right-hand bottom node (node 87 in Fig. 4.4). The equivalent von Mises stress is printed versus the horizontal displacement of the node. In the final increments, a lower stiffness is measured for the split mesh in the mentioned direction.

#### 4.2.4 IMPD and NODVAR

*Example 4.4* This example illustrates the use of the NODVAR utility subroutine in a static analysis in which a tensile test is conducted on a round rod sample ( $r = 3$ ) under displacement control. The sample is modeled by means of 44 nodes and 30 elements of type 10, i.e. 4-node quadrilateral axisymmetric elements ( $E = 73 \times 10^3$ ,  $\nu = 0.34$ , and  $t = 1$ ). The material is assumed to be linear-elastic with isotropic hardening in the plastic material range. The displacement-controlled test is simulated by applying the gradually increasing vertical displacement along the rod axis in 100 increments. The discretized structure is shown in Fig. 4.7.

The material is assumed to be elastic-plastic with isotropic hardening which is defined by the data points provided in Table. 4.1. Generally, it is possible in MARC to use either the engineering strain/stress or the true strain/stress uniaxial test data. This depends on the options which are selected in the input file. For instance, the FINITE, LARGE DISP, LARGE STRAIN and UPDATE options require the true strain/stress data (see [25]). In the current example, the necking phenomenon is simulated. Therefore, to consider the geometric nonlinearity, the LARGE STRAIN option is activated by modifying the properties of the current job:



**Fig. 4.7** Discretized model for the IMPD subroutine example

**Table 4.1** Work hardening data for the assumed plastic behavior of the material

| True plastic strain | True stress |
|---------------------|-------------|
| 0.0000              | 190.0       |
| 0.0050              | 198.0       |
| 0.0150              | 209.0       |
| 0.0300              | 225.0       |
| 0.0500              | 237.6       |
| 0.0700              | 242.4       |
| 0.1500              | 256.0       |
| 0.2416              | 266.9       |
| 0.3220              | 273.4       |
| 0.3839              | 276.6       |
| 0.5112              | 281.0       |
| 0.6499              | 283.8       |
| 0.7753              | 285.6       |
| 0.9028              | 286.8       |
| 0.9959              | 287.6       |
| 1.0491              | 287.9       |
| 1.5200              | 289.2       |
| 2.5600              | 291.0       |
| 4.0000              | 292.4       |

③ Jobs > Jobs > Properties > Analysis Options > Nonlinear Procedure: Large Strain

Next, at the same dialog to activate the updated Lagrangian procedure, execute the following:

Advanced Options > Large Strain: Updated Lagrange

The aim of this example is to calculate the true and engineering values of the stress and strain in the specimen. The engineering stress ( $\sigma$ ) is calculated using the following equation:

$$\sigma = \frac{F}{A_0}. \quad (4.3)$$

The total force of the cross section ( $F$ ) is obtained by summing up the reaction forces of the right-hand nodes along the  $X$ -axis. The initial cross sectional area ( $A_0$ ) is calculated using the initial radius of the specimen.

The longitudinal engineering strain ( $\epsilon$ ) is calculated using

$$\epsilon = \frac{\Delta L}{L}, \quad (4.4)$$

where the elongation of the specimen ( $\Delta L$ ) is divided by its initial length ( $L$ ). The elongation is calculated by subtracting the initial length of the specimen from the distance between node 44 and node 34, i.e. the instantaneous length of the specimen.

Since larger deformations and strains are considered in this example, the true stress ( $\sigma_{tr}$ ) and the true strain ( $\epsilon_{tr}$ ) are more commonly used in such cases. Additionally, the true plastic strain ( $\epsilon_{tr}^{pl}$ ) is a useful value to be calculated [30]. The true stress is calculated using:

$$\sigma_{tr} = \frac{F}{A}, \quad (4.5)$$

where the total force of the cross section ( $F$ ) is divided by the actual area of the specimen ( $A$ ). Since in this example we are interested in the necking zone, the actual cross sectional area is calculated using the necking radius, i.e. the distance between nodes 1 and 34.

The true strain is simply calculated using the engineering strain:

$$\sigma_{tr} = \ln(1 + \epsilon). \quad (4.6)$$

Finally, the true plastic strain is obtained using:

$$\epsilon_{tr}^{pl} = \ln(1 + \epsilon) - \frac{\sigma_{tr}}{E}, \quad (4.7)$$

where the elastic strain part ( $\frac{\sigma_{tr}}{E}$ ) is subtracted from the total true strain ( $E$  is the modulus of elasticity).

The NODVAR utility subroutine is used to extract the nodal values in a model. It is available in any user subroutine of MARC. However, depending on the subroutine in which it is executed, the returned values may be for the previous increment. Therefore, it is advised to use NODVAR at the end of each increment, i.e. within the UEDINC subroutine, when the solution is converged. This way, one ensures that the values are the final values of the current increment. Then, the nodal values can be shared between other subroutines in the subsequent increments. Alternatively, if this utility is called within the IMPD subroutine, the values are those at the end of the current increment [26].

Also note that only the requested nodal post codes are accessible by the NODVAR subroutine. For more information on the nodal post codes refer to the description of the POST input file option in [24].

In the MarcTools module, the CalcNodVal subroutine is used to extract the nodal values of a node by means of the NODVAR subroutine. The MakeNodValLst subroutine creates a list of those values for a selected number of nodes and the PrintNodValLst subroutine prints the values to a file.

The extraction of the reaction force is done for nodes 11, 22, 33, and 44. For a brief list of nodes/elements, the USDATA subroutine can be used to pass the selected nodes/elements to the subroutine (see Sect. 3.2.6). However, a more complicated problem includes numerous nodes and elements. Therefore, it is advised to put the selected items in a set and extract the members of the set within the subroutine. This has been made possible via the ExtractSetItemLst subroutine of the MarcTools module (Sect. 5.4).

In this example, the right-hand nodes, i.e. nodes 11, 22, 33 and 44, are selected as a set of nodes named ForceNodes and nodes 1 and 34 are selected as a set named NeckNodes. Additionally, nodes 34 and 44 are selected in a set named DistanceNodes.

The UEDINC automatic subroutine is used to collect the data at the end of each increment. The complete listing for this subroutine is as follows:

```

1 #include 'MarcTools.f'
2     SUBROUTINE UEDINC (inc, incsub)
3         USE MarcTools, ONLY : MakeNodValLst, GetDistance, GetNodCoord,
4             & ExtractSetItemLst
5         USE FileTools
6
7         IMPLICIT NONE
8
9         INCLUDE 'matdat'
10
11     ! ** Start of generated type statements **
12     INTEGER inc, incsub
13     ! ** END of generated type statements **
14
15     CHARACTER (*), PARAMETER :: FILENAME = 'result.txt',
16     & FORCE_NOD_SET = 'ForceNodes',
17     & NECK_NOD_SET = 'NeckNodes',
18     & DISTANCE_NOD_SET = 'DistanceNodes'
19     REAL*8, PARAMETER :: PI = 4.D0 * ATAN (1.D0)
20
21     REAL*8, ALLOCATABLE, DIMENSION (:,:) :: reactionForceLst
22     REAL*8, ALLOCATABLE, DIMENSION (:, :) :: reactionForceSumLst
23
24     REAL*8, SAVE :: initialArea, initialRadius, initialLength,
25     & elasticModulus
26
27     REAL*8 :: engStress, trueStress, engStrain, trueStrain,
28     & truePlasticStrain, trueElasticStrain,
29     & currentRadius, currentArea, currentLength
30
31     INTEGER, SAVE, ALLOCATABLE, DIMENSION (:) :: distanceNodLst,
32     & forceNodLst,
33     & neckNodLst
34
35     INTEGER, SAVE :: nDistanceNodLst, nForceNodLst, nNeckNodLst,
36     & fileUnit
37
38     INTEGER :: i, nComp, nReactionForceLst

```



```

39
40 100  FORMAT (109(' -'))
41 200  FORMAT (A3, X, 8(A14, X))
42 300  FORMAT (I3, X, 8(F14.4, X))
43
44      IF (inc .EQ. 0) THEN
45          CALL FindFreeUnit (fileUnit)
46          OPEN (UNIT = fileUnit, File = FILENAME, ACCESS = 'SEQUENTIAL',
47 &           STATUS = 'REPLACE', ACTION = 'WRITE')
48
49          CALL ExtractSetItemLst (DISTANCE_NOD_SET, distanceNodLst,
50 &                               ndistanceNodLst)
51
52          CALL ExtractSetItemLst (FORCE_NOD_SET, forceNodLst,
53 &                               nForceNodLst)
54
55          CALL ExtractSetItemLst (NECK_NOD_SET, neckNodLst, nNeckNodLst)
56
57          initialRadius = GetDistance (GetNodCoord (neckNodLst(1)),
58 &                                     GetNodCoord (neckNodLst(2)))
59
60          initialArea  = PI * (initialRadius ** 2.D0)
61
62          initialLength = GetDistance (GetNodCoord (distanceNodLst(1)),
63 &                                     GetNodCoord (distanceNodLst(2)))
64
65          elasticModulus = et(1)
66
67          WRITE (fileUnit, 100)
68          WRITE (fileUnit, 200) 'Inc', 'React. Force X', 'Eng. Stress',
69 & 'True Stress', 'Eng. Strain', 'True Strain', 'True Pl.Strain',
70 & 'True El.Strain'
71          WRITE (fileUnit, 100)
72      ELSE
73          CALL MakeNodValLst (5, reactionForceLst, nReactionForceLst
74 &                             , nComp, forceNodLst, nForceNodLst)
75
76          IF (.NOT. ALLOCATED (reactionForceSumLst))
77 &           ALLOCATE (reactionForceSumLst(nComp))
78
79          reactionForceSumLst = SUM (reactionForceLst, 1)
80
81          currentRadius = GetDistance (GetNodCoord (neckNodLst(1),2),
82 &                                     GetNodCoord (neckNodLst(2),2))
83
84          currentArea  = PI * (currentRadius ** 2.D0)
85
86          engStress = reactionForceSumLst(1) / initialArea
87          trueStress = reactionForceSumLst(1) / currentArea
88
89          currentLength= GetDistance(GetNodCoord (distanceNodLst(1), 2),
90 &                                     GetNodCoord (distanceNodLst(2), 2))
91
92          engStrain = (currentLength - initialLength) / initialLength
93
94          trueStrain = Log (currentLength / initialLength)
95          trueElasticStrain = trueStress / elasticModulus
96          truePlasticStrain = trueStrain - trueElasticStrain
97
98          WRITE (fileUnit, 300) inc, reactionForceSumLst(1),
99 & engStress, trueStress, engStrain, trueStrain,
100 & truePlasticStrain, trueElasticStrain
101      END IF
102
103      IF (inc .EQ. 10) THEN
104          WRITE (fileUnit, 100)
105          CLOSE (fileUnit)

```

106  
107  
108  
109

```

END IF
RETURN
END SUBROUTINE UEDINC
    
```

In this subroutine, the ONLY option is used with the USE statement to avoid any conflict between the variables of the common blocks with the local variables such as the inc variable (lines 3 and 4). The matdat common block is used to obtain the elastic modulus (lines 9 and 65).

The data declaration part starts with the constant parameters (lines 15 to 19). In this part, the name of the output file (FILENAME) and the name of the sets, e.g. FORCE\_NOD\_SET, are defined. In addition, instead of explicitly specifying the digits of the Pi number ( $\pi = 3.1415 \dots$ ), the PI constant is defined using the ATAN intrinsic function to obtain higher precision (line 19).

The data declaration is finished after declaring various variables (20 to 38). Since the output file is opened for FORMATTED writing, the required format statements are defined in lines 40 to 42.

At the end of increment zero, the output file is connected to a free unit (lines 45 and 46) and the old file is replaced by the current one. The ExtractSetItemLst subroutine is used to extract the items of the defined sets (lines 50 to 56). The initial geometric properties of the specimen are calculated in lines 58 to 64, e.g. the initial radius (initialRadius) and the initial cross sectional area (initialArea). Finally, the header of the table is written to the output file in lines 67 to 71.

At the end of every increment (not for increment zero) the nodal values are printed. First the list of nodal values, i.e. the reactions forces, is obtained using the MakeNodValLst subroutine (lines 73 and 74). Then, the sum of the reaction forces of the selected nodes are calculated along X- and Y-directions (line 79). Next, all the current geometric values are calculated using the acquired coordinates of the nodes (line 81 to 90). These values are used along with initial geometrical values to calculate the stresses and strains (line 92 to 96). All the stresses and strains are printed to the output file in a single line (lines 98 to 100).

At the end of increment 10, the last row of the file is printed and the file unit is closed (lines 103 to 106). The resulting output file will contain the following lines:

| Inc | React. Force X | Eng. Stress | True Stress | Eng. Strain | True Strain | True Pl. Strain | True El. Strain |
|-----|----------------|-------------|-------------|-------------|-------------|-----------------|-----------------|
| 1   | 6089.7454      | 215.3807    | 220.8539    | 0.0250      | 0.0247      | 0.0217          | 0.0030          |
| 2   | 6333.3023      | 223.9947    | 235.8550    | 0.0500      | 0.0488      | 0.0456          | 0.0032          |
| 3   | 6352.1811      | 224.6624    | 248.6905    | 0.0750      | 0.0723      | 0.0689          | 0.0034          |
| 4   | 6134.6940      | 216.9704    | 271.9314    | 0.1001      | 0.0954      | 0.0917          | 0.0037          |
| 5   | 5747.5788      | 203.2790    | 298.9705    | 0.1253      | 0.1181      | 0.1140          | 0.0041          |
| 6   | 5179.8638      | 183.2002    | 331.0129    | 0.1506      | 0.1403      | 0.1358          | 0.0045          |
| 6   | 4862.8925      | 171.9896    | 354.0936    | 0.1634      | 0.1513      | 0.1465          | 0.0049          |
| 7   | 4543.7224      | 160.7013    | 385.4573    | 0.1762      | 0.1623      | 0.1570          | 0.0053          |
| 8   | 3980.5280      | 140.7824    | 470.9491    | 0.2019      | 0.1839      | 0.1775          | 0.0065          |
| 9   | 3544.3844      | 125.3570    | 588.0141    | 0.2277      | 0.2051      | 0.1971          | 0.0081          |
| 10  | 3190.6989      | 112.8479    | 751.9715    | 0.2534      | 0.2258      | 0.2155          | 0.0103          |

*Example 4.5* In the previous example, the customized subroutines of the MarcTools module were used to calculate the nodal values. This method is merely using the NODVAR subroutine in a structured way. The resulting code is compact and easily understood. However, one may prefer to approach the same example from another angle, namely not using the customized subroutines or the NODVAR utility subroutine. The IMPD subroutine provides another way of obtaining the nodal values. Since it is not a utility subroutine, the approach is rather different.

The IMPD subroutine makes several nodal quantities available at the end of every increment. Nodal displacements, coordinates, reaction forces, velocities and accelerations are available within the subroutine as the input arguments. The UDUMP option is used for the activation of the IMPD subroutine. The same option is also used for activating the ELEVAR subroutine. This subroutine is used to obtain various elemental values and it will be discussed in Sect. 4.2.5. However, for the IMPD subroutine a list of nodes, and for the ELEVAR subroutine a list of elements, should be provided. This information is specified in the second block of the UDUMP option whereas the first block holds the option keyword, i.e. UDUMP. In the second block, the selected nodes and elements are specified in terms of the user ID ranges. For the specified nodes, the IMPD subroutine and for the specified elements, the ELEVAR subroutine will be executed at the end of each increment. In our case, it is necessary to run the IMPD subroutine for the right-hand nodes of the mesh, i.e. nodes 11, 22, 33, 34, 44 and 1, but the ELEVAR subroutine is not used. Therefore, the following lines are added to the model definition part of the input file:

```
1 UDUMP
2 1,44,,
```

These lines select the nodes 1 to 44 and by default all the elements of the model for which the corresponding subroutines will be executed. The selected range of the nodes includes the required nodes plus many other ones which is a result of the limitation imposed by the UDUMP option. The exact selection will be done within the subroutine itself using conditional statements. Therefore, sometimes it would be easier to run the IMPD subroutine for all existing nodes and the ELEVAR subroutine for all elements. To do so, execute the following command for the current job:

③ Jobs▷ Jobs▷ Properties▷ Job Results▷ Output File▷ User Subroutines▷ Impd (Nodes) or Elevar/Elevec (Elements)

This command is the equivalent of the UDUMP option followed by an empty line which covers all the elements and nodes of the model.

In this example, the IMPD subroutine is used to collect the nodal data and the UEDINC subroutine to operate on them. Note that even in increment zero, both of these subroutines will run. In each increment, first the IMPD subroutine is executed and at the end of the increment the UEDINC subroutine is executed. The UBGINC subroutine is used to allocate the arrays and prepare the output file for writing. The CommonData module is used to share the data between these two subroutines.

Although the general approach in this example is rather different than that of the previous one, the customized subroutines are also used to facilitate the procedure. The listing of the FORTRAN file is as follows:

```

1 #include 'MarcTools.f'
2   MODULE CommonData
3     CHARACTER (*), PARAMETER :: FILENAME      = 'result.txt',
4     &                               FORCE_NOD_SET = 'ForceNodes',
5     &                               NECK_NOD_SET  = 'NeckNodes',
6     &                               DISTANCE_NOD_SET = 'DistanceNodes'
7
8     REAL*8, PARAMETER :: PI = 4.D0 * ATAN (1.D0)
9
10    REAL*8, ALLOCATABLE, SAVE, DIMENSION(:) :: reactionForceLst
11
12    REAL*8, SAVE :: reactionForceSum, elasticModulus,
13    &            initialArea, initialRadius, initialLength
14
15    REAL*8 :: engStress, trueStress, engStrain, trueStrain,
16    &            truePlasticStrain, trueElasticStrain,
17    &            currentRadius, currentArea, currentLength
18
19    INTEGER, SAVE, ALLOCATABLE, DIMENSION(:) :: distanceNodLst,
20    &                                           forceNodLst,
21    &                                           neckNodLst
22
23    REAL*8, SAVE, ALLOCATABLE, DIMENSION(:,:) :: neckNodCoordLst,
24    &                                           distanceNodCoordLst
25
26    INTEGER, SAVE :: nDistanceNodLst, nForceNodLst, nNeckNodLst,
27    &            fileUnit, nReactionForceLst
28  END MODULE CommonData
29
30  SUBROUTINE UBGINC (inc, incsub)
31    USE MarcTools, ONLY : ncrd, ExtractSetItemLst
32    USE FileTools
33    USE CommonData
34    IMPLICIT NONE
35
36  !   ** Start of generated type statements **
37    INTEGER inc, incsub
38  !   ** End of generated type statements **
39
100  FORMAT (109('-',))
200  FORMAT (A3, X, 8(A14, X))
42
43  IF (inc .EQ. 0) THEN
44    CALL FindFreeUnit (fileUnit)
45    OPEN (UNIT = fileUnit, File = FILENAME, ACCESS = 'SEQUENTIAL',
46    &     STATUS = 'REPLACE', ACTION = 'WRITE')
47
48    CALL ExtractSetItemLst (DISTANCE_NOD_SET, distanceNodLst,
49    &                       nDistanceNodLst)
50    ALLOCATE (distanceNodCoordLst(nDistanceNodLst, 3))
51
52    distanceNodCoordLst = 0.D0
53
54    CALL ExtractSetItemLst (FORCE_NOD_SET, forceNodLst,
55    &                       nForceNodLst)
56
57    ALLOCATE (reactionForceLst (nForceNodLst))
58
59    CALL ExtractSetItemLst (NECK_NOD_SET, neckNodLst, nNeckNodLst)
60    ALLOCATE (neckNodCoordLst(nNeckNodLst, 3))
61    neckNodCoordLst = 0.D0
62
63    WRITE (fileUnit, 100)
64    WRITE (fileUnit, 200) 'Inc', 'React. Force X', 'Eng. Stress',
65    & 'True Stress', 'Eng. Strain', 'True Strain', 'True Pl. Strain',
66    & 'True El. Strain'
67    WRITE (fileUnit, 100)

```

```

68     END IF
69     RETURN
70     END
71
72     SUBROUTINE IMPD (Inode , dd , td , xord , f , v , a , ndeg , ncrd )
73
74         USE MarcTools , ONLY: GetIndex
75         USE CommonData
76
77         IMPLICIT NONE
78
79         ! ** Start of generated type statements **
80         REAL*8 a , dd , f
81         INTEGER Inode , ncrd , ndeg
82         REAL*8 td , v , xord
83         ! ** End of generated type statements **
84         DIMENSION Inode (2)
85         DIMENSION dd (ndeg) , td (ndeg) , xord (ncrd) , f (ndeg) ,
86         *           v (ndeg) , a (ndeg)
87
88         INTEGER :: nodIndex
89
90         NodIndex = GetIndex (distanceNodLst , nDistanceNodLst , Inode (1))
91         IF (NodIndex .NE. 0) THEN
92             distanceNodCoordLst (nodIndex , 1:2) = xord (1:2) + td (1:2)
93         END IF
94
95         NodIndex = GetIndex (neckNodLst , nNeckNodLst , Inode (1))
96         IF (NodIndex .NE. 0) THEN
97             neckNodCoordLst (nodIndex , 1:2) = xord (1:2) + td (1:2)
98         END IF
99
100        NodIndex = GetIndex (forceNodLst , nForceNodLst , Inode (1))
101        IF (NodIndex .NE. 0) THEN
102            reactionForceLst (nodIndex) = f (1)
103        END IF
104        RETURN
105    END
106
107
108    SUBROUTINE UEDINC (inc , incsub)
109        USE MarcTools , ONLY : MakeNodValLst , GetDistance ,
110        & ExtractSetItemLst
111        USE FileTools
112        USE CommonData
113
114        IMPLICIT NONE
115
116        INCLUDE 'matdat'
117
118        ! ** Start of generated type statements **
119        INTEGER inc , incsub
120        ! ** END of generated type statements **
121
122
123    100    FORMAT (109(' -'))
124    300    FORMAT (I3 , X , 8(F14.4 , X))
125
126        IF (inc .EQ. 0) THEN
127            & initialRadius = GetDistance (neckNodCoordLst (1 , 1:3) ,
128            & neckNodCoordLst (2 , 1:3))
129
130            initialArea = PI * (initialRadius ** 2.D0)
131
132            & initialLength = GetDistance (distanceNodCoordLst (1 , :) ,
133            & distanceNodCoordLst (2 , :))
134            elasticModulus = et (1)

```

```

135      ELSE
136          reactionForceSum = SUM ( reactionForceLst)
137
138          &
139          currentRadius = GetDistance (neckNodCoordLst(1, :),
140                                     neckNodCoordLst(2, :))
141
142          currentArea    = PI * (currentRadius ** 2.D0)
143
144          engStress    = reactionForceSum / initialArea
145          trueStress   = reactionForceSum / currentArea
146
147          &
148          currentLength= GetDistance (distanceNodCoordLst(1, :),
149                                     distanceNodCoordLst(2, :))
150
151          engStrain    = (currentLength - initialLength) / initialLength
152
153          trueStrain   = Log (currentLength / initialLength)
154          trueElasticStrain = trueStress / elasticModulus
155          truePlasticStrain = trueStrain - trueElasticStrain
156
157          WRITE (fileUnit , 300) inc , reactionForceSum ,
158          & engStress , trueStress , engStrain , trueStrain ,
159          & truePlasticStrain , trueElasticStrain
160          END IF
161
162          IF (inc .EQ. 10) THEN
163              WRITE (fileUnit , 100)
164              CLOSE (fileUnit)
165          END IF
166
167          RETURN
168      END SUBROUTINE UEDINC

```

The CommonData module holds all the required data for the calculation of the stresses and strains. In addition, the constants are also defined and shared via this module (lines 2 to 28).

The UBGINC subroutine is executed at the beginning of each increment. However, it is only required to carry out some initial constructions at increment zero. At the beginning of this increment, it prepares the output file for writing, extracts the items of the sets and allocates the arrays. For instance, the nodes of the necking zone (neckNodLst) are extracted from the NECK\_NOD\_SET. In addition, the header of the result table is printed in the output file.

The IMPD subroutine collects the nodal values and stores them in the previously allocated arrays. This is done by searching for the current node in the list of nodes which was extracted from the sets. For example, to extract the coordinates for the nodes of the necking zone (neckNodCoordLst), the Inode(1) variable, i.e. the user node ID for which the current subroutine is running, is searched in the neckNodLst. If a match exists, its index is returned by the GetIndex function.

The UEDINC subroutine handles the calculations of the current increment and prints them into the result file. The variables with a constant value during the analysis, e.g. the initial length of the specimen (initialLength), are calculated once at the end of increment zero and saved by the SAVE attribute in the CommonData module. In the remaining increments, the current values are calculated and written in the output file. At the end of the last increment, the last raw is written to the file and the file unit is disconnected (lines 161 to 163).

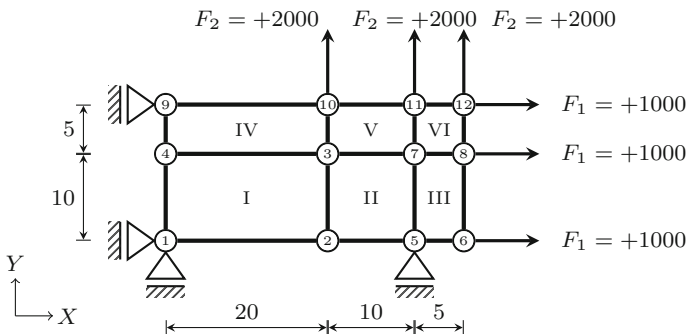
The result of this example is the same as the previous one. However, in comparison with the previous example, the current listing is longer and the number of subroutines is greater. It is worth mentioning that without using the customized subroutines, the listing of this example would be even longer.

### 4.2.5 ELMVAR and ELEVAR

*Example 4.6* This example illustrates the use of the ELMVAR utility subroutine in a static analysis in which a plate is loaded with several horizontal and vertical point loads (Fig. 4.8). The plate is modeled by means of 12 nodes and 6 elements of type 3, i.e. 4-node quadrilateral plane stress elements ( $E = 200 \times 10^3$ ,  $\nu = 0.3$ , and  $t = 1$ ). The material is assumed to be linear-elastic. The horizontal loads ( $F_1 = +1000$ ) are applied to each of the right-hand nodes whereas the vertical loads ( $F_2 = +2000$ ) are applied to the nodes 10, 11, and 12. All loads are applied gradually in 10 steps. The purpose of this example is to obtain the nodal values for some scalar elemental quantities. All the calculated outputs are printed to an external text file.

The ELMVAR utility subroutine is used to extract the elemental quantities at the integration points of the elements. To do so, the required inputs are the element ID, integration point number, internal layer number, and the *element post code*. The standard element post codes are positive values which are used in the POST input file option (see [24]). Note that this subroutine does not support negative post codes which indicate user-defined variables used in conjunction with the PLOTV subroutine.

In addition, depending on the stage of the analysis, the values returned by the ELMVAR subroutine may not be the converged ones. Therefore, it is good practice to call this subroutine at the end of each increment to obtain the final values of the increment. In this example, the UEDINC subroutine is used to collect the results at the end of increments.



**Fig. 4.8** Discretized model for the ELMVAR subroutine example

In MENTAT, three methods can be selected for the calculation of the nodal values of elemental quantities. The Linear method extrapolates the values of the integration points to the nodal points. The Translate method copies the values of the integration points to the neighboring node. The Average method calculates the average of the values obtained from the integration points and copies it to all nodes of the element. In the post-processing stage, the user can switch between these methods via the Element Extrapolation Settings dialog box. For instance, to select the Translate method, one should execute the following command:

③ Results▷ Settings▷ Extrapolation▷ Method: Translate

A node, which is connected to several elements, receives a contribution from the neighboring integration points of those elements. The received values are usually averaged with no regard to the weight of the contribution, i.e. with an *unweighted averaging* method. Namely, any parameters related to the contributor element, such as its size, are not considered in this method. In this example, the standard results of MARC are exported to an external file for later post-processing. In addition, the same results are calculated using an assumed *weighted averaging* method in which the surface area of each element is used as the weight of the quantity.

The developed subroutines of the MarcTools module use the ELMVAR utility subroutine to obtain the initial values. The FileTools module is also used to obtain a free file unit. The listing is as follows:

```

1 #INCLUDE 'MarcTools.f'
2
3     SUBROUTINE UEDINC(uInc , uIncsub)
4         USE MarcTools
5         USE FileTools
6
7         IMPLICIT NONE
8
9         ! ** Start of generated type statements **
10        INTEGER uInc , uIncsub
11        ! ** END of generated type statements **
12
13        CHARACTER(LEN=*), PARAMETER :: fileName = 'result.txt'
14        INTEGER :: i
15        INTEGER, SAVE :: fileUnit
16    100    FORMAT (A20,I2)
17
18
19        IF (uInc .EQ. 1) THEN
20            CALL FindFreeUnit (fileUnit)
21            OPEN (UNIT = fileUnit , File = fileName , ACCESS = 'SEQUENTIAL' ,
22    &           STATUS = 'REPLACE' , ACTION = 'WRITE')
23
24            CALL PrintIPCordLst (fileUnit)
25            CALL PrintNodCoordLst (1, fileUnit)
26            CALL PrintElapsedTime (fileUnit)
27        END IF
28
29        IF (uInc .GE. 1) THEN
30            WRITE (fileUnit ,100) 'Increment No. ', uInc
31
32            CALL PrintNodCoordLst (2, fileUnit)
33
34            DO i = 1, 3
35                WRITE (fileUnit ,100) 'Stress ', i

```



36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50

```

CALL PrintNodValIPLst (10+i, fileUnit)
END DO
DO i = 1, 3
WRITE (fileUnit,100) 'Stress ', i
CALL PrintNodValIPLst (10+i, fileUnit)
END DO
END IF
WRITE(fileUnit,*) 'timinc=', timinc
IF (ulnc .EQ. 10) THEN
CALL PrintElapsedTime (fileUnit)
CLOSE(fileUnit)
END IF
RETURN
END

```

Note that the values obtained from MENTAT might be slightly different from those which MARC produces [25].

In the results.txt file, the stresses are printed for each increment. For instance, the first component of the stress in the nodes in the last increment is as follows:

| Stress 1        |            |              |                    |            |              |                  |
|-----------------|------------|--------------|--------------------|------------|--------------|------------------|
| Element Post 11 |            |              |                    |            |              |                  |
| Node            | Translated | Extrapolated | Unweighted Average | Translated | Extrapolated | Weighted Average |
| 1               | 273.5087   | 314.0950     | 218.0668           | 273.5087   | 314.0950     | 218.0668         |
| 2               | 213.2047   | 209.9672     | 217.6273           | 230.1795   | 239.2612     | 217.7738         |
| 3               | 163.7480   | 143.6618     | 191.1864           | 163.3117   | 136.4540     | 200.0000         |
| 4               | 148.8546   | 118.0265     | 190.9666           | 156.5712   | 124.7791     | 200.0000         |
| 5               | 313.8684   | 391.0555     | 208.4290           | 303.6830   | 371.2765     | 211.3486         |
| 6               | 270.4015   | 322.1804     | 199.6701           | 270.4015   | 322.1804     | 199.6701         |
| 7               | 163.2923   | 139.5057     | 195.7855           | 198.3260   | 197.1006     | 200.0000         |
| 8               | 107.2877   | 39.2968      | 200.1649           | 89.8303    | 9.1805       | 200.0000         |
| 9               | 151.8576   | 143.0665     | 163.8665           | 151.8576   | 143.0665     | 163.8665         |
| 10              | 173.3683   | 179.6807     | 164.7454           | 182.9216   | 196.4419     | 164.4524         |
| 11              | 203.7399   | 218.8186     | 183.1421           | 191.1000   | 201.2002     | 177.3028         |
| 12              | 335.7245   | 434.5987     | 200.6598           | 335.7245   | 434.5987     | 200.6598         |

The elemental values for the first stress component are as follows:

| Stress 1        |          |  |
|-----------------|----------|--|
| Element Post 11 |          |  |
| Element IP      | Value    |  |
| 1 1             | 273.5087 |  |
| 1 2             | 264.1292 |  |
| 1 3             | 172.0044 |  |
| 1 4             | 162.6248 |  |
| 2 1             | 162.2803 |  |
| 2 2             | 283.3122 |  |
| 2 3             | 151.0635 |  |
| 2 4             | 272.0954 |  |
| 3 1             | 344.4247 |  |
| 3 2             | 270.4015 |  |
| 3 3             | 128.9387 |  |
| 3 4             | 54.9155  |  |

|       |   |          |
|-------|---|----------|
| 4     | 1 | 125.7049 |
| 4     | 2 | 175.8754 |
| 4     | 3 | 151.8576 |
| 4     | 4 | 202.0281 |
| ----- |   |          |
| 5     | 1 | 165.4285 |
| 5     | 2 | 186.5401 |
| 5     | 3 | 144.7085 |
| 5     | 4 | 165.8202 |
| ----- |   |          |
| 6     | 1 | 65.5951  |
| 6     | 2 | 159.6599 |
| 6     | 3 | 241.6596 |
| 6     | 4 | 335.7245 |
| ----- |   |          |

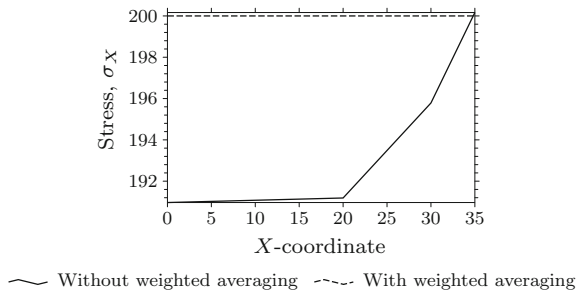
In Fig. 4.9, the result of the analysis is illustrated for the middle row of nodes, i.e. node 4, 3, 11, and 12. In this graph, the stress in the X-direction ( $\sigma_X$ ) is plotted along the length of the plate (its X-coordinate). The stress is calculated using the average weighted and average unweighted nodal values. A notable feature of this graph is that the weighted average method gives a constant stress in each node of the middle row.

*Example 4.7* This example is modeled using the custom subroutines of the MarcTools module which enables the user to obtain the elemental quantities by a few lines of code. This approach does not depend on a specific subroutine and it is accessible from most of the user subroutines. Alternatively, it is possible to use the ELEVVAR subroutine to produce similar results.

The ELEVVAR subroutine provides the user with elemental quantities at the end of every increment. The subroutine is executed for all integration points of the nominated elements. This execution repeats at the end of every increment before the execution of the UEDINC subroutine. Several input arguments are available within the subroutine, e.g. the total strain, total stress, plastic strain, Cauchy stress, equivalent plastic strain, state variables, crack indicators etc.

Since only the data regarding a particular integration point of an element is available at each run, the required data must be collected. For instance, calculating the

**Fig. 4.9** Result of using the ELMVAR subroutine in the last increment



average stress of the integration points is not possible until the execution of the ELEVVAR subroutine for the last integration point of that element. This drawback is magnified if the calculation process requires the elemental values for all the elements, which demands a large memory allocation.

The ELEVVAR subroutine is the counterpart of the NODVAR subroutine. The former provides the elemental values and the latter supplies the nodal ones at the end of each increment. The activation can be done simultaneously for both of them using the UDUMP option (see Sect. 4.2.4). In this example, it is required to run the subroutine for all elements.

The ELEVVAR subroutine is used in conjunction with the UBGINC and UEDINC subroutines. Additionally, the CommonData module is used to share the common data between these subroutines. The FindFreeUnit subroutine of the FileTools module is used to get a free file unit number. The listing is as follows:

```

1 #INCLUDE 'FileTools.f'
2
3     MODULE CommonData
4         CHARACTER(LEN=*), PARAMETER :: fileName = 'result.txt'
5         INTEGER, SAVE :: fileUnit
6
7         REAL*8, DIMENSION(3,4), SAVE :: stressLst
8         REAL*8, DIMENSION(3), SAVE :: meanStressLst
9     END MODULE CommonData
10
11     SUBROUTINE UBGINC (inc, incsub)
12         USE FileTools
13         USE CommonData
14         IMPLICIT NONE
15     ! ** Start of generated type statements **
16     INTEGER inc, incsub
17     ! ** End of generated type statements **
18
19     100    FORMAT (A60,X,I3)
20     200    FORMAT (A7,X,A2,X,6(A15,X))
21     300    FORMAT (61X,A15)
22     400    FORMAT (107('-'))
23
24     IF (inc .EQ. 0) THEN
25         CALL FindFreeUnit (fileUnit)
26         OPEN (UNIT = fileUnit, File = fileName, ACCESS = 'SEQUENTIAL',
27     &        STATUS = 'REPLACE', ACTION = 'WRITE')
28
29         WRITE (fileUnit, 400)
30         WRITE (fileUnit, 300) 'Mean Values'
31
32         WRITE (fileUnit, 200) 'Element', 'IP',
33     &        'Stress(1)', 'Stress(2)', 'Stress(3)',
34     &        'Stress(1)', 'Stress(2)', 'Stress(3)'
35
36         WRITE (fileUnit, 400)
37     END IF
38
39     IF (inc .GT. 0) THEN
40         WRITE (fileUnit, 100) 'Increment ', inc
41         WRITE (fileUnit, 400)
42     END IF
43
44     RETURN
45     END
46
47     SUBROUTINE ELEVVAR(n,nn,kcus,gstran,gstres,stress,pstran,

```

```

48 1 cstran , vstran , cauchy , eplas , equivc , swell , krtyp , prang , dt ,
49 2 gsv , ngens , ngen1 , nstats , nstass , thmstr )
50
51 USE FileTools
52 USE CommonData
53
54 IMPLICIT NONE
55
56 INCLUDE 'concom'
57
58 ! ** Start of generated type statements **
59 REAL*8 cauchy , cstran , dt , eplas , equivc , gstran , gstres , gsv
60 INTEGER kcus , krtyp , n , ngen1 , ngens , nn , nstass , nstats
61 REAL*8 prang , pstran , stress , swell , thmstr , vstran
62
63 DIMENSION gstran (ngens) , gstres (ngens) ,
64 1 stress (ngen1) , pstran (ngen1) , cstran (ngen1) , vstran (ngen1) ,
65 2 cauchy (ngen1) , dt (nstats) , gsv (*)
66 3 , thmstr (*) , prang (3,3) , krtyp (4) , kcus (2)
67 ! ** End of generated type statements **
68
69 INTEGER :: i
70
71 300 FORMAT (I7 ,X ,I2 ,X ,3(F15.4 ,X))
72 400 FORMAT (107(' -'))
73 500 FORMAT (I7 ,X ,I2 ,X ,6(F15.4 ,X))
74
75 IF (inc .GT. 0) THEN
76 DO i = 1 , 3
77 stressLst(i ,nn) = stress(i)
78 END DO
79
80 IF (nn .EQ. 4) THEN
81 DO i = 1 , 3
82 meanStressLst = Sum (stressLst , 2) / 4.D0
83 END DO
84
85 WRITE (fileUnit ,500) n , nn , stress(1) , stress(2) , stress(3) ,
86 & meanStressLst(1) , meanStressLst(2) , meanStressLst(3)
87 WRITE (fileUnit , 400)
88 ELSE
89 WRITE (fileUnit ,300) n , nn , stress(1) , stress(2) , stress(3)
90 END IF
91
92
93 END IF
94 RETURN
95 END
96
97 SUBROUTINE UEDINC (inc ,incsub)
98
99 USE CommonData
100
101 IMPLICIT NONE
102
103 ! ** Start of generated type statements **
104 INTEGER inc , incsub
105 ! ** End of generated type statements **
106
107 IF (inc .EQ. 10) THEN
108 CLOSE (fileUnit)
109 END IF
110
111 RETURN
112 END

```

In the CommonData module, two arrays are used to collect the stress of each integration point and the mean stress for each element, i.e. the StressLst and the meanStressLst arrays. Note that for the current calculations, it is not really necessary to share these arrays. The fileName is used to specify the name of the output file and the fileUnit holds its file unit number.

The UBGINC subroutine is used to prepare the output file and to print the table header in increment zero. In addition, it prints a line indicating the number of increments in each subsequent increment.

In the ELEVAR subroutine, from increment 1 onwards, the stress is stored in the StressLst array for each integration point. The last execution of the subroutine for each element is the one that corresponds to the last integration point, i.e. integration point 4 in our case. If this condition is true (line 80), the mean stress of the element is calculated using the Sum intrinsic function (line 83). Note that the second argument of this function, an optional one, indicates that summing is carried out over the second dimension of the stressLst array. The result of this function is an array of rank 3 which holds the sum of the values of the second dimension of the stressLst array. Nevertheless, in addition to the integration point stresses, the mean stress values are printed in the last increment of each element whereas in all other increments, only the stresses of the integration points are printed.

The UEDINC subroutine is used to close the file unit in the last increment, i.e. increment 10.

The result.txt file will contain the following lines for the last increment:

| Element      | IP | Stress(1) | Stress(2) | Stress(3) | Mean Values<br>Stress(1) | Stress(2) | Stress(3) |
|--------------|----|-----------|-----------|-----------|--------------------------|-----------|-----------|
| Increment 10 |    |           |           |           |                          |           |           |
| 1            | 1  | 273.5087  | 59.8075   | 128.2171  |                          |           |           |
| 1            | 2  | 264.1292  | 28.5423   | 57.1641   |                          |           |           |
| 1            | 3  | 172.0044  | 29.3562   | 122.7457  |                          |           |           |
| 1            | 4  | 162.6248  | -1.9090   | 51.6927   | 218.0668                 | 28.9492   | 89.9549   |
| 2            | 1  | 162.2803  | 121.9526  | -181.7169 |                          |           |           |
| 2            | 2  | 283.3122  | 525.3921  | -185.6428 |                          |           |           |
| 2            | 3  | 151.0635  | 118.5875  | -40.5131  |                          |           |           |
| 2            | 4  | 272.0954  | 522.0271  | -44.4389  | 217.1878                 | 321.9898  | -113.0779 |
| 3            | 1  | 344.4247  | 595.9185  | 239.8525  |                          |           |           |
| 3            | 2  | 270.4015  | 349.1743  | 202.1425  |                          |           |           |
| 3            | 3  | 128.9387  | 531.2727  | 67.1316   |                          |           |           |
| 3            | 4  | 54.9155   | 284.5285  | 29.4216   | 199.6701                 | 440.2235  | 134.6371  |
| 4            | 1  | 125.7049  | -41.3143  | -38.0495  |                          |           |           |
| 4            | 2  | 175.8754  | 125.9207  | -1.4357   |                          |           |           |
| 4            | 3  | 151.8576  | -33.4685  | -23.4164  |                          |           |           |
| 4            | 4  | 202.0281  | 133.7665  | 13.1974   | 163.8665                 | 46.2261   | -12.4260  |
| 5            | 1  | 165.4285  | 201.9295  | -5.2660   |                          |           |           |
| 5            | 2  | 186.5401  | 272.3017  | -19.7700  |                          |           |           |

|       |   |          |          |          |          |          |          |
|-------|---|----------|----------|----------|----------|----------|----------|
| 5     | 3 | 144.7085 | 195.7135 | 7.0491   |          |          |          |
| 5     | 4 | 165.8202 | 266.0857 | -7.4548  | 165.6243 | 234.0076 | -6.3604  |
| ----- |   |          |          |          |          |          |          |
| 6     | 1 | 65.5951  | 363.8960 | 45.0434  |          |          |          |
| 6     | 2 | 159.6599 | 677.4454 | 106.6660 |          |          |          |
| 6     | 3 | 241.6596 | 416.7153 | 154.7857 |          |          |          |
| 6     | 4 | 335.7245 | 730.2647 | 216.4083 | 200.6598 | 547.0804 | 130.7259 |
| ----- |   |          |          |          |          |          |          |

It is alternatively possible to calculate the nodal stresses using the ELEVAR subroutine which is not done for this example.

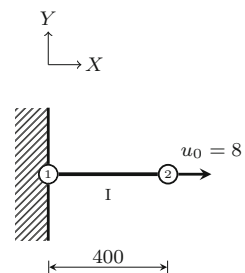
### 4.2.6 UVSCPL

*Example 4.8* This example illustrates the use of the UVSCPL subroutine to perform a linear static analysis in which a rod is elongated by applying a linear displacement. The rod is modeled by means of 2 nodes and one element of type 9, i.e. a two-node simple linear straight truss element with a constant cross section ( $E = 70 \times 10^3$ ,  $L = 400$ , and  $A = 100$ ). The displacement is assumed to be increased incrementally in 10 equal steps. The total displacement is  $u_0 = +8$  which is applied on node 2. The discretized structure is shown in Fig. 4.10.

The UVSCPL subroutine is used to implement general constitutive laws of elastic-viscoplastic materials. There are several inputs provided within the subroutine, e.g. the material properties, stress state, state variables, strains etc. These inputs are used to calculate the required outputs, i.e. the inelastic strain increment (avgine), stress increment (sinc), inelastic strain rate (ustrrt), tangent stiffness matrix (b), and the change in stress due to the state variables (gf). In the most general case, the output variables must be compatible with the employed creep law. In addition, they must satisfy the following equation within a tolerance one order less than that of the global Newton-Raphson algorithm:

$$\text{sinc} = b * (e - \text{avgine} - \text{thmstri}) + \text{gf}, \tag{4.8}$$

**Fig. 4.10** Axial loading of a nonlinear rod defined by the UVSCPL subroutine



where  $\text{thmstri}$  is the thermal strain increment and  $\mathbf{e}$  is the current strain increment. For the general 3D case, all the variables in this equation are arrays but for the current 1D example, all tensorial variables can be simplified to scalar ones. Furthermore, by omitting the effect of the state variables, e.g. temperature, and assuming a time-independent behavior, i.e. omitting the viscous behavior, this equation can be simplified to the following:

$$\text{sinc} = b * (\mathbf{e} - \text{avginc}), \quad (4.9)$$

which can be used to describe an elasto-plastic behavior. This equation is written in terms of the input variables of the subroutine. In mathematical terms, it can be written as the following equation:

$$d\sigma = E^{\text{elpl}} \times (d\epsilon - d\epsilon^{\text{pl}}), \quad (4.10)$$

where the total stress increment  $d\sigma$  is related to the total strain increment  $d\epsilon$  and the plastic strain increment  $d\epsilon^{\text{pl}}$  by the elasto-plastic modulus  $E^{\text{elpl}}$ .

Equation (4.10) is the result of classical additive decomposition of strain and it is valid for the small stress and strain increments and it must be integrated [6]. As mentioned earlier, this equation is simplified for the 1D problem and a similar, but more general, tensorial equation is valid for the 3D case (see [26]). Note that the irreversible nature of the plastic behavior, i.e. the path-dependency, is manifested in the form of an incremental formulation [28].

For the current example, a linear-elastic behavior is assumed and thus, Eq. (4.10) reduces to:

$$d\sigma = E \times d\epsilon, \quad (4.11)$$

which is the incremental form of Hooke's law. In this equation,  $E$  is the elastic modulus of the material.

In every increment, the UVSCPL subroutine is executed twice. Once during the stiffness matrix calculation ( $\text{lovl}=4$ ) and the second run is during the calculation of the residuals ( $\text{lovl}=6$ ) (see Table 2.8). It is the responsibility of the user to provide a compatible set of data in both of these executions. The material behavior and the execution stage determines which output variables are required. For the current linear-elastic case, in the second stage the stress increment must be calculated using Eq. (4.11) whereas in the first stage a zero value must be returned for the stress increment. Since the material has no plastic properties, the plastic strain increment is equal to zero in both stages.

The UVSCPL subroutine can be activated using the VISCO PLAS keyword in the ISOTROPIC or ORTHOTROPIC model definition options. Alternatively, it is possible to activate it in MENTAT by modifying the properties of the material. To do this, the Viscoplasticity check-box must be checked in the Viscoplastic Properties dialog box:

③ Material Properties > Properties > Viscoplasticity

In the same dialog box, the method must be set to User Sub. Uvscpl. Namely, the following command should be executed:

## ③ Material Properties▷ Properties▷ Viscoplasticity▷ Method: User Sub. Uvscpl

Another important note in engaging the UVSCPL subroutine during the analysis is to use a creep loadcase instead of a static one. This should be done even if the subroutine is dealing with a case in which no time-dependent behavior is considered.

The listing of the subroutine is as follows:

```

1 #include 'MarcTools.f'
2
3     SUBROUTINE UVSCPL(young,poiss,shear,b,ustrrt,etot,e,thmsti,eelas,
4         1         s,sinc,gf,epl,avgine,eqcrp,eqcpc,yd,yd1,vscpar,
5         2         dt,dtdl,cptim,timinc,xintp,ngens,m,nn,kcus,
6         3         matus,ndi,nshear,ncrd,ianiso,nstats,inc,ncycle,
7         4         lovl,nvsplm)
8     USE FileTools
9     IMPLICIT NONE
10    ! ** Start of generated type statements **
11    REAL*8 avgine, b, cptim, dt, dtdl, e, eelas, epl, eqcpc, eqcrp
12    REAL*8 etot, gf
13    INTEGER ianiso, inc, kcus, lovl, m, matus, ncrd, ncycle, ndi
14    INTEGER ngens, nn, nshear, nstats, nvsplm
15    REAL*8 poiss, s, shear, sinc, thmsti, timinc, ustrrt, vscpar
16    REAL*8 xintp, yd, yd1, young
17    ! ** End of generated type statements **
18
19    DIMENSION poiss(3,2),young(3,2),b(ngens,ngens),ustrrt(ngens),
20    1         etot(ngens),e(ngens),thmsti(ngens),eelas(ngens),
21    2         s(ngens),sinc(ngens),gf(ngens),epl(ngens),avgine(ngens),
22    3         dt(nstats),dtdl(nstats),xintp(ncrd),
23    4         shear(3,2),vscpar(nvsplm),matus(2),kcus(2)
24
25    CHARACTER (*), PARAMETER :: FILENAME = 'result.txt'
26    INTEGER, SAVE :: fileUnit
27    REAL*8 :: eIMod, strainInc, strain, stress, force
28
29    100    FORMAT (38('-'))
30    200    FORMAT (A4, X, 3(A10, X))
31    300    FORMAT (I4, X, 3(F10.3, X))
32
33    IF (inc .EQ. 0) THEN
34        CALL FindFreeUnit (fileUnit)
35        OPEN (UNIT = fileUnit, File = FILENAME, ACCESS = 'SEQUENTIAL',
36    &        STATUS = 'REPLACE', ACTION = 'WRITE')
37        WRITE (fileUnit, 100)
38        WRITE (fileUnit, 200) 'Inc.', 'Strain', 'Stress', 'Force'
39        WRITE (fileUnit, 300)
40    ELSE
41        eIMod = young(1, 2)
42        strainInc = e(1)
43
44        IF (lovl .EQ. 4) THEN
45            sinc(1) = 0.0D0
46            avgine(1) = 0.0D0
47        ELSE
48            sinc(1) = eIMod * strainInc
49            avgine(1) = 0.0D0
50
51            strain = etot(1) + strainInc
52            stress = s(1) + sinc(1)
53            force = stress * 100.D0
54
55            WRITE (fileUnit, 300) inc, strain, stress, force
56            WRITE (fileUnit, 100)
57
58    IF (inc .EQ. 10) THEN

```



```

59         CLOSE (fileUnit)
60     END IF
61 END IF
62 END IF
63
64     END

```

The output file contains the following lines:

| Inc. | Strain | Stress   | Force      |
|------|--------|----------|------------|
| 1    | 0.002  | 140.000  | 14000.000  |
| 2    | 0.004  | 280.000  | 28000.000  |
| 3    | 0.006  | 420.000  | 42000.000  |
| 4    | 0.008  | 560.000  | 56000.000  |
| 5    | 0.010  | 700.000  | 70000.000  |
| 6    | 0.012  | 840.000  | 84000.000  |
| 7    | 0.014  | 980.000  | 98000.000  |
| 8    | 0.016  | 1120.000 | 112000.000 |
| 9    | 0.018  | 1260.000 | 126000.000 |
| 10   | 0.020  | 1400.000 | 140000.000 |

*Example 4.9* The previous example can be repeated by changing the constitutive material equation to a nonlinear behavior. In the current example,<sup>1</sup> the material is assumed to be isotropic and elasto-plastic with an exponential isotropic hardening behavior. The classic way of capturing the nonlinear behavior of this rate-independent material is utilizing an incremental-iterative procedure which is explained briefly here. For a more descriptive explanation of the nonlinear behavior, one may refer to [10–12, 29, 33, 35].

The general approach for an elasto-plastic material is to divide the load and apply it in increments. Within each increment, several iterations are performed. This type of analysis is called the incremental-iterative analysis. In this type of analysis, the solution at a stage is known, the increment is applied and the solution for the next increment is sought [2].

In a nonlinear finite element analysis, the iterative part of the procedure is carried out on two levels. The first level is the *global level* which establishes the global equilibrium equation. The second level is the *local level* or the *material level* in which the plasticity equations must be satisfied. Generally, each of these levels requires the solution of a nonlinear system of simultaneous equations. In this example, the Newton-Raphson method is used to numerically solve each of these systems.

<sup>1</sup>The original example is adapted from [29]. One may refer to it for details in the hand-calculations of the solution.

MARC handles the global iterative procedure whereas the convergence of the local level is achieved within the UVSCPL subroutine. However, this subroutine is partially engaged with the solution of the global equilibrium equations. This engagement is in terms of providing the tangent stiffness matrix to MARC along with a zero displacement increment to initiate the Newton–Raphson algorithm in increment zero.

To deal with the nonlinearity at the material level, the fully implicit backward-Euler algorithm will be used. This algorithm is a *predictor-corrector method* which predicts a linear behavior for the material and then adjusts this behavior by means of a plastic corrector. The correction part of the algorithm is generally an iterative process which produces the result within a given tolerance. The Newton–Raphson algorithm is used for this part. However, for some special cases such as the simple 1D case of a linear-isotropic hardening, this algorithm results in a closed-form formulation which replaces the iterative procedure [29].

In the current example, the following quadratic flow function is chosen to describe the behavior of the material:

$$k(\kappa) = (350 + 12900\kappa - 1.25 \times 10^5 \kappa^2), \quad (4.12)$$

where the isotropic hardening parameter  $\kappa$  is selected as the effective (equivalent) plastic strain,

$$\epsilon_{\text{eff}}^{\text{pl}} = |\epsilon^{\text{pl}}|. \quad (4.13)$$

Note that in the 1D case, the effective plastic strain can simply be replaced by the plastic strain. Also the plastic modulus can be calculated using the following relation:

$$E^{\text{pl}} = \frac{dk}{d\kappa} = 12900 - 2.5 \times 10^5 \kappa. \quad (4.14)$$

To obtain a better understanding of the local iterative procedure, let us consider a monotonic loading case in which equal displacement increments are applied. In all the increments, elastic behavior is initially considered, i.e. without any regards for the real material behavior, an elastic behavior is predicted. Therefore, the obtained stress is called the *trial stress*  $\sigma_{n+1}^{\text{trial}}$  for the increment.

For the increments located within the elastic region, this prediction corresponds to the real behavior of the material. However, in a particular increment  $n$ , the stress within the material passes the initial yield stress. Namely, the material enters the elasto-plastic region. Now, by using the elastic predictor in the nonlinear region, the stress state is placed outside of the yield surface. This results in an incorrect final stress, incorrect hardening parameter and a positive yield function. These parameters differ from the actual ones. Therefore, in mathematical terms, the produced deviations are non-zero values which can be expressed in terms of three *residuals*. A residual function  $\mathbf{m}$  defines these three residuals in a vector:

$$\mathbf{m}(\sigma, \kappa, \Delta\lambda) = \begin{bmatrix} r_\sigma(\sigma, \Delta\lambda) \\ r_\kappa(\kappa, \Delta\lambda) \\ r_F(\sigma, \kappa) \end{bmatrix} = \begin{bmatrix} E^{-1}\sigma - E^{-1}\sigma_{n+1}^{\text{trial}} + \Delta\lambda \text{sgn}(\sigma) \\ -\kappa + \kappa_{n+1}^{\text{trial}} + \Delta\lambda \\ |\sigma| - k(\kappa) \end{bmatrix}, \quad (4.15)$$

where the vector  $\mathbf{m}$  is a function of the three scalar variables  $\sigma$ ,  $\kappa$  and  $\Delta\lambda$  in a specific increment. These values can be gathered into a *solution vector*  $\mathbf{v}$  and the function can be rewritten in terms of the new vector variable:

$$\mathbf{m}(\mathbf{v}) = \begin{bmatrix} r_\sigma(\mathbf{v}) \\ r_\kappa(\mathbf{v}) \\ r_F(\mathbf{v}) \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} \sigma \\ \kappa \\ \Delta\lambda \end{bmatrix}. \quad (4.16)$$

To capture the real behavior, the goal is to eliminate the residuals. Namely, these three equations should be equal, or at least approximately equal, to zero, namely:

$$\mathbf{m}(\mathbf{v}) = \mathbf{0}. \quad (4.17)$$

This simultaneous system of equations can be solved for its three variables, i.e. the correct stress  $\sigma_{n+1}$ , the correct hardening parameter  $\kappa_{n+1}$ , and the correct plastic multiplier  $\Delta\lambda_{n+1}$ .

Since this system of equations is generally nonlinear, an iterative procedure must be carried out. This is where the full Newton–Raphson method is used—to obtain the roots of the residual function  $\mathbf{m}$ . This iterative procedure starts with the following initial vector:

$$\mathbf{v}^{(0)} = \begin{bmatrix} \sigma^{(0)} \\ \kappa^{(0)} \\ 0 \end{bmatrix}, \quad (4.18)$$

whereas for iteration  $i$  the vector of solution looks like the following:

$$\mathbf{v}^{(i)} = \begin{bmatrix} \sigma^{(i)} \\ \kappa^{(i)} \\ \Delta\lambda^{(i)} \end{bmatrix}. \quad (4.19)$$

In each iteration of the algorithm, the solution is improved. The improved answer for the next iteration ( $i + 1$ ) can be obtained using the following equation:

$$\mathbf{v}^{(i+1)} = \mathbf{v}^{(i)} - (\mathbf{J}(\mathbf{v}^{(i)}))^{-1} \times \mathbf{m}(\mathbf{v}^{(i)}), \quad (4.20)$$

where the inverse of the Jacobian matrix of the residual function  $\mathbf{J}$  is defined as follows:

$$(\mathbf{J}(\mathbf{v}^{(i)}))^{-1} = \frac{\partial \mathbf{m}}{\partial \mathbf{v}^{(i)}} = \frac{E}{E + \frac{dk}{d\kappa}} \begin{bmatrix} \frac{dk}{d\kappa} & -\text{sgn}(\sigma) \frac{dk}{d\kappa} & \text{sgn}(\sigma) \\ \text{sgn}(\sigma) & -1 & -E^{-1} \\ \text{sgn}(\sigma) & E^{-1} \frac{dk}{d\kappa} & -E^{-1} \end{bmatrix}. \quad (4.21)$$

After finishing the iterations, the final vector is the solution of the current increment, i.e. it gives the corrected stress  $\sigma_{n+1}$ , the corrected hardening parameter  $\kappa_{n+1}$ , and corrected plastic multiplier  $\Delta\lambda_{n+1}$  for the current increment. These values are used to calculate the *plastic corrector*.

From the geometrical point of view, the trial stress results in a point outside the yield surface in the stress space. Because any point outside the yield surface is invalid, this trial point should be projected back onto the yield surface. Therefore, a correction is required to be applied to the trial stress state of the current increment. This can be done by the plastic corrector  $\Delta\sigma^{\text{pl}}$  which can be calculated using the following equation:

$$\Delta\sigma^{\text{pl}} = \sigma_{n+1}^{\text{trial}} - \sigma_{n+1}. \quad (4.22)$$

The solution of the residual system of equations  $\mathbf{m}$  in terms of  $\sigma$ ,  $\kappa$  and  $\Delta\lambda$  allows to calculate the plastic predictor and to correct the trial stress. Obviously, the solution is the one which satisfies a condition within a tolerance. The Euclidean norm of the residual vector is used to obtain such a criterion.

The described algorithm can be implemented in the UVSCPL subroutine to handle the local convergence. In addition, this subroutine is engaged with the global iterations. Both of these aspects are incorporated in a flowchart (see Fig. 4.11).

The UVSCPL subroutine requires the elasto-plastic matrix (b), the stress increment (sinc) and the plastic strain increment (avginc) depending on the execution stage which is indicated by the *lovl* flag (see Table 2.8).

The UVSCPL subroutine is executed first before the solution of the global equilibrium equation (*lovl*=4) and for the second time in the same increment, after obtaining the solution at the stress recovery stage (*lovl*=6). In the former stage, in order to update the global residual, the tangent stiffness matrix (elasto-plastic matrix) is required for the global Newton–Raphson algorithm. In the latter stage, the stress increment and the plastic strain increment are required for the local Newton–Raphson algorithm. As stated earlier, the execution of the global Newton–Raphson is carried out by MARC whereas the user is responsible for the local algorithm.

As mentioned earlier, Eq. (4.8) must be satisfied within a tolerance one order less than that of the global Newton–Raphson algorithm. The global tolerance can be set for a particular loadcase by executing the following command:

```
③ Loadcases > Properties > Convergence Testing > Relative Force Tolerance: 0.1
```

Note that the Relative and Residuals radio buttons should be selected in the same dialog box. In the current example, a tolerance of 0.1 is set for the global convergence of the relative residuals and the local convergence testing of the residual increments.

Note that the current UVSCPL subroutine is designed only for monotonic displacement-control loading. In addition, the flowchart focuses on the mathematical calculations and their relation to the subroutine parameters rather than other programming routines, such as the code for handling files. Such parts are omitted to maintain the clarity of the flowchart. The listing for the subroutine is as follows:

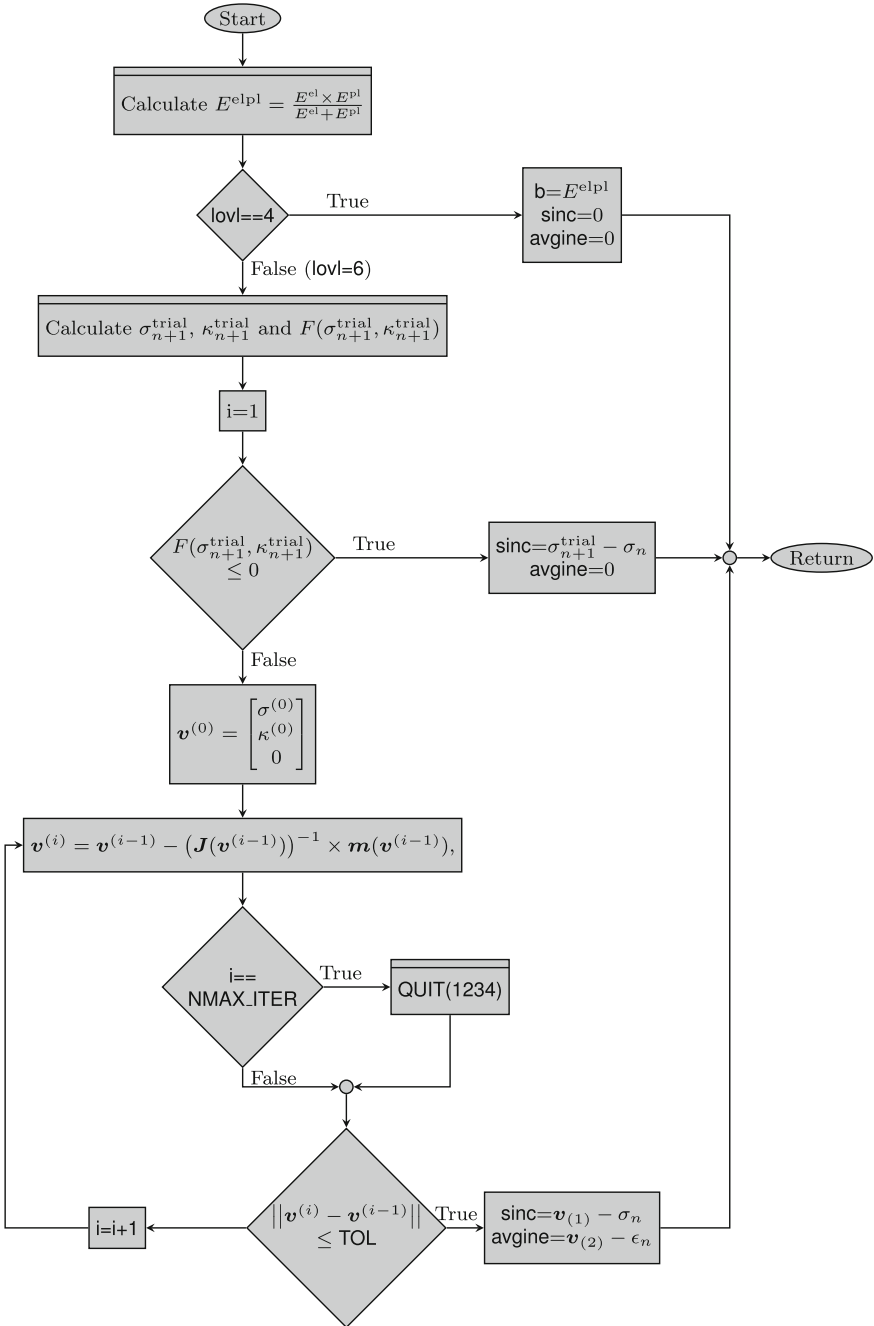


Fig. 4.11 Flowchart for the UVSCPL subroutine (elasto-plastic material behavior)

```

1 #include 'MarcTools.f'
2
3     SUBROUTINE UVSCPL(young,poiss,shear,b,ustrrt,etot,e,thmsti,eelas,
4     1             s,sinc,gf,epl,avgine,eqcrp,eqcpnc,yd,yd1,vscpar,
5     2             dt,dtdl,cptim,timinc,xintp,ngens,m,nn,kcus,
6     3             matus,ndi,nshear,ncrd,ianiso,nstats,inc,ncycle,
7     4             lovl,nvsplm)
8
9     USE FileTools
10    IMPLICIT NONE
11
12    ! ** Start of generated type statements **
13    REAL*8 avgine, b, cptim, dt, dtdl, e, eelas, epl, eqcpnc, eqcrp
14    REAL*8 etot, gf
15    INTEGER ianiso, inc, kcus, lovl, m, matus, ncrd, ncycle, ndi
16    INTEGER ngens, nn, nshear, nstats, nvsplm
17    REAL*8 poiss, s, shear, sinc, thmsti, timinc, ustrrt, vscpar
18    REAL*8 xintp, yd, yd1, young
19    ! ** End of generated type statements **
20
21    DIMENSION poiss(3,2),young(3,2),b(ngens,ngens),ustrrt(ngens),
22    1          etot(ngens),e(ngens),thmsti(ngens),eelas(ngens),
23    2          s(ngens),sinc(ngens),gf(ngens),epl(ngens),avgine(ngens),
24    3          dt(nstats),dtdl(nstats),xintp(ncrd),
25    4          shear(3,2),vscpar(nvsplm),matus(2),kcus(2)
26
27    REAL*8,      PARAMETER :: TOL      = 0.1D0
28    INTEGER,     PARAMETER :: NMAX_ITER = 10
29    CHARACTER (*), PARAMETER :: FILENAME = 'result.txt'
30
31    REAL*8 :: stressLastInc, strainLastInc, elplMod, elMod,
32    & stressTrial, strainTrialPl, strainThisInc,
33    & strainInc, yieldFuncTrial, strainPlLastInc, norm
34
35    REAL*8, DIMENSION(3) :: resFunc, resVar, newResVar
36    REAL*8, DIMENSION(3,3) :: jacobianInv
37
38    LOGICAL :: converged
39    INTEGER :: i
40    INTEGER, SAVE :: fileUnit
41
42    100 FORMAT (109('-'))
43    200 FORMAT (A4, X, A5, X, 7(A13, X))
44    300 FORMAT (I4, X, I5, X, 7(F13.7, X))
45
46    IF (inc .EQ. 0) THEN
47        CALL FindFreeUnit (fileUnit)
48        OPEN (UNIT = fileUnit, File = FILENAME, ACCESS = 'SEQUENTIAL',
49    & STATUS = 'REPLACE', ACTION = 'WRITE')
50        WRITE (fileUnit, 100)
51        WRITE (fileUnit, 200) 'Inc.', 'lter.', 'Tot. Strain',
52    & 'Trial Stress', 'Stress', 'Pl. Strain', 'Pl. Multi.',
53    & 'Elpl. Mod.', 'Norm'
54        WRITE (fileUnit, 100)
55    ELSE
56        elMod          = young(1, 2)
57        stressLastInc = s(1)
58        strainLastInc = etot(1)
59        strainPlLastInc = epl(1)
60        strainInc      = e(1)
61        strainThisInc  = strainLastInc + strainInc
62
63        elplMod = GetEIPIMod (strainPlLastInc, elMod)
64
65        IF (lovl .EQ. 4) THEN
66            b          = elplMod
67            sinc(1)    = 0.0D0

```

```

68     avgine(1) = 0.0D0
69     ELSE
70         stressTrial    = stressLastInc + (elMod * strainInc)
71         strainTrialPI = strainPILastInc
72         yieldFuncTrial = GetYieldFunc (stressTrial , strainTrialPI)
73         i = 1
74         IF (yieldFuncTrial .LE. 0.D0) THEN
75             sinc(1)    = stressTrial - stressLastInc
76             avgine(1) = 0.D0
77             ustrrt    = 0.D0
78             WRITE (fileUnit , 300) inc , i ,
79             &      strainThisInc , stressTrial , stressTrial ,
80             &      0.D0 , 0.D0 , 0.D0 , 0.D0
81
82         ELSE
83             resVar(1) = stressTrial
84             resVar(2) = strainTrialPI
85             resVar(3) = 0.D0
86
87             converged = .FALSE.
88
89             DO WHILE (.NOT. converged)
90                 CALL CalcResFunc()
91                 CALL CalcJacobianInverse()
92                 CALL CalcNewResVar()
93                 CALL CalcNorm()
94
95                 resVar = newResVar
96
97                 WRITE (fileUnit , 300) inc , i ,
98                 &      strainThisInc , stressTrial , resVar(1) ,
99                 &      resVar(2) , resVar(3) , elpIMod , norm
100
101                 IF (norm .LE. tol) converged = .TRUE.
102                 IF (i .EQ. NMAX_ITER) CALL QUIT(1234)
103                 i = i + 1
104             END DO
105
106             sinc(1)    = resVar(1) - stressLastInc
107             avgine(1) = resVar(2) - strainPILastInc
108             ustrrt    = 0.D0
109
110         END IF
111
112         WRITE (fileUnit , 100)
113         IF (inc .EQ. 10) THEN
114             CLOSE (fileUnit)
115         END IF
116     END IF
117 END IF
118 RETURN
119
120 CONTAINS
121 SUBROUTINE CalcNorm ()
122     REAL*8, DIMENSION(3) :: tempVector
123
124     norm = 0.D0
125     tempVector = newResVar - resVar
126     tempVector = tempVector ** 2
127     norm = SQRT (SUM (tempVector))
128     RETURN
129 END SUBROUTINE CalcNorm
130
131 SUBROUTINE CalcNewResVar ()
132     newResVar = ResVar - Matmul(jacobianInv , resFunc)
133     RETURN
134 END SUBROUTINE CalcNewResVar

```

```

135 FUNCTION GetYieldStress (kappa)
136   REAL*8, INTENT(IN) :: kappa
137   REAL*8 :: GetYieldStress
138
139   GetYieldStress = 350.D0 + (12900.D0*kappa) - (1.25D5*kappa**2)
140   RETURN
141 END FUNCTION GetYieldStress
142
143
144 FUNCTION GetYieldFunc(sigma, kappa)
145   REAL*8, INTENT(IN) :: sigma, kappa
146   REAL*8 :: GetYieldFunc
147
148   GetYieldFunc = Abs(sigma) - GetYieldStress (kappa)
149   RETURN
150 END FUNCTION
151
152 SUBROUTINE CalcResFunc()
153   REAL*8 :: sigma, kappa, dLambda
154
155   sigma = resVar(1)
156   kappa = resVar(2)
157   dLambda = resVar(3)
158
159   & resFunc(1) = (sigma) - (stressTrial)
160     + dLambda * eIMod * Sign(1.D0, sigma)
161   resFunc(2) = kappa - strainTrialPI - dLambda
162   resFunc(3) = Abs(sigma) - GetYieldStress(kappa)
163
164   RETURN
165 END SUBROUTINE CalcResFunc
166
167 SUBROUTINE CalcJacobianInverse()
168   REAL*8 :: sigma, kappa, dLambda, coeff, sigmaSign, currentModPI
169   sigma = resVar(1)
170   kappa = resVar(2)
171   dLambda = resVar(3)
172   currentModPI = GetPIMod (kappa)
173
174   coeff = eIMod / (eIMod + currentModPI)
175   sigmaSign = Sign (1.D0, sigma)
176
177   jacobianInv(1,1) = coeff * currentModPI
178   jacobianInv(1,2) = -1.D0 * coeff * sigmaSign * currentModPI
179   jacobianInv(1,3) = coeff * sigmaSign
180
181   jacobianInv(2,1) = coeff * sigmaSign
182   jacobianInv(2,2) = coeff * -1.D0
183   jacobianInv(2,3) = coeff * -1.D0 / eIMod
184
185   jacobianInv(3,1) = coeff * sigmaSign
186   jacobianInv(3,2) = coeff * (1.D0 / eIMod) * currentModPI
187   jacobianInv(3,3) = coeff * -1.D0 / eIMod
188
189   RETURN
190 END SUBROUTINE CalcJacobianInverse
191
192 FUNCTION GetPIMod (kappa)
193   REAL*8, INTENT (IN) :: kappa
194   REAL*8 :: GetPIMod
195
196   GetPIMod = 12900.D0 - 2.5D5 * kappa
197
198   RETURN
199 END FUNCTION GetPIMod
200
201 FUNCTION GetEIPIMod (kappa, Eel)

```



202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216

```

REAL*8, INTENT (IN) :: kappa, Eel
REAL*8 :: GetEIPIMod

REAL*8 :: Epl

IF (kappa .NE. 0.D0) THEN
    Epl = GetPIMod (kappa)
    GetEIPIMod = (Eel * Epl) / (Eel + Epl)
ELSE
    GetEIPIMod = Eel
END IF

RETURN
END FUNCTION GetEIPIMod
END
    
```

In this listing, only the FileTools module is used to handle the output file. This subroutine contains eight internal subprograms to modularize the process and to make it more clear. These subprograms along with a summary of their function are listed in Table 4.2. Note that in this example, the internal subroutines operate on the global variables. Therefore, no explicit input arguments are specified in this table.

In lines 27 to 29 of the listing, the constant parameters are defined, i.e. the tolerance for convergence (TOL), the number of maximum iterations (NMAX\_ITER), and the name of the output file (FILENAME). In lines 31 to 40, the auxiliary variables are declared and in the next three lines, the FORMAT statements define the arrangement of the output data. Lines 47 to 54 prepare the output file and print the header of the table in increment zero. In lines 56 to 60, some auxiliary variables are used to store the arguments. These lines are not necessary in practice but they are used for educational purposes and for increasing the readability of the code. In line 63, the elasto-plastic modulus (elpIMod) is calculated.

Lines 65 to 68 are responsible for returning the elasto-plastic modulus to the global solver without any stress and plastic strain increments. In line 71 and 72, the trial stress is calculated and the value of the trial hardening parameter is set to be equal to that of the previous increment. These two variables are used in the next line of the code to calculate the value of the trial yield function. Lines 74 to 80 cover the

**Table 4.2** Summary of the internal subroutines used in the UVSCPL subroutine

| Subprogram          | Type       | Input(s)                               | Output(s)                             |
|---------------------|------------|--|---------------------------------------|
| CalcJacobianInverse | Subroutine | None                                   | Jacobian inverse                      |
| CalcNorm            | Subroutine | None                                   | Norm of the residual vector increment |
| CalcResFunc         | Subroutine | None                                   | Residual function                     |
| GetEIpIMod          | Function   | Hardening variable and elastic modulus | Elasto-plastic modulus                |
| GetPIMod            | Function   | Hardening variable                     | Plastic modulus                       |
| GetYieldFunction    | Function   | Hardening variable and stress          | Yield function                        |
| GetYieldStress      | Function   | Hardening variable                     | Yield stress                          |

elastic case and print the obtained values to the output file whereas lines 83 to 108 handle the plastic iterative core.

Lines 83 to 87 set the residual variable equal to the starting values and set the convergence status to .FALSE.. In lines 90 to 102, the iterative procedure is carried out by calculating the residual function, inverse of the Jacobian, updated residual and the norm of the residual increment. Then the values of the iteration are printed to the output file (lines 97 to 99). Next, the norm is checked and if it is less than the tolerance, the convergence status will change to .TRUE.. Also to avoid an infinity loop, which occurs when convergence is not possible, the number of iterations (i) is checked to be less than the maximum number (NMAX\_ITER). If the number of iterations exceeds the maximum number, the program quits with an error number 1234.

In the case of convergence, the increment for the stress and hardening parameter is calculated in lines 106 to 108. Note that because this example is rate-independent, the usrrt argument is always zero. Finally, the output file is closed when increment 10 is reached (lines 113 to 115).

The output file containing the result of the analysis consists of the following lines:

| Inc. | Iter. | Tot. Strain | Trial Stress | Stress      | Pl. Strain | Pl. Multi. | Elpl. Mod.    | Nom         |
|------|-------|-------------|--------------|-------------|------------|------------|---------------|-------------|
| 1    | 1     | 0.0020000   | 139.9999999  | 139.9999999 | 0.0000000  | 0.0000000  | 0.0000000     | 0.0000000   |
| 2    | 1     | 0.0040000   | 279.9999997  | 279.9999997 | 0.0000000  | 0.0000000  | 0.0000000     | 0.0000000   |
| 3    | 1     | 0.0060000   | 419.9999996  | 360.8926417 | 0.0008444  | 0.0008444  | 10170.9401717 | 59.1073579  |
| 3    | 2     | 0.0060000   | 419.9999996  | 360.8171937 | 0.0008455  | 0.0008455  | 10170.9401717 | 0.0754480   |
| 4    | 1     | 0.0080000   | 500.8171935  | 382.3002987 | 0.0025386  | 0.0016931  | 9803.4398045  | 118.5168948 |
| 4    | 2     | 0.0080000   | 500.8171935  | 381.9954000 | 0.0025429  | 0.0016975  | 9803.4398045  | 0.3048987   |
| 4    | 3     | 0.0080000   | 500.8171935  | 381.9954084 | 0.0025429  | 0.0016975  | 9803.4398045  | 0.0000084   |
| 5    | 1     | 0.0100000   | 521.9954083  | 402.8671391 | 0.0042448  | 0.0017018  | 9431.3967877  | 119.1282692 |
| 5    | 2     | 0.0100000   | 521.9954083  | 402.5574811 | 0.0042492  | 0.0017063  | 9431.3967877  | 0.3096580   |
| 5    | 3     | 0.0100000   | 521.9954083  | 402.5574790 | 0.0042492  | 0.0017063  | 9431.3967877  | 0.0000021   |
| 6    | 1     | 0.0120000   | 542.5574788  | 422.8082754 | 0.0059599  | 0.0017107  | 9054.7263701  | 119.7492035 |
| 6    | 2     | 0.0120000   | 542.5574788  | 422.4937329 | 0.0059644  | 0.0017152  | 9054.7263701  | 0.3145425   |
| 6    | 3     | 0.0120000   | 542.5574788  | 422.4937014 | 0.0059644  | 0.0017152  | 9054.7263701  | 0.0000314   |
| 7    | 1     | 0.0140000   | 562.4937013  | 442.1137527 | 0.0076841  | 0.0017197  | 8673.3416794  | 120.3799486 |
| 7    | 2     | 0.0140000   | 562.4937013  | 441.7941959 | 0.0076887  | 0.0017243  | 8673.3416794  | 0.3195567   |
| 7    | 3     | 0.0140000   | 562.4937013  | 441.7942031 | 0.0076887  | 0.0017243  | 8673.3416794  | 0.0000072   |
| 8    | 1     | 0.0160000   | 581.7942030  | 460.7734350 | 0.0094175  | 0.0017289  | 8287.1536551  | 121.0207681 |
| 8    | 2     | 0.0160000   | 581.7942030  | 460.4487293 | 0.0094222  | 0.0017335  | 8287.1536551  | 0.3247057   |
| 8    | 3     | 0.0160000   | 581.7942030  | 460.4487361 | 0.0094222  | 0.0017335  | 8287.1536551  | 0.0000068   |
| 9    | 1     | 0.0180000   | 600.4487359  | 478.7768045 | 0.0111603  | 0.0017382  | 7896.0709791  | 121.6719315 |
| 9    | 2     | 0.0180000   | 600.4487359  | 478.4468101 | 0.0111650  | 0.0017429  | 7896.0709791  | 0.3299943   |
| 9    | 3     | 0.0180000   | 600.4487359  | 478.4467989 | 0.0111650  | 0.0017429  | 7896.0709791  | 0.0000112   |
| 10   | 1     | 0.0200000   | 618.4467988  | 496.1130792 | 0.0129127  | 0.0017476  | 7500.0000036  | 122.3337196 |
| 10   | 2     | 0.0200000   | 618.4467988  | 495.7776510 | 0.0129175  | 0.0017524  | 7500.0000036  | 0.3354282   |
| 10   | 3     | 0.0200000   | 618.4467988  | 495.7776485 | 0.0129175  | 0.0017524  | 7500.0000036  | 0.0000025   |

The results of the constitutive law implementation using user coding must be verified by an alternative source such as another software or a built-in model of the same software. Additionally, several special cases must be selected to test the subroutine. For instance, the conditions must be switched between a single element and multiple elements, axial and pure shear condition, displacement-control and load-control increments, uniaxial and multi-axial stress state, uniform and non-uniform stress and strain field etc. Another quick way of testing the subroutine is testing the special cases of the general formulation. For instance, it is possible to test only the undamaged elasto-plastic behavior in the implemented elasto-plastic model with damage. Note that this reveals only one aspect of the whole model [12].

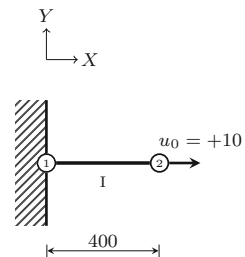
### 4.2.7 USELEM

*Example 4.10* This example illustrates the use of the USELEM subroutine to perform a linear static analysis in which a rod is elongated by applying a linear displacement. The rod is modeled by means of 2 nodes and one element of type 9, i.e. a two-node simple linear straight truss element with a constant cross section ( $E = 200 \times 10^3$ ,  $L = 400$ , and  $A = 100$ ). However, in order to investigate the characteristics of the USELEM subroutine, the same formulation is carried out by the subroutine. The total displacement  $u_0 = +10$  is applied gradually in ten equal increments on node 2. The discretized structure is shown in Fig. 4.12. The goal of this example is to reintroduce the two-node truss element but with the user implementation of the finite element formulations.

In order to incorporate a new element type in MARC/MENTAT, the following four tasks must be accomplished:

1. the new user-defined element type should be introduced in the input file by means of the ELEMENTS parameter,
2. the properties of the element type must be defined using the USER option,
3. the new element type must be assigned to some elements by means of either the CONNECTIVITY option or the UFCONN subroutine, and finally
4. the formulation of the element behavior must be implemented by the user within the USELEM subroutine.

**Fig. 4.12** Axial loading of a linear-elastic rod defined by the USELEM subroutine



In an input file, all the element types of the model are listed in the parameter definition part. The introduction is carried out by the ELEMENTS parameter which is a mandatory keyword in every model. It can be used several times to define different element types in a multi-element simulation. The typical positive values for the built-in element types can be found in [23] whereas the user-defined element types are distinguished by negative numbers. For instance, the quadrilateral four-node plane stress element (type 3), two-node truss elements (type 9) and an arbitrary user-defined element (type -1) are introduced into a model using the following line in the input file:

```
1 ELEMENTS,3 , 9 , - 1 ,
```

Usually a user-defined element is similar to a standard element with a few improved aspects. The goal of the current example is to implement the formulation of the standard element type 9. Therefore, to show the relation of the user-defined element to the existing built-in element, -9 is used as the element type. There is another advantage to this approach which will be discussed later. The following line is added to the parameter definition section of the input file to introduce this new type of element:

```
1 ELEMENTS, -9 ,
```

The USER input file parameter is used to define the parameters of the user-defined element. By default, such an element type is not a standard part of the program. The user-defined element types are indicated by negative numbers whereas MARC uses positive integer numbers to distinguish various standard element types.

The properties of the element type -9 must be defined in the input file using the USER parameter. All the critical values for this user-defined element are summarized in Table 4.3. The following line defines the element using the values of the table:

**Table 4.3** Summary of the element properties for the user-defined element type -9

| Field no. | Property                                 | Value     |
|-----------|--|-----------|
| 2         | Element type                             | -9        |
| 3         | Degrees of freedom per node              | 2         |
| 4         | Stress quantities per integration point  | 1         |
| 5         | Number of nodes per element              | 2         |
| 6         | Number of generalized strains            | 1         |
| 7         | Number of coordinates per node           | 2         |
| 8         | Number of integration points per element | 1         |
| 9         | Number of direct components of stress    | 1         |
| 10        | Number of shear components of stress     | 0         |
| 11        | Element class                            | 1         |
| 12        | Heat transfer flag                       | 0         |
| 13        | Associated heat transfer element         | no values |
| 14        | Topology class                           | 11        |

```
1 USER, -9,2,1,2,1,2,1,1,0,1,0,,11,
```

One simple way of engaging the USELEM subroutine is using the negative element type number as the element type given in the CONNECTIVITY model definition option. Note that an introduction must have been made a priori using the ELEMENTS parameter. This requires modifying the input file. However, the same task can be handled in a more elegant way by means of the UFCONN subroutine.

The UFCONN user subroutine is used to modify the CONNECTIVITY input file option. Alternatively, it is used to generate all the connectivities of the mesh or add new connections to the existing mesh, i.e. expanding the connections.

The activation is done via the UFCONN model definition option followed by the list of elements for which the subroutine is called. The element number can point to either an existing element or a new one.

Similar to the CONNECTIVITY model definition option, the subroutine can be called several times by using multiple UFCONN options in the input file. There is no limitation in regards to the number of the UFCONN option which is used in a model. In our case, the following lines in the model definition section of the input file calls the subroutine for element 1:

```
1 UFCONN
2 1,
```

One powerful aspect of this subroutine is the ability to change the type of the existing element. Therefore, it is possible to keep the existing connectivity of the element but change the type of the element. Obviously, if changing the type of the element results in changing the number of nodes, the connectivity must also be updated. This specific facility is the one which can be used in conjunction with the USELEM subroutine.

The USELEM subroutine is used to formulate the behavior of the user-defined elements. Based on the type of the analysis, various element-related quantities must be calculated and thus, this subroutine is executed multiple times for each iteration of all the increments. Within this subroutine, the iflag variable is used to indicate the stage of every execution. The value of the flag along with the corresponding required outputs are listed in Table 4.4. The calculation of the equivalent nodal loads (iflag=1), the stiffness matrices (iflag=2), the mass matrices (iflag=3), the stresses and strains, and the internal forces (iflag=4) of the elements are carried out within this subroutine. In addition, at the fifth step (iflag=5) the elemental output results are prepared. Note that all of the mentioned values are specified in the global coordinate system.

All the elemental formulations for a user-defined element are implemented within the USELEM subroutine. As mentioned earlier, quantities such as the equivalent nodal loads, the stiffness matrix, the internal load vector, the mass matrix and others should be calculated by the user. However, not all of these quantities are usually required to be calculated. As mentioned earlier, a user-defined element builds up on a standard element. Most of the properties of the new element are the same as those of the standard one which are already provided by MARC. It makes sense to delegate the identical parts to MARC and to concentrate on programming the new aspects. For instance, one may change the stiffness matrix of an element but keep the mass matrix untouched. MARC can readily handle the generation of the mass matrix by using the

**Table 4.4** Summary of the required outputs for the USELEM subroutine in a pure structural analysis

| iflag | Output(s)  | Comments   |
|-------|--|--|
| 1     | Equivalent nodal loads (f)   | Calculate total load: required for the ELASTIC and FOLLOW FOR parameter and the AUTO STEP, AUTO TIME and AUTO INCREMENT options<br>Calculate incremental load: otherwise |
| 2     | Element stiffness matrix (k) total internal forces (r)   | In a nonlinear analysis the element tangential stiffness matrix must be returned. In a linear analysis the total internal force is not required                          |
| 3     | Mass matrix (m)  | Required only for a dynamic analysis   |
| 4     | Incremental strain (de)<br>Generalized stress (gsigs)<br>Total internal forces (r)<br>Total strain (etota) | For a linear analysis if only the displacements are required,<br>There is no need for any of these outputs   |
| 5     | Any elemental output quantities  | Optional   |

already existing formulation. This is done by changing the element type during the execution of the USELEM for that specific stage. In the stage indicated by iflag=3, it is required to return the mass matrix. The following lines will ask MARC to handle the mass matrix generation for our user element type exactly the same way as the standard truss:

```

1      IF (iflag .EQ. 3) THEN
2          jtype = -jtype
3      END IF
    
```

This is the other virtue of naming the user-defined element type -9 which is based on the element type 9. By giving back the original element type, MARC handles the required value as before. The same approach can be used for other stages of execution such as the output calculation (iflag=5). Note that generally the results of the USELEM subroutine cannot be shown in MENTAT unless all the output values are calculated within the subroutine and additionally, the output stage is handled by MARC. For instance in a linear-elastic analysis, it is optional to calculate the internal nodal forces at the iflag=2 stage. However, if the values are required to be present in the post-processing stage of MENTAT, they must be calculated at the iflag=5 stage. For this, the output stage should be handled in a similar fashion as mentioned earlier:

```

1      IF (iflag .EQ. 5) THEN
2          jtype = -jtype
3      END IF

```

It is worth mentioning that while using this subroutine, both the incremental and total displacements are available. At a particular increment, before the solution of the global equilibrium equations, the number of iterations or cycles for the analysis is zero. At this stage, all the calculations must be based on the incremental values. The solution of the global equations is generally done in several cycles; at best with one cycle for a linear case. From this point onwards, the calculation must be carried out using the current displacement which is the total displacement plus the incremental one. One may refer to [25] for more information.

A truss element consists of two nodes with two degrees of freedom in the global coordinate system (plane problem). However, each node has only one degree of freedom in the local coordinate system. For the current case of the horizontal bar, both of the coordinate systems are the same. Therefore, the stiffness matrix of the horizontal truss is:

$$\mathbf{K} = \frac{EA}{L} \begin{bmatrix} +1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad (4.23)$$

where  $E$  is the elastic modulus,  $A$  is the cross sectional area and  $L$  is the length of the truss. Note that only the degrees of freedom along the  $X$ -direction are active because of the non-zero values. The axial strain  $\epsilon$  can be calculated by the following equation:

$$\epsilon = \frac{\delta}{L}, \quad (4.24)$$

where  $\delta$  is the displacement and  $L$  is the length of the rod. Then, the normal stress  $\sigma$  can be calculated using the Hooke's law:

$$\sigma = E\epsilon, \quad (4.25)$$

where  $E$  is the elastic modulus. Finally, the internal force along the  $X$ -direction will be equal to:

$$F_X = \sigma A, \quad (4.26)$$

where  $A$  is the cross sectional area of the rod. These formulas are implemented in the following listing of the subroutine:

```

1  #include 'MarcTools.f'
2      SUBROUTINE UFCOON(j , itype , lm , nnodmx)
3      IMPLICIT NONE
4
5      !      ** Start of generated type statements **
6      INTEGER itype , j , lm , nnodmx
7      !      ** End of generated type statements **
8      DIMENSION lm(*)
9

```

```

10      itype = -9
11
12      RETURN
13      END
14
15      SUBROUTINE USELEM (m,xk,xm,nnode,ndeg,f,r,
16 * jtype ,dispt ,disp ,ndi ,nshear ,jpass ,nstats ,ngenel ,
17 * intel ,coord ,ncrd ,iflag ,idss ,t ,dt ,etota ,gsigs ,de ,
18 * geom ,jgeom ,sigxx ,nstrmu)
19
20      USE MarcTools , ONLY : GetDistance
21      IMPLICIT NONE
22
23      INCLUDE 'concom'
24      INCLUDE 'matdat'
25      ! ** Start of generated type statements **
26      REAL*8 coord , de , disp , dispt , dt , etota , f , geom , gsigs
27      INTEGER idss , iflag , intel , jpass , jgeom , jtype , m , ncrd , ndeg
28      INTEGER ndi , ngenel , nnode , nshear , nstats , nstrmu
29      REAL*8 r , sigxx , t , xk , xm
30      ! ** End of generated type statements **
31      DIMENSION xk(idss , idss) , xm(idss , idss) , dispt(ndeg , *) , disp(ndeg , *)
32      DIMENSION t(nstats , *) , dt(nstats , *) , coord(ncrd , *)
33      DIMENSION etota(ngenel , *) , gsigs(ngenel , *) , de(ngenel , *)
34      DIMENSION f(ndeg , *) , r(ndeg , *) , sigxx(nstrmu , *) , geom(*) , jgeom(*)
35      ! If there is an Inc suffix , the variable is an incremental one ,
36      ! otherwise it is a total value .
37      INTEGER :: i
38      REAL*8 :: eIMod , area , length ,
39      &          disp1 , disp2 , strain , stress ,
40      &          displnc1 , displnc2 , strainInc , stressInc ,
41      &          internalForceInc , internalForce
42
43      REAL*8 , DIMENSION(3) :: point1 , point2
44
45      eIMod = et(3)
46      area = geom(1)
47
48      point1 = 0.D0
49      point2 = 0.D0
50      DO i = 1 , ncrd
51          point1(i) = coord(i , 1)
52          point2(i) = coord(i , 2)
53      END DO
54      length = GetDistance (point1 , point2)
55
56      IF ((iflag .EQ. 2) .OR. (iflag .EQ. 4)) THEN
57          IF (ncycle .EQ. 0) THEN
58              displnc1 = disp(1 , 1)
59              displnc2 = disp(1 , 2)
60          ELSE
61              disp1 = disp(1 , 1) + dispt(1 , 1)
62              disp2 = disp(1 , 2) + dispt(1 , 2)
63          END IF
64      END IF
65
66      SELECT CASE (iflag)
67          CASE (1 , 3 , 5)
68              jtype = -jtype
69          CASE (2 , 4)
70              CALL CalcStiffness ()
71              CALL CalcStressStrain ()
72              CALL CalcInternalForce ()
73      END SELECT
74
75      RETURN
76      CONTAINS

```



```

77
78      SUBROUTINE CalcStiffness ()
79          xk(1,1) = +1.D0
80          xk(1,2) = 0.D0
81          xk(1,3) = -1.D0
82          xk(1,4) = 0.D0
83
84          xk(2,:) = 0.D0
85
86          xk(3,1) = -1.D0
87          xk(3,2) = 0.D0
88          xk(3,3) = +1.D0
89          xk(3,4) = 0.D0
90
91          xk(4,:) = 0.D0
92
93          xk = (elMod * area / length) * xk
94      RETURN
95  END SUBROUTINE CalcStiffness
96
97  SUBROUTINE CalcInternalForce ()
98      internalForceInc = stressInc * area
99      internalForce     = stress * area
100
101      IF (ncycle .GT. 0) THEN
102          r(1,1) = +internalForce
103          r(2,1) = 0.D0
104          r(1,2) = -internalForce
105          r(2,2) = 0.D0
106      ELSE
107          r(1,1) = +internalForceInc
108          r(2,1) = 0.D0
109          r(1,2) = -internalForceInc
110          r(2,2) = 0.D0
111      END IF
112
113      RETURN
114  END SUBROUTINE CalcInternalForce
115
116  SUBROUTINE CalcStressStrain ()
117      strainInc = (displnc2 - displnc1) / length
118      strain    = (disp2 - disp1) / length
119      stress    = elMod * strain
120      stressInc = elMod * strainInc
121
122      IF (ncycle .GT. 0) THEN
123          de(1,1) = strain
124          etota(1,1) = strain
125          gsgs(1,1) = stress
126          sigxx(1,1) = stress
127          r(1,1) = stress * area
128      ELSE
129          de(1,1) = strainInc
130          etota(1,1) = strainInc
131          gsgs(1,1) = stressInc
132          sigxx(1,1) = stressInc
133          r(1,1) = stressInc * area
134      END IF
135      RETURN
136  END SUBROUTINE CalcStressStrain
137
138  END

```

In line 1, the MarcTools module is included just to use the GetDistance function. Lines 2 to 13 hold the body of the UCONN subroutine with only one line as the execution statement, namely line 10 which changes the type of the element to -9.

In the lines 23 and 24 of the USELEM subroutine, the `concom` and the `matdat` predefined common blocks are included to gain access to the number of cycles (`ncycle`) and the material properties (`et` and `geom`), respectively (see Sect. 2.3.3). To highlight the difference between the incremental and total values, separate variables are used for each one as it should be in an educational listing (lines 37 to 43).

In lines 45 and 46, the elastic modulus (`elmod`) and the cross sectional area (`area`) of the rod are obtained. The length (`length`) of the rod is calculated based on the coordinates of its nodes (lines 48 to 54). In lines 56 to 64, the displacements are extracted from the arguments of the subroutine based on the number of cycles.

In lines 66 to 73, a case selection structure is used to direct the flow of the code based on the flag (`iflag`). For the flag values of 1, 3 and 5, the original element type number is returned and thus, MARC handles the calculations for these cases. For the flag values 2 and 4, the stiffness matrix, the stresses and strains, and the internal forces are calculated by calling the `CalcStiffness`, the `CalcStressStrain` and the `CalcInternalForce` internal subroutines, respectively (lines 70 to 73).

The `CalcStiffness` subroutine calculates the stiffness matrix and returns it via the `xk` output variable (lines 78 to 95). The `CalcInternalForce` calculates the incremental internal forces for the case of cycle number equal to zero and otherwise, the total internal force is returned. The `CalcStressStrain` subroutine calculates the incremental and total stresses and strains. Similarly, depending on the cycle number, proper values are returned.

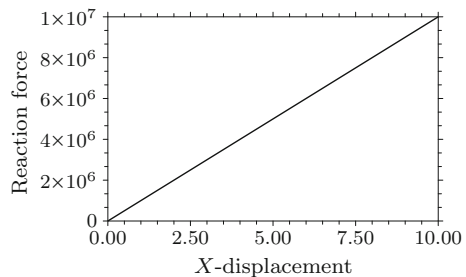
The result of the analysis is illustrated in Fig. 4.13 in terms of displacement along the X-axis versus the reaction force for node 2.

*Example 4.11* In this example, the formulation of a horizontal rod with a variable cross section is implemented using the USELEM subroutine. The model of the previous example is used in the current one and similarly, the simulation of the element type 9 is repeated. However, the cross sectional area varies linearly from  $A_1 = 40$  to  $A_2 = 80$  as follows:

$$A(X) = A_1 + \frac{A_2 - A_1}{L} X, \tag{4.27}$$

where  $L$  is the total length of the rod. The variation of the cross section affects the calculation of the element stiffness matrix  $\mathbf{K}^e$  which can be obtained by evaluating

**Fig. 4.13** Result of using the USELEM subroutine for a horizontal rod with a constant cross section



the following equation:

$$\mathbf{K}^e = \int_0^L \mathbf{B} E A(X) \mathbf{B}^T dX, \quad (4.28)$$

where  $E$  is the elastic modulus,  $A(X)$  is the cross sectional area as a function of  $X$ . The  $\mathbf{B}$ -matrix can be evaluated using the following equation:

$$\mathbf{B}(X) = \frac{d\mathbf{N}(X)}{dX}, \quad (4.29)$$

where  $\mathbf{N}$  is the column matrix of interpolation functions. For the case of a rod, the interpolation function matrix can be simplified to the following:

$$\mathbf{N}(X) = \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} = \begin{bmatrix} 1 - \frac{X}{L} \\ \frac{X}{L} \end{bmatrix}. \quad (4.30)$$

Since the analytical integration of Eq. (4.28) is not possible via FORTRAN, a numerical approach is acquired. For this case, the Gauss–Legendre quadrature is used which requires the transformation from the Cartesian to the natural coordinate system. The transformation will result in the following equation:

$$\mathbf{K}^e = \int_{-1}^{+1} \frac{d\mathbf{N}(\xi)}{d\xi} E A(\xi) \frac{d\mathbf{N}^T(\xi)}{d\xi} \frac{1}{J} d\xi, \quad (4.31)$$

where  $J$  is the Jacobian. This equation can be expanded by substituting  $J = \frac{L}{2}$  and  $\frac{d\mathbf{N}(\xi)}{d\xi} = \left[-\frac{1}{2} + \frac{1}{2}\right]^T$  and considering only one integration point (abscissa  $\xi = 0$  and weight  $\omega = 2$ ) to the following:

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} = \begin{bmatrix} +\frac{1}{4} & -\frac{1}{4} \\ +\frac{1}{4} & -\frac{1}{4} \end{bmatrix} E A(\xi) \frac{2}{L} \times 2. \quad (4.32)$$

This equation gives the element stiffness matrix in the local coordinates (see [30] for more details). Since a horizontal rod is considered in the current example, it is not required to carry out any transformations to the global coordinate system.

The FORTRAN listing for the current example is as follows:

```

1 #include 'MarcTools.f'
2   SUBROUTINE UFCONN(j , itype , lm , nnodmx)
3     IMPLICIT NONE
4
5     ! ** Start of generated type statements **
6     INTEGER itype , j , lm , nnodmx
7     ! ** End of generated type statements **
8     DIMENSION lm(*)
9
10    INTEGER :: i
11

```

```

12         itype = -9
13
14     RETURN
15     END
16
17     SUBROUTINE USELEM (m,xk,xm,nnode,ndeg,f,r,
18 * jtype ,dispt ,disp ,ndi ,nshear ,jpass ,nstats ,ngenel ,
19 * intel ,coord ,ncrd ,iflag ,idss ,t ,dt ,etota ,gsigs ,de ,
20 * geom ,jgeom ,sigxx ,nstrmu)
21
22     USE MarcTools , ONLY : GetDistance
23     IMPLICIT NONE
24
25     INCLUDE 'concom'
26     INCLUDE 'matdat'
27
28     ! ** Start of generated type statements **
29     REAL*8 coord , de , disp , dispt , dt , etota , f , geom , gsigs
30     INTEGER idss , iflag , intel , jpass , jgeom , jtype , m , ncrd , ndeg
31     INTEGER ndi , ngenel , nnode , nshear , nstats , nstrmu
32     REAL*8 r , sigxx , t , xk , xm
33     ! ** End of generated type statements **
34     DIMENSION xk(idss , idss) , xm(idss , idss) , dispt(ndeg , *) , disp(ndeg , *)
35     DIMENSION t(nstats , *) , dt(nstats , *) , coord(ncrd , *)
36     DIMENSION etota(ngenel , *) , gsigs(ngenel , *) , de(ngenel , *)
37     DIMENSION f(ndeg , *) , r(ndeg , *) , sigxx(nstrmu , *) , geom(*) , jgeom(*)
38     ! If there is an lnc suffix , the variable is an incremental one ,
39     ! otherwise it is a total value.
40     INTEGER :: i
41     REAL*8 :: eIMod , length , ksi ,
42     &         dispTot1 , dispTot2 , strain , stress ,
43     &         displnc1 , displnc2 , strainlnc , stresslnc ,
44     &         internalForcelnc , internalForce , k11 , k12 , k22 , b1 , b2
45
46     REAL*8 , PARAMETER :: A1 = 40.D0 , A2 = 80.D0
47
48     REAL*8 , DIMENSION(3) :: point1 , point2
49
50     eIMod = et(3)
51
52     point1 = 0.D0
53     point2 = 0.D0
54     DO i = 1 , ncrd
55         point1(i) = coord(i , 1)
56         point2(i) = coord(i , 2)
57     END DO
58     length = GetDistance (point1 , point2)
59
60     IF ((iflag .EQ. 2) .OR. (iflag .EQ. 4)) THEN
61         IF (ncycle .EQ. 0) THEN
62             displnc1 = disp(1 , 1)
63             displnc2 = disp(1 , 2)
64         ELSE
65             dispTot1 = disp(1 , 1) + dispt(1 , 1)
66             dispTot2 = disp(1 , 2) + dispt(1 , 2)
67         END IF
68     END IF
69
70     ksi = 0.D0
71
72     SELECT CASE (iflag)
73     CASE (1)
74         jtype = -jtype
75     CASE (2)
76         CALL CalcStiffness()
77         CALL CalcStressStrain()
78         CALL CalcInternalForce(ksi)
79     CASE (3)

```

```

79      jtype = -jtype
80      CASE (4)
81          CALL CalcStiffness()
82          CALL CalcStressStrain()
83          CALL CalcInternalForce(ksi)
84      CASE (5)
85          jtype = -jtype
86      END SELECT
87
88      RETURN
89  CONTAINS
90
91      FUNCTION GetArea(ksi)
92          REAL*8, INTENT(IN) :: ksi
93          REAL*8 :: GetArea
94
95          GetArea = A1 + (A2 - A1) * GetX(ksi) / LENGTH
96
97      RETURN
98  END FUNCTION GetArea
99
100     FUNCTION GetX(ksi)
101         REAL*8, INTENT(IN) :: ksi
102         REAL*8 :: GetX
103
104         GetX = 0.5D0 * (ksi + 1.D0) * Length
105     RETURN
106  END FUNCTION GetX
107
108     SUBROUTINE CalcStiffness()
109         REAL*8 :: x, ksi, jacobian, weight
110
111         jacobian = length / 2.D0
112         b1       = -0.5D0
113         b2       = +0.5D0
114
115         ksi      = 0.D0
116         weight   = 2.D0
117
118         k11 = ((b1 * b1) * eIMod * GetArea(ksi) / jacobian) * weight
119         k12 = ((b1 * b2) * eIMod * GetArea(ksi) / jacobian) * weight
120         k22 = ((b2 * b2) * eIMod * GetArea(ksi) / jacobian) * weight
121
122         xk(1,1) = k11
123         xk(1,2) = 0.D0
124         xk(1,3) = k12
125         xk(1,4) = 0.D0
126
127         xk(2,:) = 0.D0
128
129         xk(3,1) = k12
130         xk(3,2) = 0.D0
131         xk(3,3) = k22
132         xk(3,4) = 0.D0
133
134         xk(4,:) = 0.D0
135
136     RETURN
137  END SUBROUTINE CalcStiffness
138
139     SUBROUTINE CalcInternalForce(ksi)
140         REAL*8, INTENT(IN) :: ksi
141
142         internalForceInc = stressInc * GetArea(ksi)
143         internalForce    = stress * GetArea(ksi)
144
145         IF (ncycle .GT. 0) THEN

```

```

146         r(1,1) = +internalForce
147         r(2,1) = 0.D0
148         r(1,2) = -internalForce
149         r(2,2) = 0.D0
150     ELSE
151         r(1,1) = +internalForceInc
152         r(2,1) = 0.D0
153         r(1,2) = -internalForceInc
154         r(2,2) = 0.D0
155     END IF
156     RETURN
157 END SUBROUTINE CalcInternalForce
158
159 SUBROUTINE CalcStressStrain()
160     strainInc = (displnc2 - displnc1) / length
161     strain    = (dispTot2 - dispTot1) / length
162     stress    = elMod * strain
163     stressInc = elMod * strainInc
164
165     IF (ncycle .GT. 0) THEN
166         de(1,1) = strain
167         etota(1,1) = strain
168         gsgs(1,1) = stress
169         sigxx(1,1) = stress
170     ELSE
171         de(1,1) = strainInc
172         etota(1,1) = strainInc
173         gsgs(1,1) = stressInc
174         sigxx(1,1) = stressInc
175     END IF
176     RETURN
177 END SUBROUTINE CalcStressStrain
178
179 END

```

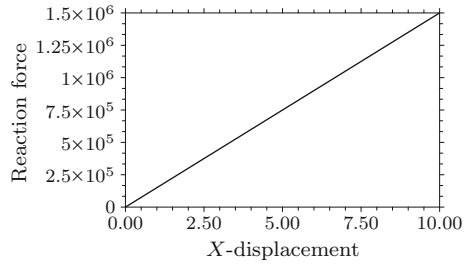
Lines 1 to 90 carry out a similar job as the listing for the previous example. There are two new internal functions in the current one: the `GetArea` and the `GetX` functions. The `GetArea` function receives the abscissa in the natural coordinates ( $\xi$ ) and returns the cross sectional area of the element (lines 91 to 98). This function uses the `GetX` function to transform the abscissa from natural to Cartesian coordinates (lines 100 to 106). In lines 118 to 120, the components of the element stiffness matrix are calculated using the Gauss–Legendre numerical integration. Note that the Jacobian is a constant scalar in this example but it can be replaced by a function for more complicated examples.

Only one integration point is defined in the middle of the rod. Therefore, the internal force is also calculated using the cross sectional area of this point which corresponds to  $\xi = 0$  in the natural coordinate (lines 142 and 143). The rest of the calculations are the same as for the previous example.

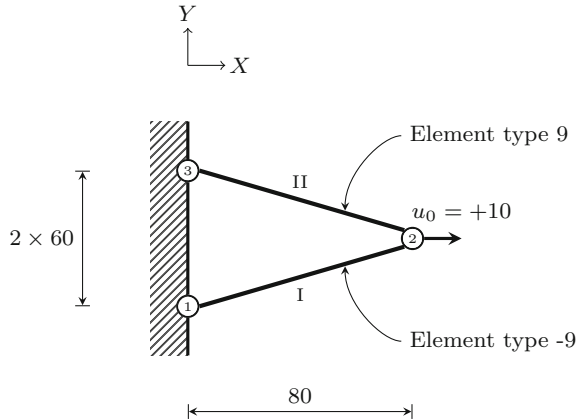
The result of the analysis is illustrated in Fig. 4.14 which shows the displacement of node 2 along the  $X$ -axis versus its reaction force.

*Example 4.12* This example consists of two inclined truss elements as illustrated in Fig. 4.15. The structure is modeled by means of 3 nodes and two truss elements, i.e. two-node simple linear straight truss elements with a constant cross section ( $E = 200 \times 10^3$ ,  $L = 100$ , and  $A = 100$ ). However, element II is a type 9 standard element, whereas element I is a user-defined element of type  $-9$ . Additionally, the

**Fig. 4.14** Result of using the USELEM subroutine for a horizontal rod with a variable cross section



**Fig. 4.15** Axial loading of two linear-elastic rods defined by the USELEM subroutine



stiffness of element I is reduced by 5% in each increment. The formulation of the latter is implemented in the USELEM subroutine by the user. The total displacement  $u_0 = +10$  is applied gradually in ten equal increments on node 2. The goal of this example is to demonstrate the behavior of a user-defined element along with a standard one.

Since the rod element is inclined in this example, the local coordinate system ( $x$ - $y$ ) and the global coordinate system ( $X$ - $Y$ ) are not identical. Therefore, a relation between these two coordinate systems must be established in order to transform several quantities. This is done using the transformation matrix  $T$ :

$$T = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ 0 & 0 & \cos\alpha & \sin\alpha \end{bmatrix}, \tag{4.33}$$

where  $\alpha$  is the angle by which the  $x$ -axis of the local coordinate is rotated with respect to the  $X$ -axis of the global coordinates. The degrees of freedom in the local coordinate system, i.e.  $\mathbf{u}_{xy} = [u_{1x} \ u_{2x}]^T$ , can be transformed to the global coordinate system values, i.e.  $\mathbf{u}_{XY} = [u_{1X} \ u_{1Y} \ u_{2X} \ u_{2Y}]^T$ , using the following equation:

$$\mathbf{u}_{XY} = T^T \mathbf{u}_{xy}. \tag{4.34}$$

Similarly, the force vector  $f$  can be transformed from the local to the global coordinate system:

$$f_{XY} = T^T f_{xy}, \quad (4.35)$$

or vice versa:

$$f_{xy} = T f_{XY}. \quad (4.36)$$

The elemental stiffness matrix in the local coordinate system  $K_{xy}^e$  can be transformed to the global coordinate system using the following equation:

$$K_{XY}^e = T^T K_{xy}^e T^T. \quad (4.37)$$

These equations are useful in our example because all the input and output quantities are specified in the global coordinate system. However, the calculations are carried out in the local coordinate system.

The listing for the current example is as follows:

```

1 #include 'MarcTools.f'
2   SUBROUTINE UCONN(j ,itype ,lm ,nnodmx)
3     IMPLICIT NONE
4     ! ** Start of generated type statements **
5     INTEGER itype , j , lm , nnodmx
6     ! ** End of generated type statements **
7     DIMENSION lm(*)
8
9     INTEGER :: i
10
11     IF (j .EQ. 1) itype = -9
12
13     RETURN
14   END
15
16   SUBROUTINE USELEM (m ,xk ,xm ,nnode ,ndeg ,f ,r ,
17 * jtype ,dispt ,disp ,ndi ,nshear ,jpass ,nstats ,ngenel ,
18 * intel ,coord ,ncrd ,iflag ,idss ,t ,dt ,etota ,gsigs ,de ,
19 * geom ,jgeom ,sigxx ,nstrmu)
20
21     USE MarcTools , ONLY : GetDistance
22     IMPLICIT NONE
23
24     INCLUDE 'concom'
25     INCLUDE 'matdat'
26   ! ** Start of generated type statements **
27   REAL*8 coord , de , disp , dispt , dt , etota , f , geom , gsigs
28   INTEGER idss , iflag , intel , jpass , jgeom , jtype , m , ncrd , ndeg
29   INTEGER ndi , ngenel , nnode , nshear , nstats , nstrmu
30   REAL*8 r , sigxx , t , xk , xm
31   ! ** End of generated type statements **
32   DIMENSION xk(idss , idss) ,xm(idss , idss) , dispt(ndeg ,*) , disp(ndeg ,*)
33   DIMENSION t(nstats ,*) , dt(nstats ,*) , coord(ncrd ,*)
34   DIMENSION etota(ngenel ,*) , gsigs(ngenel ,*) , de(ngenel ,*)
35   DIMENSION f(ndeg ,*) , r(ndeg ,*) , sigxx(nstrmu ,*) , geom(*) , jgeom(*)
36
37   INTEGER :: i
38   REAL*8 :: elMod , length , ksi , strain , stress , internalForce ,
39   &      k11 , k12 , k22 , b1 , b2
40
41   REAL*8 , DIMENSION(3)      :: point1 , point2
42   REAL*8 , DIMENSION(ncrd)  :: cij
43   REAL*8 , DIMENSION(2,2)   :: kLocal

```



```

44 REAL*8, DIMENSION(4,4) :: kGlobal
45 REAL*8, DIMENSION(2,4) :: transMat, tempMat
46 REAL*8, DIMENSION(4,2) :: transMatT
47 REAL*8, DIMENSION(4) :: dispGlobal, intForceGlobal
48 REAL*8, DIMENSION(2) :: dispLocal, intForceLocal
49
50 eIMod = et(3)
51
52 CALL CalcTransMatrix()
53
54 IF ((iflag .EQ. 2) .OR. (iflag .EQ. 4)) THEN
55   IF (ncycle .EQ. 0) THEN
56     dispGlobal(1) = disp(1,1)
57     dispGlobal(2) = disp(2,1)
58     dispGlobal(3) = disp(1,2)
59     dispGlobal(4) = disp(2,2)
60   ELSE
61     dispGlobal(1) = disp(1,1) + dispt(1,1)
62     dispGlobal(2) = disp(2,1) + dispt(2,1)
63     dispGlobal(3) = disp(1,2) + dispt(1,2)
64     dispGlobal(4) = disp(2,2) + dispt(2,2)
65   END IF
66 END IF
67
68 ksi = 0.D0
69
70 SELECT CASE (iflag)
71 CASE (1)
72   jtype = -jtype
73 CASE (2)
74   CALL CalcStiffness()
75   CALL CalcStressStrain()
76   CALL CalcInternalForce(ksi)
77 CASE (3)
78   jtype = -jtype
79 CASE (4)
80   CALL CalcStiffness()
81   CALL CalcStressStrain()
82   CALL CalcInternalForce(ksi)
83 CASE (5)
84   jtype = -jtype
85 END SELECT
86
87 RETURN
88 CONTAINS
89 SUBROUTINE CalcTransMatrix()
90   point1 = 0.D0
91   point2 = 0.D0
92   DO i = 1, ncrd
93     point1(i) = coord(i,1)
94     point2(i) = coord(i,2)
95   END DO
96   length = GetDistance(point1, point2)
97
98   DO i = 1, ncrd
99     cij(i) = (coord(i,2) - coord(i,1)) / length
100  END DO
101   transMat = 0.D0
102   transMat(1,1) = cij(1)
103   transMat(1,2) = cij(2)
104   transMat(2,3) = cij(1)
105   transMat(2,4) = cij(2)
106
107   CALL GMTRA(transMat, transMatT, 2, 4)
108   RETURN
109 END SUBROUTINE CalcTransMatrix
110

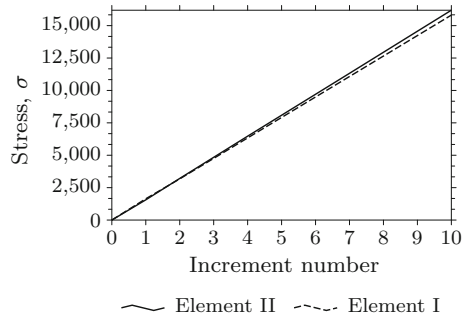
```

```

111 FUNCTION GetArea(ksi)
112 REAL*8, INTENT(IN) :: ksi
113 REAL*8 :: GetArea
114
115 GetArea = geom(1)
116 RETURN
117 END FUNCTION GetArea
118
119 SUBROUTINE CalcStiffness()
120 REAL*8 :: ksi, jacobian, weight
121
122 jacobian = length / 2.D0
123 b1 = -0.5D0
124 b2 = +0.5D0
125
126 ksi = 0.D0
127 weight = 2.D0
128
129 k11 = ((b1 * b1) * eMod * GetArea(ksi) / jacobian) * weight
130 k12 = ((b1 * b2) * eMod * GetArea(ksi) / jacobian) * weight
131 k22 = ((b2 * b2) * eMod * GetArea(ksi) / jacobian) * weight
132
133 kLocal(1,1) = k11
134 kLocal(1,2) = k12
135 kLocal(2,1) = k12
136 kLocal(2,2) = k22
137
138 kLocal = kLocal * (1.D0 - inc * 0.05D0)
139 CALL GMPRD(kLocal, transMat, tempMat, 2, 2, 4)
140 CALL GMPRD(transMatT, tempMat, kGlobal, 4, 2, 4)
141 xk = kGlobal
142 RETURN
143 END SUBROUTINE CalcStiffness
144
145 SUBROUTINE CalcInternalForce(ksi)
146 REAL*8, INTENT(IN) :: ksi
147
148 internalForce = stress * GetArea(ksi)
149 intForceLocal(1) = -internalForce
150 intForceLocal(2) = +internalForce
151
152 CALL GMPRD(transMatT, intForceLocal, intForceGlobal, 4, 2, 1)
153
154 r(1,1) = intForceGlobal(1)
155 r(2,1) = intForceGlobal(2)
156 r(1,2) = intForceGlobal(3)
157 r(2,2) = intForceGlobal(4)
158 RETURN
159 END SUBROUTINE CalcInternalForce
160
161 SUBROUTINE CalcStressStrain()
162 CALL GMPRD(transMat, dispGlobal, dispLocal, 2, 4, 1)
163 strain = (dispLocal(2) - dispLocal(1)) / length
164 stress = eMod * strain
165
166 de(1,1) = strain
167 etota(1,1) = strain
168 gsigs(1,1) = stress
169 sigxx(1,1) = stress
170
171 RETURN
172 END SUBROUTINE CalcStressStrain
173
174 END

```

**Fig. 4.16** Results of using the USELEM subroutine for the user-defined element versus the standard element



In this listing, the MarcTools module is included to make use of the Distance function. In lines 2 to 14, the UFCONN subroutine changes the type of the element I from the standard type 9 to the user-defined type  $-9$ .

In lines 24 and 25 of the USELEM subroutine, the `concom` and the `matdat` common-blocks are included to gain access to the number of cycles (`ncycle`), the cross sectional area of the element (`geom(1)`) and the elastic modulus (`et(3)`). In lines 37 to 48, the auxiliary variables of the subroutine are declared. The general structure of the subroutine is as the previous examples but a transformation is required in this case. The `CalcTransMatrix` is used to calculate the transformation matrix. All the global displacements are acquired in lines 54 to 66 and in the following lines, a similar `CASE SELECT` statement directs the flow of the program as before.

In the lines 90 to 100 of the `CalcTransMatrix` internal subroutine, the coordinates of the nodes are used to calculate the cosine coefficients (`cij`). Then, in lines 101 to 105, the calculated values are transferred to the transformation matrix (`transMat`). In line 107, the transpose of the transformation matrix (`transMatT`) is evaluated using the `GMTRA` utility subroutine (see [23] for more information).

The `GetArea` subroutine returns the cross sectional area of the rod which is a constant in this case.

The `CalcStiffness` subroutine calculates the local elemental stiffness matrix (`kLocal`) using Gauss–Legendre numerical integration (lines 122 to 136). In line 138, a 5% reduction is applied to the stiffness matrix per increment. In lines 139 and 140, the `GMPRD` utility subroutine is used to perform the matrix multiplication required for transforming the stiffness matrix to the global coordinate system (see [23] for more information).

In lines 138 to 150 of the `CalcInternalForce` subroutine, the internal force vector is calculated in the local coordinate system (`intForceLocal`) and in line 152, it is transferred to the global coordinate system using a matrix multiplication.

Finally, in the `CalcStressStrain` subroutine, the stresses and strains are calculated as in the previous example.

The result of this analysis is shown in Fig. 4.16. In this graph, the stresses of elements I and II are plotted versus the increment number.

# Chapter 5

## Listing of the Customized Modules

### 5.1 Overview

There is no unique way of handling a complicated modeling task. However, some typical procedures are in common between various methods. In order to facilitate these recurring procedures encountered during typical subroutine coding, some useful subprograms are created. These customized subprograms are categorized in three sets of modules, namely the MarcTools, FileTools, and MiscTools modules. These subprograms are listed alphabetically in Table 5.1 along with a brief description of each. In addition, the subroutines which use the Elmvar or Nodvar utility subroutines are indicated. The subroutines using the Elmvar utility subroutine should be used in element loops. These subroutines are marked with an asterisk. In contrast, the subroutines with the Nodvar utility subroutine can be utilized everywhere. However, the user must understand that the nodal values may not be the final values of that increment. This depends on which stage of the analysis the Nodvar utility is executed. These subroutines are marked with two asterisks.

In the following sections, further details concerning the subprograms are revealed. One may get a better insight into the structure of MARC by carefully investigating the listings. This is why several subprograms are provided for some tasks. For instance, both of the ExtractElmNodLst and ExtractElmNodLst2 subroutines are capable of extracting the list of nodes for a particular element, but the mechanisms are different.

On the other hand, for those who are not concerned with specific details, the signature of every subroutine is explained.

These subroutines are successfully tested for the examples in Chap. 4. However, this would not guarantee the accuracy of them for every possible model. Proper benchmark examples and validation of the outputs must be carried out prior to accepting the output. However, to provide a good foundation for a keen user to further develop, debug, and remove the limitations of these subprograms, their full listing is provided in the current in this chapter.

**Table 5.1** Summary of the customized modules and subprograms

| Subprogram                         | Description  | Sects. |
|------------------------------------|--|--------|
| MarcTools module                   |  |        |
| CalcNodVal <sup>b</sup>            | Calculates the values for nodal quantities                           | 5.4.1  |
| DelElmFreeEdge                     | Removes the exterior edges of elements from a list of edges          | 5.4.2  |
| ExtractElmEdgeLst                  | Extracts the list of edges for an element                            | 5.4.3  |
| ExtractElmNodLst                   | Extract the list of nodes for an element using a utility subroutine  | 5.4.4  |
| ExtractElmNodLst2                  | Extract the list of nodes for an element using common blocks         | 5.4.5  |
| ExtractNodClosestPLst <sup>b</sup> | Extracts the closest integration points to a node                    | 5.4.6  |
| ExtractSetItemLst                  | Extracts the items in a set  | 5.4.7  |
| GetElmArea <sup>b</sup>            | Returns the surface area of a 4-node quadrilateral element           | 5.4.8  |
| GetElmAveVal <sup>a</sup>          | Returns the average of an elemental value for an element             | 5.4.9  |
| GetElmCenCoord <sup>b</sup>        | Returns the coordinates of the center of an element                  | 5.4.10 |
| GetElmEdgeVal <sup>a</sup>         | Returns the calculated edge value for an elemental quantity          | 5.4.11 |
| GetElmExtID                        | Converts the element internal ID to external/user ID                 | 5.4.12 |
| GetElmIPCCount                     | Returns the number of integration points for an element              | 5.4.13 |
| GetElmIntID                        | Converts the element external/user ID to internal ID                 | 5.4.14 |
| GetIPCoord                         | Returns the coordinates of the integration points                    | 5.4.15 |
| GetIPVal <sup>a,b</sup>            | Evaluates the value of an elemental quantity on an integration point | 5.4.16 |
| GetNodCoord <sup>b</sup>           | Returns the undeformed/deformed coordinates of a node                | 5.4.17 |
| GetNodExtID                        | Converts the node internal ID to external/user ID                    | 5.4.18 |
| GetNodExtraVal                     | Extrapolates the integration point value to a node                   | 5.4.19 |
| GetNodIPVal <sup>a</sup>           | Returns the value of an elemental quantity on the adjacent node      | 5.4.20 |
| GetNodIntID                        | Converts the node external/user ID to internal ID                    | 5.4.21 |
| IsElmIDValid                       | Checks the existence of an element                                   | 5.4.22 |
| IsItemInSet                        | Checks the existence of an item in a list                            | 5.4.23 |
| IsNodIDValid                       | Checks the existence of a node                                       | 5.4.24 |
| MakeElmIDLst                       | Makes a list of all the element IDs                                  | 5.4.25 |
| MakeIPCoordLst                     | Makes a list of integration point coordinates                        | 5.4.26 |
| MakeIPValLst <sup>a,b</sup>        | Makes a list of integration point values                             | 5.4.27 |
| MakeNodCoordLst <sup>b</sup>       | Makes a list of node coordinates                                     | 5.4.28 |
| MakeNodIDLst                       | Makes a list of all node external/user IDs                           | 5.4.29 |
| MakeNodValIPLst <sup>a</sup>       | Makes a list of elemental quantities on the neighboring node         | 5.4.30 |
| MakeNodValLst <sup>b</sup>         | Makes a list of nodal values   | 5.4.31 |
| PrintElmIDGroupedLst               | Prints the list of element IDs and their element groups              | 5.4.32 |
| PrintElmIDLst                      | Prints the list of user element IDs                                  | 5.4.33 |
| PrintIPCoordLst                    | Prints the list of integration point coordinates                     | 5.4.34 |
| PrintIPValLst <sup>a,b</sup>       | Prints the list of elemental quantities                              | 5.4.35 |
| PrintNodCoordLst <sup>b</sup>      | Prints the list of node coordinates                                  | 5.4.36 |
| PrintNodIDLst                      | Prints the list of node IDs  | 5.4.37 |

(continued)

**Table 5.1** (continued)

| Subprogram                   | Description  | Sects. |
|------------------------------|--|--------|
| PrintNodValPLst <sup>a</sup> | Prints the list of nodal values obtained from the integration points     | 5.4.38 |
| PrintNodValLst <sup>b</sup>  | Prints the list of nodal quantities                                      | 5.4.39 |
| PrintSetItemLstID            | Prints the information regarding a set ID                                | 5.4.40 |
| PrintSetItemLstName          | Prints the information regarding a set name                              | 5.4.41 |
| PrintSetLst                  | Prints the general information on sets                                   | 5.4.42 |
| FileTools module             |  |        |
| AutoFilename                 | Renames the trailing three digits of a file name                         | 5.5.3  |
| DeleteFile                   | Deletes an existing file   | 5.5.2  |
| FindFreeUnit                 | Returns a free unit number   | 5.5.1  |
| MiscTools module             |  |        |
| DelRepeated                  | Removes the recurring items of a 1D array                                | 5.6.1  |
| DelRepeated2D                | Removes the recurring items of a 2D array                                | 5.6.2  |
| ExtractIntersectLst          | Extracts the intersecting elements of two arrays                         | 5.6.3  |
| GetDistance                  | Returns the distance between two points                                  | 5.6.4  |
| GetIndex                     | Returns the index of an element in a 1D array                            | 5.6.5  |
| GetRandNum                   | Returns a random real number between $-1$ and $+1$                       | 5.6.6  |
| PrintElapsedTime             | Prints the elapsed time between two consecutive executions               | 5.6.7  |
| PutSmallFirst                | Arranges a 2D array to have the smaller element as the first of the pair | 5.6.8  |
| SwapInt                      | Swaps two integers with each other                                       | 5.6.9  |
| SwapReal                     | Swaps two double precision real numbers with each other                  | 5.6.10 |

<sup>a</sup>Elmvar used<sup>b</sup>Nodvar used

## 5.2 Naming Rules and Abbreviations

A consistent naming of the entities is employed when developing the subprograms. Table 5.2 contains a list of the majority of the commonly used variables. This list helps to better understand the construction of subprograms. Note that in addition to the normal programming conventions (see Sect. 1.1.5), the limitation of the common block variables has affected the selection of the names to avoid conflicts.

The following rules are considered in naming the subprograms:

1. All subprogram names start with a verb.
2. No plural nouns are used in the names.
3. To choose the name of a function:
  - the Get prefix is used if the function returns a numerical value,
  - the Is prefix is used if the function returns a logical value,

**Table 5.2** Summary of the variable names and their descriptions

| Variable name | Description                                  |
|---------------|--|
| nodID         | User ID of the node                          |
| nodIDLst      | A list of node IDs                           |
| intNodID      | Internal ID of the node                      |
| elmID         | User ID of the element                       |
| elmIDLst      | List of the element IDs                      |
| intElmID      | Internal ID of the element                   |
| nodLst        | Array of nodes                               |
| elmLst        | Array of elements                            |
| nElm          | Number of elements                           |
| nNod          | Number of nodes                              |
| IP            | IP number                                    |
| nIP           | Number of IPs                                |
| IPLst         | Array of IPs                                 |
| setName       | Set name                                     |
| itemID        | ID of an item in a set                       |
| lst           | A 1D array (items are general)               |
| nlst          | Number of elements in the list               |
| itemLst       | List of items (items are of type integer)    |
| nItemLst      | Number of items in the list                  |
| tItemLst      | Temp list of items                           |
| tCoordLst     | Temporary list of coordinates                |
| curElm        | Current element                              |
| curNod        | Current node                                 |
| curIP         | Current integration point                    |
| distLst       | List of distances                            |
| curElmVal     | Current elemental value                      |
| curNodVal     | Current nodal value                          |
| curIPVal      | Current integration point value              |
| totIPVal      | Total integration point value                |
| curVal        | Current value                                |
| totVal        | Total value                                  |
| meanVal       | Mean value                                   |
| IPCoord       | Coordinates of the integration point         |
| curIPCoord    | Current coordinates of the integration point |
| totIPCoord    | Total coordinates of the integration point   |
| nodCoord      | Coordinates of the node                      |
| transVal      | Translated value                             |
| extraVal      | Extrapolated value                           |

**Table 5.3** Summary of the abbreviations used in naming the entities

| Abbreviation | Description            |
|--------------|------------------------|
| Nod          | Node/nodal             |
| Elm          | Element/elemental      |
| IP           | Integration point      |
| ID           | Identification         |
| Lst          | List (1D array)        |
| Ave          | Average/averaged       |
| Dist         | Distance               |
| Del          | Delete/remove          |
| Int          | Internal               |
| Ext          | External               |
| Cen          | Center/central         |
| Val          | Value                  |
| Calc         | Calculate/calculation  |
| Coord        | Coordinates            |
| Cur          | Current                |
| Tot          | Total                  |
| Extra        | Extrapolated           |
| Trans        | Translated             |
| Num          | Number                 |
| Rand         | Random                 |
| Rep          | Repetition             |
| n            | Number/count           |
| t            | Temporary              |
| Del          | Delete/remove          |
| Print        | Print to output        |
| Put          | Put/move               |
| Extract      | Extract                |
| Make         | Make/collect/gather    |
| Meth         | Method                 |
| fName        | File name              |
| fUnit        | File unit              |
| Out          | Output                 |
| Inter        | Intersect/intersecting |

4. To choose the name of a subroutine:

- the Calc prefix is used for complex calculations,
- the Print prefix is used if any output information is printed,
- the Make prefix is used if an array of data is collected for a range of nodes/elements (this group usually calls the subprograms with the Get prefix multiple times), and



- the Lst suffix is used if the output parameter is an array.
5. Three levels of subprograms are considered:
    - those which operate on a single node/element, usually start with a Get prefix,
    - the subroutines with the Make prefix are used to make a list of values using the Get group, and
    - the subroutines with the Print prefix create a proper output using the Make group.
  6. Checking the validity of the nodID or elmID is done in the Get functions which are responsible for the single node/element handling.
  7. Any critical errors detected by the subprograms will cause the analysis to stop with an exit code of 1234.

In addition, the used naming abbreviations are listed in Table 5.3.

### 5.3 Modules

The MarcTools module is the main package containing the majority of the subprograms which are related to the finite element practice. These small tools are designed to facilitate the interactions with the MARC/MENTAT package. They mainly deal with the operations involving sets, nodes, elements, edges, integration points, and others. The other two modules, i.e. the MiscTools and the FileTools modules, are used to facilitate various tasks. The MarcTools module uses these two modules by default. In addition, since interacting with MARC requires the access to some of its predefined common blocks, several INCLUDE statements are used to provide them. The specification part of the MarcTools module consists of the following lines:

```

1 #include 'MiscTools.f'
2 #include 'FileTools.f'
3
4     MODULE MarcTools
5         USE MiscTools
6         USE FileTools
7         IMPLICIT NONE
8
9         INCLUDE "pntelm"
10        INCLUDE "crprops"
11        INCLUDE 'spaceivec'
12        INCLUDE 'spaceset'
13        INCLUDE 'array2'
14        INCLUDE 'cdominfo'
15        INCLUDE 'dimen'
16        INCLUDE 'elemdata'
17        INCLUDE 'elmcom'
18        INCLUDE 'heat'
19        INCLUDE 'space'
20        INCLUDE 'lass'
21        INCLUDE 'far'
22        INCLUDE 'blnk'
23        INCLUDE 'prepro'
24        INCLUDE 'nzro1'
25        INCLUDE 'iautcr'
26        INCLUDE 'creeps'

```

27  
28  
29  
30  
31

```

INCLUDE 'concom'

INTEGER, PARAMETER :: MAX_SET_ITEM = 1000
CONTAINS
...
    
```

Aside from the inevitable inclusion of the predefined common blocks, the use of any global variable is generally avoided. The only exception is the `MAX_SET_ITEM` constant which is used to indicate the maximum number of items in a set.

The `FileTools` module contains subprograms for typical I/O operations such as deleting or renaming files. The `MiscTools` module contains subprograms covering miscellaneous tasks such as operations on arrays, calculating the elapsed time of operations, and generating random numbers. Note that not all of the provided subprograms will be used at the same time. Therefore, it is good practice to use the `ONLY` option with the `USE` statement to avoid any possible naming conflicts (see Sect. 1.6.1).

It is worth mentioning that these subroutines are not fully developed to cover every possible circumstance. For instance, if the number of nodes is increased in a model, some modifications may be required to maintain the stability of the subprograms in terms of memory allocation. Additionally, certain modifications will be required for models containing multiple element types or those involved in parallel computations. However, all of the modules are tested to run successfully by the examples in the current chapter.

## 5.4 MarcTools Module

### 5.4.1 CalcNodVal

This subroutine extracts the nodal values of the specified nodal post code.

---

**Input(s):**

|         |         |                          |
|---------|---------|--------------------------|
| nodID   | INTEGER | node user ID             |
| nodCode | INTEGER | nodal post code          |
|         |         | Example node post codes: |
|         |         | 0 Coordinates            |
|         |         | 1 Displacement           |
|         |         | 2 Rotation               |
|         |         | 3 External Force         |
|         |         | 4 External Moment        |
|         |         | 5 Reaction Force         |
|         |         | 6 Reaction Moment        |
|         |         | 79 Total displacement    |

---

**Output(s):**

|         |           |                              |
|---------|-----------|------------------------------|
| valLst  | REAL*8(:) | list of values               |
| nValLst | INTEGER   | number of values in the list |

---

```

325 SUBROUTINE CalcNodVal (nodID, nodCode, valLst, nValLst)
326   INTEGER, INTENT(IN) :: nodID, nodCode
327   REAL*8, ALLOCATABLE, INTENT(OUT) :: valLst(:)
328   INTEGER, INTENT(OUT) :: nValLst
329
330   INTEGER :: dataType
331   REAL*8, DIMENSION(10) :: tValLst
332
333   CALL NodVar (nodCode, nodID, tValLst, nValLst, dataType)
334
335   IF (nValLst.NE. 0) THEN
336     ALLOCATE(valLst, SOURCE = tValLst(1:nValLst))
337   END IF
338
339 END SUBROUTINE CalcNodVal

```

This subroutine uses the NodVar utility subroutine to extract the nodal post codes. The tValLst temporary array is used to hold the returned values from the utility subroutine. After obtaining the number of retrieved values, i.e. nValLst, the output array is allocated using the values as the source.

### 5.4.2 DelElmFreeEdge

This subroutine finds the free edges of an element (exterior edges) in a list and deletes them.

---

#### Input(s):

|         |              |                             |
|---------|--------------|-----------------------------|
| edgeLst | INTEGER(2,*) | list of edges               |
| nEdge   | INTEGER      | number of edges in the list |

---

#### Output(s):

|                |         |                |
|----------------|---------|----------------|
| refinedEdgeLst | INTEGER | list of values |
|----------------|---------|----------------|

---

```

492 SUBROUTINE DeIElmFreeEdge (edgeLst, nEdge, refinedEdgeLst)
493   INTEGER, DIMENSION(2,*) , INTENT(INOUT) :: edgeLst
494   INTEGER, INTENT(IN) :: nEdge
495   INTEGER, ALLOCATABLE, INTENT(OUT) :: refinedEdgeLst(:,:)
496
497   INTEGER :: edgeNod1, nElmLstNod1, edgeNod2, nElmLstNod2
498   INTEGER :: i, nInterLst
499
500   LOGICAL, ALLOCATABLE, DIMENSION(:) :: aMask
501   INTEGER, ALLOCATABLE, DIMENSION(:) :: anIndex, interLst,
502   & elmLstNod1, elmLstNod2
503
504   ALLOCATE (aMask(nEdge))
505   ALLOCATE (elmLstNod1(maxnp))
506   ALLOCATE (elmLstNod2(maxnp))
507
508   elmLstNod1 = 0
509   elmLstNod2 = 0
510   nElmLstNod1 = 0
511   nElmLstNod2 = 0
512
513   DO i = 1, nEdge
514     edgeNod1 = edgeLst(1, i)
515     edgeNod2 = edgeLst(2, i)
516
517     CALL UT_ELEMENTS_AT_NODE (edgeNod1, elmLstNod1, nElmLstNod1)

```

```

518      CALL UT_ELEMENTS_AT_NODE (edgeNod2, elmLstNod2, nElmLstNod2)
519
520      CALL ExtractIntersectLst (elmLstNod1, nElmLstNod1,
521      &      elmLstNod2, nElmLstNod2, interLst, nInterLst )
522
523      aMask(i) = nInterLst .GT. 1
524      END DO
525
526      ALLOCATE(anIndex, source = PACK([(i, i = 1, nEdge)], aMask))
527      ALLOCATE(refinedEdgeLst, source = edgeLst(:, anIndex))
528      RETURN
529      END SUBROUTINE DelElmFreeEdge
    
```

This subroutine receives an array of node couples. Each couple represents the edge of an element. The UT\_ELEMENTS\_AT\_NODE utility subroutine is used to extract the elements connected to each other at the same node. These elements are put in two separate lists. Next, the ExtractIntersectLst subroutine is used to find the intersection of these two lists. Considering that the number of intersections for an exterior edge is one, any two nodes with more than one common element do not belong to an exterior edge. A mask array is used to mark the internal edges by the .TRUE. value. Finally, the refined list of nodes is allocated using this mask.

### 5.4.3 ExtractElmEdgeLst

This subroutine extracts the list of edges for an element.

---

|                  |         |                 |
|------------------|---------|-----------------|
| <b>Input(s):</b> |         |                 |
| elmID            | INTEGER | user element ID |

---

|                   |               |                             |
|-------------------|---------------|-----------------------------|
| <b>Output(s):</b> |               |                             |
| edgeLst           | INTEGER(:, :) | list of edges               |
| nEdgeLst          | INTEGER       | number of edges in the list |

---

```

562      SUBROUTINE ExtractElmEdgeLst (elmID, edgeLst, nEdgeLst)
563      INTEGER, INTENT(IN) :: elmID
564      INTEGER, ALLOCATABLE, DIMENSION(:, :), INTENT(OUT) :: edgeLst
565      INTEGER, INTENT(OUT):: nEdgeLst
566
567      INTEGER, ALLOCATABLE, DIMENSION(:) :: nodLst
568      INTEGER :: nNodLst, i
569
570      CALL ExtractElmNodLst (elmID, nodLst, nNodLst)
571      nEdgeLst = nNodLst
572      ALLOCATE (edgeLst(2, nNodLst))
573      DO i = 1, (nNodLst - 1)
574      edgeLst(1, i) = nodLst(i)
575      edgeLst(2, i) = nodLst(i+1)
576      END DO
577      edgeLst(1, nNodLst) = nodLst(nNodLst)
578      edgeLst(2, nNodLst) = nodLst(1)
579      RETURN
580      END SUBROUTINE ExtractElmEdgeLst
    
```

The ExtractElmEdgeLst subroutine returns the list of edges (edgeList) making up the element (elmID) and the number of edges (nEdge). It uses the ExtractElmNodLst

subroutine to obtain the nodes of the element and then stores the edges in the 2D array of edges.

### 5.4.4 *ExtractElmNodLst*

This subroutine extracts the nodes attached to an element using the ELNODES utility subroutine. It returns zero for an unsuccessful execution as the number of nodes (nNod).

---

**Input(s):**

elmID            INTEGER            user element ID

---

**Output(s):**

nodLst            INTEGER(:)            list of node user IDs  
nNodLst            INTEGER            number of nodes in the list

---

```

1114            SUBROUTINE ExtractElmNodLst (elmID, nodLst, nNod)
1115            INTEGER, INTENT(IN) :: elmID
1116            INTEGER, ALLOCATABLE, DIMENSION(:), INTENT(OUT) :: nodLst
1117            INTEGER, INTENT(OUT) :: nNod
1118
1119            INTEGER :: i, intElID
1120            INTEGER, ALLOCATABLE :: tNodLst(:)
1121
1122            ALLOCATE(tNodLst(nnodmx))
1123
1124            intElID = GetElmIntID (elmID)
1125            IF (intElID .NE. 0) THEN
1126            CALL ElNodes (intElID, nNod, tNodLst)
1127            ALLOCATE (nodLst(nNod))
1128            DO i = 1, nNod
1129                nodLst(i) = GetNodExtID(tNodLst(i))
1130            END DO
1131            ELSE
1132                nNod = 0
1133            END IF
1134            END SUBROUTINE ExtractElmNodLst

```

There are two subroutines which can extract the nodes of elements, i.e. the ExtractElmNodLst and the ExtractElmNodLst2 subroutines. The ExtractElmNodLst subroutine extracts the nodes of an element (elmID) by means of the ELNODES utility subroutine. The ELNODES subroutine receives the internal element ID (intElID) and gives back the temporary list of nodes (tNodLst) along with the exact number of retrieved nodes (nNod). Note that the tNodLst dynamic array must be large enough to hold all the node IDs. It is the user's responsibility to ensure this. To do so, the nnodmx variable from the dimen common block is used. This variable indicates the maximum number of nodes per element (see Table 2.8).

After allocating the temporary node list, the element user ID is converted to the internal ID by means of the GetElmIntID function. A successful execution of this function is indicated by a non-zero return value. In such a case, the ELNODES subroutine can be called. The exact number of nodes (nNod) is calculated by the

subroutine. The temporary list of nodes (tNodLst) contains the internal IDs of the nodes which are converted to the user IDs by means of the GetNodExtID function.

An unsuccessful execution usually happens when the elmID is not valid. This case is pointed out by returning a zero value for the number of nodes (nNod).

### 5.4.5 ExtractElmNodLst2

This subroutine extracts the nodes, attached to an element, using the IELCON array provided in the ELCOM common block. It returns zero for an unsuccessful execution as the number of nodes (nNod).

| Input(s):  |            |                             |
|------------|------------|-----------------------------|
| elmID      | INTEGER    | user element ID             |
| Output(s): |            |                             |
| nodLst     | INTEGER(:) | list of node user IDs       |
| nNodLst    | INTEGER    | number of nodes in the list |

```

1172      SUBROUTINE ExtractElmNodLst2 (elmID, nodLst, nNod)
1173      INTEGER, INTENT(IN) :: elmID
1174      INTEGER, ALLOCATABLE, INTENT(OUT) :: nodLst(:)
1175      INTEGER, INTENT(OUT) :: nNod
1176
1177      INTEGER :: i, intEINum, dataIndex
1178
1179      nNod = 0
1180      IF (nEltyp .EQ. 1) THEN
1181          intEINum = GetElmintID (elmID)
1182          IF (intEINum .NE. 0) THEN
1183              ALLOCATE (nodLst(nNode))
1184              dataIndex = nNode * (intEINum - 1)
1185              DO i = 1, nNode
1186                  nodLst(i) = GetNodExtID (iElCon(dataIndex + i))
1187              END DO
1188              nNod = nNode
1189          END IF
1190      END IF
1191      END SUBROUTINE ExtractElmNodLst2
    
```

The second subroutine for extracting the nodes of an element is the ExtractElmNodLst2 subroutine. This subroutine is an alternative to the ExtractElmNodLst subroutine. Note that it is not as powerful as the ExtractElmNodLst subroutine since it cannot be used for models with various element types.

This subroutine uses the array of the element connectivity (iElCon) and the number of nodes per element (nNode) from the common blocks elmdata and elmcom, respectively. The dataIndex variable indicates the preceding location of the first node of the element in the array. Note that the nNode variable is in an element-based common block (see Table 2.10) but it is obtained outside of an element-based subroutine. Therefore, for a model with various types of elements, it only holds the number of data for the current type which is the last group of elements by default (see Sect. 2.3.3). Namely, the proper execution of this subroutine is limited to the models with only

one type of element. The number of element types (nEltyp) is provided by the dimen common block and it is checked in line 9 to ensure a proper execution. It is possible to further develop the coverage of this subroutine by using additional information from the elemdata common block, e.g. ielgroupinfo. The nNod variable returns zero for an unsuccessful execution.

### 5.4.6 *ExtractNodCloseIPLst*

This subroutine extracts the closest integration points of the adjacent elements with respect to a node.

---

#### Input(s):

|       |         |                       |
|-------|---------|-----------------------|
| nodID | INTEGER | external/user node ID |
|-------|---------|-----------------------|

---

#### Output(s):

|        |            |                                     |
|--------|------------|-------------------------------------|
| nLst   | INTEGER()  | number of items in elmLst and IPLst |
| elmLst | INTEGER(:) | list of user element IDs            |
| IPLst  | INTEGER(:) | list of integration point numbers   |

---

```

619 SUBROUTINE ExtractNodCloseIPLst (nodID, nItemLst, elmLst, IPLst)
620   INTEGER, INTENT(IN) :: nodID
621   INTEGER, INTENT(OUT):: nItemLst
622   INTEGER, ALLOCATABLE, INTENT(OUT):: elmLst(:), IPLst(:)
623
624   INTEGER, ALLOCATABLE :: tElmLst(:)
625   INTEGER :: nTElmLst, i, curIP, curElm
626
627   REAL*8, DIMENSION(3) :: nodCoord, IPCoord
628   REAL*8, ALLOCATABLE :: distanceLst(:)
629
630   ALLOCATE (tElmLst(maxNP))
631   CALL ut_elements_at_node (nodID, tElmLst, nTElmLst)
632   IF (nTElmLst .EQ. 0) THEN
633     nItemLst = 0
634     CALL QUIT(1234)
635     RETURN
636   ELSE
637     nItemLst = nTElmLst
638     ALLOCATE (elmLst(nItemLst))
639     ALLOCATE (IPLst(nItemLst))
640
641     nodCoord = GetNodCoord (nodID, 1)
642
643     ALLOCATE (distanceLst(nintbmx))
644
645     DO i = 1, nItemLst
646       curElm = tElmLst(i)
647       elmLst(i) = curElm
648       DO curIP = 1, nintbmx
649         IPCoord = GetIPCoord(curElm, curIP)
650         distanceLst(curIP) = GetDistance(nodCoord, IPCoord)
651       END DO
652       IPLst(i) = Minloc(distanceLst, 1)
653     END DO
654   END IF
655 END SUBROUTINE ExtractNodCloseIPLst

```

The `ExtractNeighborIP` subroutine receives the user ID of a node (`nodID`) and returns two arrays and a number indicating the quantity of items in these arrays (`nLst`). The `IPList` array contains the list of adjacent integration points with respect to the node. The `elmLst` array contains the element number corresponding to each of the mentioned integration points. Those integration points are chosen which have the minimum distance with respect to the position of the node. Note that both of the output arrays, i.e. the `elmLst` and the `IPList` arrays, are of the `ALLOCATABLE` type because the number of neighboring elements are not known a priori.

The two variables `maxNP` and `nintbmx` are used from the `dimen` common block which indicate the maximum number of connections to a node and the maximum number of integration points, respectively (Sect. 2.3.3). The maximum number of connections is used as an estimated initial value for the number of neighboring elements and are allocated in the `tElmLst` array. Alternatively, a constant could have been used in the module to allocate a sufficiently large array. This array is used to call the utility subroutine `ut_elements_at_node` in order to extract the connected elements to the specified node. The exact number of neighbors is returned by this subroutine via `nTElmLst`. If the call was unsuccessful, no value will be returned for this variable. An unsuccessful call may be a result of asking for the information regarding a non-existing user node. In such a case, depending on the user's preference, the program could either terminate with a specific error code (`CALL QUIT(1234)`) or return with a zero value for the number of items.

In the case of a successful execution, the exact number of neighboring elements (`nItemLst`) is used to allocate the dynamic output arrays (lines 20 and 21). The coordinates of the node are extracted using the `GetNodCoord` function. The `distanceLst` array is allocated using the `nintbmx` variable which holds the list of the distances between the `nodID` node and each integration point (`curlP`) of the current element (`curElm`). Note that lines 27 and 28 may look trivial but the `curElm` auxiliary variable increases the readability of the code. Although one may think that using an extra variable will be a waste of memory, most of the compilers use optimization techniques to eliminate redundant computations [34].

The coordinates of the integration points (`IPCoord`) are extracted using the `GetIPCoord` function and the distance is calculated using the `GetDistance` function. Finally, using the `Minloc` intrinsic function, the index of the minimum distance is obtained. This index corresponds to the number of the integration point as well and it is saved in the `IPList` array in-parallel to the `elmLst` array which contains the respective element IDs.

### 5.4.7 *ExtractSetItemLst*

This subroutine extracts the items in the set which is named as the `setName` parameter.



**Input(s):**

|         |              |                     |
|---------|--------------|---------------------|
| setName | CHARACTER(*) | the name of the set |
|---------|--------------|---------------------|

**Output(s):**

|          |            |  |
|----------|------------|--|
| itemLst  | INTEGER(:) | list of items (either nodes or elements) |
| nItemLst | INTEGER    | number of items in the list              |

```

689      SUBROUTINE ExtractSetItemLst (setName, itemLst, nItemLst)
690
691          CHARACTER(LEN=*) , INTENT(IN)           :: setName
692          INTEGER, INTENT(OUT), ALLOCATABLE, DIMENSION(:) :: itemLst
693          INTEGER, INTENT(OUT)                     :: nItemLst
694
695          INTEGER :: tItemLst(MAX_SET_ITEM), isFound, setType, i
696
697          CALL Marc_SetInf(setName, isFound, tItemLst, setType, nItemLst)
698
699          IF (isFound .EQ. 1) THEN
700              ALLOCATE (itemLst(nItemLst))
701              itemLst = [(tItemLst(i), i = 1, nItemLst)]
702          ELSE
703              nItemLst = 0
704          END IF
705
706          RETURN
707      END SUBROUTINE ExtractSetItemLst

```

The `ExtractSetItems` subroutine is used to extract the members of a set named `setName`. A set name in MARC is generally a 32-character variable. However, to make the unnamed constants strings acceptable as an argument of the subroutine, no length is defined for the `setName` argument, i.e. the `CHARACTER(LEN=*)` declaration is used. It returns the list of the items (`itemLst`) and the number of the items in the list (`nItemLst`). The `itemLst` is a one-dimensional allocatable array to cover every case concerning the unknown number of the members in any set. The subroutine is intended to deal with nodes and elements but it can be modified to return two-dimensional arrays for the sets defining edges, faces etc. Similar to the mechanism utilized in the `IsItemInSet` function, the `Marc_SetInfo` utility subroutine is used in the current subroutine to deal with the sets. A failed execution of the subroutine returns with a zero number of items in the list, i.e. `nItemLst` is assigned a zero value.

### 5.4.8 *GetElmArea*

This function returns the surface area of a 4-node quadrilateral element based on its original shape.

**Input(s):**

|       |         |                 |
|-------|---------|-----------------|
| elmID | INTEGER | user element ID |
|-------|---------|-----------------|

**Output(s):**

|        |                     |
|--------|---------------------|
| REAL*8 | area of the element |
|--------|---------------------|

```

746 FUNCTION GetElmArea (elmID)
747   INTEGER, INTENT(IN) :: elmID
748   REAL*8 :: GetElmArea
749
750   INTEGER, ALLOCATABLE :: nodLst(:)
751   REAL*8, ALLOCATABLE :: nodCoord(:, :)
752   REAL*8 :: elmArea, x1, x2, y1, y2
753   INTEGER :: nNodLst, iNod
754
755   CALL ExtractElmNodLst (elmID, nodLst, nNodLst)
756
757   IF (nNodLst .EQ. 0) THEN
758     CALL QUIT(1234)
759   ELSE
760     ALLOCATE (nodCoord(nNodLst,3))
761     DO iNod = 1, nNodLst
762       nodCoord(iNod,:) = GetNodCoord (nodLst(iNod), 1)
763     END DO
764     elmArea = 0.D0
765     DO iNod = 1, nNodLst
766       x1 = nodCoord(iNod,1)
767       y1 = nodCoord(iNod,2)
768       IF (iNod .EQ. nNodLst) THEN
769         x2 = nodCoord(1,1)
770         y2 = nodCoord(1,2)
771       ELSE
772         x2 = nodCoord(iNod+1,1)
773         y2 = nodCoord(iNod+1,2)
774       END IF
775       elmArea = (x1*y2) - (y1*x2) + elmArea
776     END DO
777     GetElmArea = elmArea * 0.5D0
778   END IF
779 END FUNCTION GetElmArea

```

This subroutine extracts the list of nodes for the element (elmID) by calling the ExtractElmNodLst subroutine. Then, the original coordinates of the nodes are obtained using the GetNodCoord function. It uses Heron’s formula to obtain the area of the polygon.

### 5.4.9 GetElmAveVal

This function returns the average of an elemental value for an element.

---

|                  |         |                               |
|------------------|---------|-------------------------------|
| <b>Input(s):</b> |         |                               |
| elmID            | INTEGER | user element ID               |
| elmCode          | INTEGER | element post code             |
|                  |         | Example elemental post codes: |
|                  |         | 1-6 Components of strain      |
|                  |         | 11-16 Components of stress    |
|                  |         | 17 von Mises stress           |

---

|                   |        |                                      |
|-------------------|--------|--------------------------------------|
| <b>Output(s):</b> |        |                                      |
|                   | REAL*8 | average value of the elemental value |

---



822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854

```

curCoord = 0.D0
totCoord = 0.D0

intElmID = GetElmIntID (elmID)

IF (intElmID .EQ. 0) THEN
  CALL QUIT(1234)
ELSE
  SELECT CASE (calcMeth)
  CASE (1:2)
    CALL ExtractElmNodLst (elmID , nodLst , nNod)
    DO iNod = 1 , nNod
      IF (calcMeth .EQ. 1) THEN
        curCoord = GetNodCoord(nodLst(iNod) , 1)
      ELSE
        curCoord = GetNodCoord(nodLst(iNod) , 2)
      END IF
      totCoord = totCoord + curCoord
    END DO
    GetElmCenCoord = totCoord / nNod
  CASE(3)
    nIP = GetElmIPCount(elmID)

    DO IP = 1 , nIP
      curCoord = GetIPCoord (elmID , IP)
      totCoord = totCoord + curCoord
    END DO
    GetElmCenCoord = totCoord / nIP
  END SELECT
END IF
RETURN
END FUNCTION GetElmCenCoord

```

In this function, three methods are used to obtain the center of the element: original and deformed nodal coordinates and the integration point coordinates. The first two methods use the ExtractElmNodLst subroutine and the GetNodCoord function with the corresponding flag whereas the last method uses the GetElmIPCount and GetIPCoord functions. All the acquired coordinates are summed up in the totCoord array and the average of the values is returned as the coordinates of the center.

### 5.4.11 GetElmEdgeVal

This function returns the calculated edge value for an elemental quantity.

---

**Input(s):**

- |         |         |  |
|---------|---------|--|
| nodID1  | INTEGER | first external/user node ID of the edge  |
| nodID2  | INTEGER | second external/user node ID of the edge |
| elmCode | INTEGER | element post code                        |
- Example elemental post codes:
- 1-6 Components of strain
  - 11-16 Components of stress
  - 17 von Mises stress

---

|          |         |   |
|----------|---------|---|
| calcMeth | INTEGER | calculation method<br>Possible methods:<br>1 Translation (unweighted averaging)<br>2 Extrapolation (unweighted averaging)<br>3 Average (unweighted averaging)<br>4 Translation (weighted averaging)<br>5 Extrapolation (weighted averaging)<br>6 Average (weighted averaging) |
|----------|---------|---|

---

**Output(s):**

|        |                                 |
|--------|---------------------------------|
| REAL*8 | value of the requested quantity |
|--------|---------------------------------|

---

```

901     FUNCTION GetElmEdgeVal (nodID1, nodID2, elmCode, calcMeth)
902     INTEGER, INTENT(IN) :: nodID1, nodID2, elmCode, calcMeth
903     REAL*8 :: GetElmEdgeVal
904
905     GetElmEdgeVal =
906     &         (GetNodIPVal(nodID1, elmCode, calcMeth) +
907     &         GetNodIPVal(nodID2, elmCode, calcMeth)) / 2.0D0
908
909     END FUNCTION GetElmEdgeVal

```

The `GetElmEdgeVal` function simply returns the stress of an edge comprising of two nodes (`nodID1` and `nodID2`). The stress on the edge of the element is defined as the average stress of the nodes which is calculated by the `GetNodIPVal` function. Therefore, it is possible to ask for the same methods used in the `GetNodIPVal` function.

### 5.4.12 *GetElmExtID*

This function converts the internal ID of an element to its corresponding external/user ID.

**Input(s):**

|          |         |                     |
|----------|---------|---------------------|
| intElmID | INTEGER | internal element ID |
|----------|---------|---------------------|

---

**Output(s):**

|         |                          |
|---------|--------------------------|
| INTEGER | external/user element ID |
|---------|--------------------------|

---

```

951     INTEGER FUNCTION GetElmExtID (intElmID)
952     INTEGER, INTENT (IN) :: intElmID
953
954     INTEGER :: ielext, tIntElmID
955
956     tIntElmID = ielext (intElmID)
957
958     IF (tIntElmID .LE. 0) THEN
959     CALL QUIT(1234)
960     ELSE
961     GetElmExtID = tIntElmID
962     END IF
963
964     END FUNCTION GetElmExtID

```

The GetElmExtID function receives the internal element ID (intElmID) and returns the external/user element number. The ielx utility function is used for this conversion. An error code is issued in the case of a wrong conversion.

### 5.4.13 GetElmIPCount

This function returns the number of integration points for an element.

---

**Input(s):**

intElmID    INTEGER    internal element ID

---

**Output(s):**

INTEGER    number of integration points

---

992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012

```

FUNCTION GetElmIPCount (elmID)
  INTEGER, INTENT(IN) :: elmID
  INTEGER :: GetElmIPCount

  INTEGER :: intElmID

  GetElmIPCount = 0

  IF (IsElmIDValid(elmID) .EQV. .TRUE.) THEN
    intElmID = GetElmIntID(elmID)
    IF (intElmID .EQ. 0) THEN
      CALL QUIT(1234)
    ELSE
      CALL SetEl (intElmID)
      GetElmIPCount = jintel
    END IF
  ELSE
    CALL QUIT(1234)
  END IF
  RETURN
END FUNCTION GetElmIPCount
    
```

This function returns the number of integration points for an element (elmID). The default return value for the function is zero and in the case of an error the QUIT subroutine is called with an error code of 1234. The SetEl utility subroutine is used to set the current element and the number of integration points is obtained using the jintel variable.

### 5.4.14 GetElmIntID

This function converts the external ID of an element to its corresponding internal ID.

**Input(s):**

|       |         |                     |
|-------|---------|---------------------|
| elmID | INTEGER | external element ID |
|-------|---------|---------------------|

**Output(s):**

|         |                     |
|---------|---------------------|
| INTEGER | internal element ID |
|---------|---------------------|

```

1068     INTEGER FUNCTION GetElmIntID (elmID)
1069     INTEGER, INTENT (IN) :: elmID
1070
1071     INTEGER :: ielint , tElmIntID
1072
1073     tElmIntID = ielint (elmID)
1074
1075     IF (tElmIntID .LE. 0) THEN
1076         CALL QUIT (1234)
1077     ELSE
1078         GetElmIntID = tElmIntID
1079     END IF
1080
1081     END FUNCTION GetElmIntID

```

The GetElmIntID function makes use of the ielint utility function to convert the external element ID to the internal element number. An error code is issued in the case of a wrong conversion.

### 5.4.15 *GetIPCoord*

This function returns the coordinates of the integration points of an element.

**Input(s):**

|       |         |                          |
|-------|---------|--------------------------|
| elmID | INTEGER | external/user element ID |
| IP    | INTEGER | integration point number |

**Output(s):**

|           |                                      |
|-----------|--------------------------------------|
| REAL*8(3) | coordinates of the integration point |
|-----------|--------------------------------------|

```

1308     FUNCTION GetIPCoord (elmID, IP)
1309     INTEGER, INTENT(IN) :: elmID, IP
1310     REAL*8, DIMENSION(3) :: GetIPCoord
1311
1312     REAL*8 :: tCoord
1313     INTEGER :: i, nElInGroup, intElmID, dataIndex, n, elmIndex
1314
1315     GetIPCoord = [0.D0, 0.D0, 0.D0]
1316
1317     intElmID = GetElmIntID (elmID)
1318     iGroup = 0
1319     elmIndex = 0
1320     DO WHILE ((iGroup .LT. nELGroups) .AND. (elmIndex .EQ. 0))
1321         iGroup = iGroup + 1
1322         CALL Setup_EIGroups (iGroup, nElInGroup, 0, 0, 0)
1323         elmIndex = GetIndex (iElGroup_EINum, nElInGroup, intElmID)
1324     END DO
1325
1326     IF (elmIndex .EQ. 0) THEN

```

1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341

```

CALL QUIT(1234)
ELSE
  ityp = ieltype(intElmID)
  CALL SetEl (intElmID)
  CALL wrat3n (VarsElem(ielsbn), n, elmIndex, iGroup, 0)
  lofr = (n - 1) * nelstr
  dataIndex = icrxpt + (IP - 1) * ncrdmx + lofr
  DO i = 1, ncrd
    tCoord = varselem(dataIndex)
    GetIPCoord(i) = tCoord
    dataIndex = dataIndex + 1
  END DO
END IF
END FUNCTION GetIPCoord
    
```

The GetIPCoord function returns the coordinates of the integration points (IP) of an element (elmID). The function returns an array of three double precision floats and the default values are three zeros. The element number is the user element ID which will be converted to the internal user ID (intElmID). Then, a search will be conducted in every element group, using the IndexArray function, to find this element ID (line 13–17). In line 19, if the elmIndex variable is zero, the element is not found in the iElGroup\_EINum array. The function returns the (0, 0, 0) coordinates by default. It is possible to use the QUIT subroutine of MARC which stops the subroutine and returns an error code. In this case, the code is 1234 (line 21). It is a personal preference to choose between these two options. In any case, the whole listing is adapted from the documentations of MARC [27] which can be referred to a detailed description.

### 5.4.16 GetIPVal

This function evaluates the value of an elemental quantity in an integration point.

**Input(s):**

|         |         |                               |
|---------|---------|-------------------------------|
| elmID   | INTEGER | external/user element ID      |
| IP      | INTEGER | integration point number      |
| elmCode | INTEGER | element post code             |
|         |         | Example elemental post codes: |
|         |         | 1-6 Components of strain      |
|         |         | 11-16 Components of stress    |
|         |         | 17 von Mises stress           |

**Output(s):**

|        |                                 |
|--------|---------------------------------|
| REAL*8 | value of the requested quantity |
|--------|---------------------------------|

1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390

```

FUNCTION GetIPVal (elmID, IP, elmCode)
  INTEGER, INTENT(IN) :: elmID, IP, elmCode
  REAL*8 :: GetIPVal

  REAL*8 :: elmIPVal

  IF ((IP .EQ. 0) .OR. (elmID .EQ. 0))THEN
    CALL QUIT(1234)
    
```



```

1391     ELSE
1392         CALL ElmVar (elmCode, elmID, IP, 0, elmIPVal)
1393         GetIPVal = elmIPVal
1394     END IF
1395     RETURN
1396 END FUNCTION GetIPVal

```

This function returns the scalar value for an integration point. It uses the Elmvar utility subroutine to evaluate the quantity. In the case of an error, the QUIT subroutine is called with an error code of 1234.

### 5.4.17 *GetNodCoord*

This function returns the undeformed/deformed coordinates of a node ID.

---

#### Input(s):

|          |         |                                   |
|----------|---------|-----------------------------------|
| nodID    | INTEGER | external/user nod ID              |
| nodState | INTEGER | the state of the node (optional): |
|          |         | 1 undeformed state (default)      |
|          |         | 2 deformed state                  |

---

#### Output(s):

|           |                           |
|-----------|---------------------------|
| REAL*8(3) | coordinates of the center |
|-----------|---------------------------|

---

```

1435 FUNCTION GetNodCoord (nodID, nodState)
1436     INTEGER, INTENT(IN) :: nodID
1437     INTEGER, OPTIONAL, INTENT(IN) :: nodState
1438     REAL*8, DIMENSION(3) :: GetNodCoord
1439
1440     INTEGER :: nComp, dataType, i
1441     REAL*8, DIMENSION(3) :: nodCoord, nodDisp
1442     INTEGER :: coordState
1443
1444     nodCoord = [0.0D0, 0.0D0, 0.0D0]
1445
1446     IF (PRESENT(nodState)) THEN
1447         coordState = nodState
1448     ELSE
1449         coordState = 1
1450     END IF
1451
1452     CALL NodVar (0, nodID, nodCoord, nComp, dataType)
1453
1454     IF (coordState .EQ. 1) THEN
1455         GetNodCoord = nodCoord
1456     ELSE
1457         CALL NodVar (1, nodID, nodDisp, nComp, dataType)
1458         GetNodCoord = nodCoord + nodDisp
1459     END IF
1460     IF (nComp .EQ. 2) GetNodCoord(3) = 0.D0
1461 END FUNCTION GetNodCoord

```

The GetNodeCoord function returns either the original or the deformed coordinates of a node based on the value of the nodState optional argument. A value of 1 indicates

the original coordinates and a value of 2 is used for the deformed coordinates. If the argument is not provided, the original coordinates are returned by default. The return value of the function is an array of REAL items. This function uses the Nodvar utility subroutine to extract the coordinates. The first argument is the nodal post code which asks for the coordinates of the nodID node. The returned value is stored in a temporary array (tCoordLst). The nComp and dataType variables return the number of components and the returned data type, respectively (see [27]). Note that the node number must be the user node ID. For a 2D analysis which is assumed to be in the X–Y plane, the third dimension is filled with zero.

### 5.4.18 GetNodExtID

This function converts the internal ID of a node to its external/user ID.

---

**Input(s):**

intNodID    INTEGER    internal node ID

---

**Output(s):**

INTEGER    external node ID

---

1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515

```

INTEGER FUNCTION GetNodExtID (intNodID)
  INTEGER, INTENT (IN) :: intNodID

  INTEGER :: tIntNodID , nodext

  tIntNodID = nodext (intNodID)

  IF (tIntNodID .LE. 0) THEN
    CALL QUIT(1234)
  ELSE
    GetNodExtID = tIntNodID
  END IF

END FUNCTION GetNodExtID

```

The GetNodExtID function accepts the internal node ID (intNodID) and returns the user node number by means of the nodext utility subroutine. An error code is issued for the case of a wrong conversion.

### 5.4.19 GetNodExtraVal

This function returns the extrapolated value of an integration point to the corresponding nodes.

**Input(s):**

|         |         |                               |
|---------|---------|-------------------------------|
| elmID   | INTEGER | external/user element ID      |
| nodID   | INTEGER | external/user node ID         |
| IP      | INTEGER | integration point number      |
| elmCode | INTEGER | element post code             |
|         |         | Example elemental post codes: |
|         |         | 1-6 Components of strain      |
|         |         | 11-16 Components of stress    |
|         |         | 17 von Mises stress           |

**Output(s):**

REAL\*8      extrapolated value of the quantity

```

1658 FUNCTION GetNodExtraVal (elmID, nodID, IP, elmCode)
1659 INTEGER, INTENT(IN) :: elmID, nodID, IP, elmCode
1660 REAL*8 :: GetNodExtraVal
1661
1662 REAL*8, ALLOCATABLE :: IPValLst(:)
1663 INTEGER :: nIP, iIP
1664
1665 nIP = GetElmIPCount (elmID)
1666
1667 IF (nIP .NE. 4) THEN
1668 CALL QUIT(1234)
1669 ELSE
1670 ALLOCATE (IPValLst(nIP))
1671 DO iIP = 1, nIP
1672 IPValLst(iIP) = GetIPVal (elmID, iIP, elmCode)
1673 END DO
1674
1675 SELECT CASE (IP)
1676 CASE(1)
1677   GetNodExtraVal =
1678   & 0.25D0 * ((1.D0 + 3.D0 * SQRT(3.D0)) * IPValLst(1) +
1679   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(2) +
1680   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(3) +
1681   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(4))
1682 CASE(2)
1683   GetNodExtraVal =
1684   & 0.25D0 * ((1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(1) +
1685   & (1.D0 + 3.D0 * SQRT(3.D0)) * IPValLst(2) +
1686   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(3) +
1687   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(4))
1688 CASE(4)
1689   GetNodExtraVal =
1690   & 0.25D0 * ((1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(1) +
1691   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(2) +
1692   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(3) +
1693   & (1.D0 + 3.D0 * SQRT(3.D0)) * IPValLst(4))
1694 CASE(3)
1695   GetNodExtraVal =
1696   & 0.25D0 * ((1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(1) +
1697   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(2) +
1698   & (1.D0 + 3.D0 * SQRT(3.D0)) * IPValLst(3) +
1699   & (1.D0 - 1.D0 * SQRT(3.D0)) * IPValLst(4))
1700 END SELECT
1701 END IF
1702 END FUNCTION GetNodExtraVal

```

The extrapolation method used in this subroutine is only applicable for 4-node quadrilateral elements with linear interpolation functions. The integration point values are

obtained using the GetIPVal function and they are used to calculate the extrapolated values. The formulas are adapted from the stress smoothing method introduced in [15].

### 5.4.20 GetNodIPVal

This function returns the equivalent value of an elemental quantity in the neighboring node.

---

**Input(s):**

|          |         |  |
|----------|---------|--|
| nodID    | INTEGER | external/user node ID                  |
| elmCode  | INTEGER | element post code                      |
|          |         | Example elemental post codes:          |
|          |         | 1-6 Components of strain               |
|          |         | 11-16 Components of stress             |
|          |         | 17 von Mises stress                    |
| calcMeth | INTEGER | calculation method                     |
|          |         | Possible methods:                      |
|          |         | 1 Translation (unweighted averaging)   |
|          |         | 2 Extrapolation (unweighted averaging) |
|          |         | 3 Average (unweighted averaging)       |
|          |         | 4 Translation (weighted averaging)     |
|          |         | 5 Extrapolation (weighted averaging)   |
|          |         | 6 Average (weighted averaging)         |

---

**Output(s):**

|        |                                 |
|--------|---------------------------------|
| REAL*8 | value of the requested quantity |
|--------|---------------------------------|

---

```

1762 REAL*8 FUNCTION GetNodIPVal (nodID, elmCode, calcMeth)
1763 INTEGER, INTENT(IN) :: nodID, elmCode, calcMeth
1764
1765 INTEGER :: curElm, curIP, nItemLst, i
1766 INTEGER, ALLOCATABLE :: elmLst(:), IPLst(:)
1767 REAL*8, ALLOCATABLE :: areaLst(:)
1768 REAL*8 :: curVal, totVal, elmAve, elmCen(3), curArea, totArea
1769
1770 totVal = 0.D0
1771 CALL ExtractNodCloseIPLst (nodID, nItemLst, elmLst, IPLst)
1772
1773 IF (nItemLst .EQ. 0) THEN
1774 CALL QUIT(1234)
1775 ELSE
1776 ALLOCATE (areaLst(nItemLst))
1777 totArea = 0.D0
1778
1779 DO i = 1, nItemLst
1780 curElm = elmLst(i)
1781 curIP = IPLst(i)
1782 SELECT CASE (calcMeth)
1783 CASE (1)
1784 curVal = GetIPVal (curElm, curIP, elmCode)
1785 areaLst(i) = 1.D0
1786 CASE (2)

```

```

1787         curVal = GetNodExtraVal(curElm , nodID , curIP , elmCode)
1788         arealst(i) = 1.DO
1789     CASE (3)
1790         curVal = GetElmAveVal (curElm , elmCode)
1791         arealst(i) = 1.DO
1792     CASE (4)
1793         curVal = GetIPVal (curElm , curIP , elmCode)
1794         arealst(i) = GetElmArea(curElm)
1795     CASE (5)
1796         curVal = GetNodExtraVal(curElm , nodID , curIP , elmCode)
1797         arealst(i) = GetElmArea(curElm)
1798     CASE (6)
1799         curVal = GetElmAveVal (curElm , elmCode)
1800         arealst(i) = GetElmArea(curElm)
1801     END SELECT
1802     curArea = areaLst(i)
1803     totArea = totArea + curArea
1804     totVal = totVal + (curVal*curArea)
1805     END DO
1806     GetNodIPVal = totVal / totArea
1807     END IF
1808     END FUNCTION GetNodIPVal

```

The GetNodIPVal function returns the nodal value of an elemental quantity. In general, a node is a common entity between several elements. Therefore, a nodal value, which represents an elemental quantity, is affected by all the neighboring elements. To be more precise, the nodal value is affected by the closest integration point of each element. Therefore, the very first thing to do, is to find the closest integration points with respect to the node. This is done using the ExtractNodCloseIPList subroutine.

Next, all the values obtained from the integration points are projected to the node. All the contributions from elements are summed up and averaged to obtain the nodal value. This projection can be done by several methods such as translation, extrapolation, and averaging. The translation method is just copying the same values of the integration points to the node. The extrapolation method is extrapolating the average value of the element to the node. The average value is the average of the integration point values in each element which is assigned to all of the nodes of that particular element.

The three mentioned methods are provided by MENTAT as standard. In the standard method, the size of each element is not taken into account, i.e. the contribution of elements with different sizes is assumed to be identical. Three additional methods are provided by this subroutine which consider the area of each element as a weight factor, and a weighted average is calculated for the contributions (calcMeth = 5, 6 and 7).

The GetIPVal, GetNodExtraVal and the GetElmAveVal functions are used to obtain the translated, extrapolated, and the averaged values, respectively. For the cases of weighted averaging the GetElmArea function is used to obtain the area of the contributor elements.

### 5.4.21 *GetNodIntID*

This function converts the external ID of a node to its internal ID.

**Input(s):**

nodID            INTEGER    external node ID

**Output(s):**

                  INTEGER    internal node ID

```

1576            INTEGER FUNCTION GetNodIntID (nodIntID)
1577            INTEGER, INTENT (IN) :: nodIntID
1578
1579            INTEGER :: nodint , tNodIntID
1580
1581            tNodIntID = nodint (nodIntID)
1582
1583            IF (tNodIntID .LE. 0) THEN
1584            CALL QUIT (1234)
1585            ELSE
1586            GetNodIntID = tNodIntID
1587            END IF
1588
1589            END FUNCTION GetNodIntID

```

The GetNodIntID function returns the internal node ID by means of the nodint utility subroutine. An error code is issued for the case of a wrong conversion.

### 5.4.22 *IsElmIDValid*

This function returns a .TRUE. value for a valid (existing) element ID.

**Input(s):**

elmID            INTEGER    external/user element ID

**Output(s):**

                  LOGICAL    .TRUE. for an existing element

```

2988            FUNCTION IsElmIDValid (elmID)
2989            INTEGER, INTENT(IN) :: elmID
2990            LOGICAL :: IsElmIDValid
2991
2992            INTEGER, ALLOCATABLE, DIMENSION(:) :: elmIDLst
2993
2994            CALL MakeElmIDLst (elmIDLst)
2995            IF (GetIndex (elmIDLst, numel, elmID) .EQ. 0) THEN
2996            IsElmIDValid = .FALSE.
2997            ELSE
2998            IsElmIDValid = .TRUE.
2999            END IF
3000            RETURN
3001            END FUNCTION IsElmIDValid

```

This function creates a list of element IDs using the MakeElmIDLst and searches for the elmID in the list. If the search is successful a .TRUE. value is returned.

### 5.4.23 *IsItemInSet*

This function checks the existence of an item in a set.

---

#### Input(s):

|         |              |                                       |
|---------|--------------|---------------------------------------|
| setName | CHARACTER(*) | the name of the set                   |
| itemID  | INTEGER      | ID of the set item (node/element)     |
| outUnit | INTEGER      | output unit (optional)<br>default = 6 |

---

#### Output(s):

|         |   |
|---------|---|
| LOGICAL | .TRUE. if the item is in the set<br>.FALSE. if the item is not in the set or the set is not found |
|---------|---|

---

```

1843 LOGICAL FUNCTION IsItemInSet (setName, itemID, outUnit)
1844 CHARACTER (LEN=*), INTENT(IN) :: setName
1845 INTEGER, INTENT(IN) :: itemID
1846 INTEGER, INTENT(IN), OPTIONAL :: outUnit
1847
1848 INTEGER, DIMENSION(MAX_SET_ITEM) :: itemLst
1849 INTEGER :: isFound, setType, nItemLst, i, ou
1850
1851 IF (Present (outUnit)) THEN
1852   ou = outUnit
1853 ELSE
1854   ou = 6
1855 END IF
1856
1857 WRITE (ou,*) '** IsItemInSet SUBROUTINE'
1858 IsItemInSet = .FALSE.
1859 CALL Marc_SetInf(setName, isFound, itemLst, setType, nItemLst)
1860 IF (isFound .EQ. 1) THEN
1861   WRITE(ou,*) '* Set ', setName, ' was found.'
1862   DO i = 1, nItemLst
1863     IF (itemID .EQ. itemLst(i)) THEN
1864       WRITE(ou,*) '* Item ', itemID, ' was found.'
1865       IsItemInSet = .TRUE.
1866     END IF
1867   END DO
1868 ELSE IF (isFound .EQ. 0) THEN
1869   WRITE(ou,*) '* setName = ', setName, ' was not found.'
1870 END IF
1871 RETURN
1872 END FUNCTION IsItemInSet

```

The `IsItemInSet` function checks if the `itemID`, referencing a node or an element, is a member of the set named `setName`; a `.TRUE.` value is returned for a positive result. The function uses the utility subroutine `Marc_SetInfo` which generally can work on any type of sets, but here, only elements or nodes are a matter of concern. While using this utility subroutine, it is required to make sure that the array containing the items (`itemLst`) is large enough to hold the members of the set. This can be done by assigning to the constant `MAX_SET_ITEM` a large enough number. The function returns a `.FALSE.` value if either the set or the member in the set is not found.

### 5.4.24 *IsNodIDValid*

This function returns a `.TRUE.` value for a valid (existing) node ID.

---

|                  |         |                       |
|------------------|---------|-----------------------|
| <b>Input(s):</b> |         |                       |
| nodID            | INTEGER | external/user node ID |

---

|                   |         |   |
|-------------------|---------|---|
| <b>Output(s):</b> |         |   |
|                   | LOGICAL | <code>.TRUE.</code> for an existing element |

---

```

2945 FUNCTION IsNodIDValid (nodID)
2946     INTEGER, INTENT(IN) :: nodID
2947     LOGICAL :: IsNodIDValid
2948
2949     INTEGER, ALLOCATABLE, DIMENSION(:) :: nodIDLst
2950
2951     CALL MakeNodIDLst (nodIDLst)
2952     IF (GetIndex (nodIDLst, numnp, nodID) .EQ. 0) THEN
2953         IsNodIDValid = .FALSE.
2954     ELSE
2955         IsNodIDValid = .TRUE.
2956     END IF
2957     RETURN
2958 END FUNCTION IsNodIDValid

```

This function creates a list of node IDs using the `MakeNodIDLst` and searches for the `nodID` in the list. If the search is successful, a `.TRUE.` value is returned.

### 5.4.25 *MakeElmIDLst*

This subroutine creates a list of all the element IDs of the model.

---

|                  |  |  |
|------------------|--|--|
| <b>Input(s):</b> |  |  |
| none             |  |  |

---

|                   |            |                          |
|-------------------|------------|--------------------------|
| <b>Output(s):</b> |            |                          |
| elmIDLst          | INTEGER(:) | external/user element ID |

---

```

2907 SUBROUTINE MakeElmIDLst (elmIDLst)
2908     INTEGER, ALLOCATABLE, DIMENSION(:), INTENT(OUT) :: elmIDLst
2909
2910     INTEGER :: i
2911
2912     ALLOCATE (elmIDLst(numel))
2913     elmIDLst = [(GetElmExtID(i), i = 1, numel)]
2914     RETURN
2915 END SUBROUTINE MakeElmIDLst

```

The allocatable output is allocated for the total number of elements in the model (`numel`). The `GetElmExtID` subroutine is used to fill up the array with the user element IDs.



### 5.4.26 *MakeIPCoordLst*

This subroutine creates a list of coordinates for the integration points of all the elements in a model or a list of specified elements.

---

#### Input(s):

|           |            |   |
|-----------|------------|---|
| elmIDLst  | INTEGER(:) | list of external/user element ID (optional) |
| nElmIDLst | INTEGER    | number of elements in the list (optional)   |

---

#### Output(s):

|             |                 |  |
|-------------|-----------------|--|
| IPCoordLst  | REAL*8(:, :, :) | array of integration point numbers and coordinates |
| nIPCoordLst | INTEGER         | number of integration points in the list           |

---

```

3142 SUBROUTINE MakeIPCoordLst (IPCoordLst, nIPCoordLst,
3143 & elmIDLst, nElmIDLst)
3144
3145 REAL*8, ALLOCATABLE, DIMENSION(:, :, :), INTENT(OUT) :: IPCoordLst
3146 INTEGER, INTENT(OUT) :: nIPCoordLst
3147 INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: elmIDLst
3148 INTEGER, INTENT(IN), OPTIONAL :: nElmIDLst
3149
3150 INTEGER :: elmID, intElmID, i, j, nIP
3151
3152 IF (Present(elmIDLst) .AND. Present(nElmIDLst)) THEN
3153
3154 ALLOCATE (IPCoordLst(nElmIDLst, nintbmx, 3))
3155 IPCoordLst = 0.D0
3156
3157 DO i = 1, nElmIDLst
3158   elmID = elmIDLst(i)
3159   nIP = GetElmIPCount(elmID)
3160   DO j = 1, nIP
3161     IPCoordLst(i, j, :) = GetIPCoord(elmID, j)
3162   END DO
3163 END DO
3164
3165 nIPCoordLst = nElmIDLst
3166 ELSE
3167
3168 ALLOCATE (IPCoordLst(numel, nintbmx, 3))
3169 IPCoordLst = 0.D0
3170
3171 DO intElmID = 1, numel
3172   elmID = GetElmExtID(intElmID)
3173   nIP = GetElmIPCount(elmID)
3174   DO j = 1, nIP
3175     IPCoordLst(intElmID, j, :) = GetIPCoord(elmID, j)
3176   END DO
3177 END DO
3178
3179 nIPCoordLst = numel
3180 END IF
3181 RETURN
3182 END SUBROUTINE MakeIPCoordLst

```

This subroutine creates a 3D array containing the integration points of elements and their coordinates. It can be done either for all the elements of the model or for a specific set of elements indicated by the (elmIDLst) array. The GetIPCoord function is used to obtain the coordinates for each individual integration point.

### 5.4.27 *MakeIPValLst*

This subroutine makes a list of integration point values for all elements or a selected list of elements of the model.

**Input(s):**

|           |            |   |
|-----------|------------|---|
| elmCode   | INTEGER    | element post code<br>Example elemental post codes:<br>1-6 Components of strain<br>11-16 Components of stress<br>17 von Mises stress |
| elmIDLst  | INTEGER(:) | list of external/user element ID (optional)   |
| nElmIDLst | INTEGER    | number of elements in the list (optional)   |

**Output(s):**

|           |             |  |
|-----------|-------------|--|
| IPValLst  | REAL*8(:,:) | array of integration point numbers and coordinates |
| nIPValLst | INTEGER     | number of integration points in the list           |

```

3221 SUBROUTINE MakeIPValLst (elmCode, IPValLst, nIPValLst,
3222 &
3223     INTEGER, INTENT(IN) :: elmCode
3224     REAL*8, ALLOCATABLE, DIMENSION(:, :), INTENT(OUT) :: IPValLst
3225     INTEGER, INTENT(OUT) :: nIPValLst
3226     INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: elmIDLst
3227     INTEGER, INTENT(IN), OPTIONAL :: nElmIDLst
3228
3229     INTEGER :: elmID, intElmID, i, nIP, IP
3230
3231     IF (Present(elmIDLst) .AND. Present(nElmIDLst)) THEN
3232         ALLOCATE (IPValLst(nElmIDLst, nintbmx))
3233         IPValLst = 0.D0
3234
3235         DO i = 1, nElmIDLst
3236             elmID = elmIDLst(i)
3237             nIP = GetElmIPCount(elmID)
3238             DO IP = 1, nIP
3239                 IPValLst(i, IP) = GetIPVal(elmID, IP, elmCode)
3240             END DO
3241         END DO
3242     ELSE
3243         nIPValLst = nElmIDLst
3244     ELSE
3245         ALLOCATE (IPValLst(numel, nintbmx))
3246         IPValLst = 0.D0
3247
3248         DO intElmID = 1, numel
3249             elmID = GetElmExtID(intElmID)
3250             nIP = GetElmIPCount(elmID)
3251             DO IP = 1, nIP
3252                 IPValLst(intElmID, IP) = GetIPVal(elmID, IP, elmCode)
3253             END DO
3254         END DO
3255     END IF
3256
3257     nIPValLst = numel
3258     END IF
3259     RETURN
3260 END SUBROUTINE MakeIPValLst

```

This subroutine uses the `GetIPVal` function to obtain the elemental value for an integration point. The output is a list of these values which is obtained either for all of the elements or for a group of elements indicated by the `(elmIDLst)` array.

### 5.4.28 *MakeNodCoordLst*

This subroutine makes a list of the coordinates for either a selected list of nodes or all the nodes of the model.

---

#### Input(s):

|                       |            |   |
|-----------------------|------------|---|
| <code>nodState</code> | INTEGER    | the state of the node (optional):<br>1 undeformed state (default)<br>2 deformed state |
| <code>nodLst</code>   | INTEGER(:) | list of selected nodes (optional)   |
| <code>nNodLst</code>  | INTEGER    | number of selected nodes (optional)   |

---

#### Output(s):

|                           |              |                                  |
|---------------------------|--------------|----------------------------------|
| <code>nodCoordLst</code>  | INTEGER(:,:) | list of coordinates of the nodes |
| <code>nNodCoordLst</code> | INTEGER      | number of nodes in the list      |

---

```

3053 SUBROUTINE MakeNodCoordLst (nodCoordLst, nNodCoordLst, nodState,
3054 & nodLst, nNodLst)
3055
3056 REAL*8, ALLOCATABLE, DIMENSION(:,,:), INTENT(OUT) :: nodCoordLst
3057 INTEGER, INTENT(OUT) :: nNodCoordLst
3058 INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: nodLst
3059 INTEGER, INTENT(IN), OPTIONAL :: nNodLst
3060 INTEGER, INTENT(IN), OPTIONAL :: nodState
3061
3062 INTEGER :: nodID, intNodID, i, ns
3063
3064 IF (Present(nodState)) THEN
3065   ns = nodState
3066 ELSE
3067   ns = 1
3068 END IF
3069
3070 IF (Present(nodLst) .AND. Present(nNodLst)) THEN
3071   ALLOCATE (nodCoordLst(nNodLst,3))
3072   DO i = 1, nNodLst
3073     nodID = nodLst(i)
3074     nodCoordLst(:,i) = GetNodCoord (nodID, ns)
3075   END DO
3076   nNodCoordLst = nNodLst
3077 ELSE
3078   ALLOCATE (nodCoordLst(numnp,3))
3079   DO intNodID = 1, numnp
3080     nodID = GetNodExtID(intNodID)
3081     nodCoordLst(:, intNodID) = GetNodCoord (nodID, ns)
3082   END DO
3083   nNodCoordLst = numnp
3084 END IF
3085 RETURN
3086 END SUBROUTINE MakeNodCoordLst

```

This subroutine makes a list of coordinates for all the nodes in the model or alternatively, for the list of nodes provided by the nodLst array. The GetNodCoord function is used to obtain the coordinates of a node. Depending on the optional nodState flag, the original or the deformed state coordinates are returned. If the flag is not set, the original state coordinates will be returned.

### 5.4.29 *MakeNodIDLst*

This subroutine makes a list of all node user IDs in a model.

**Input(s):**

none

**Output(s):**

nodLst      INTEGER(:)      list of the node user IDs

```

2869           SUBROUTINE MakeNodIDLst (nodIDLst)
2870            INTEGER, ALLOCATABLE, DIMENSION(:), INTENT(OUT) :: nodIDLst
2871
2872            INTEGER :: i
2873
2874            ALLOCATE (nodIDLst(numnp))
2875            nodIDLst = [(GetNodExtID(i), i = 1, numnp)]
2876            RETURN
2877            END SUBROUTINE MakeNodIDLst
    
```

This subroutine makes a list of all the nodes in the model using the GetNodExtID function. The nodIDLst allocatable array is used to hold the elements of this list.

### 5.4.30 *MakeNodValIPLst*

This subroutine makes a list of the equivalent value of an elemental quantity on the neighboring node. If no nodes are specified, this is done for all of the nodes in the model.

**Input(s):**

|         |         |                               |
|---------|---------|-------------------------------|
| elmCode | INTEGER | element post code             |
|         |         | Example elemental post codes: |
|         |         | 1-6    Components of strain   |
|         |         | 11-16 Components of stress    |
|         |         | 17    von Mises stress        |

|                   |              |   |
|-------------------|--------------|---|
| calcMeth          | INTEGER      | calculation method<br>Possible methods:<br>1 Translation (unweighted averaging)<br>2 Extrapolation (unweighted averaging)<br>3 Average (unweighted averaging)<br>4 Translation (weighted averaging)<br>5 Extrapolation (weighted averaging)<br>6 Average (weighted averaging) |
| nodLst            | INTEGER(:,:) | list of selected nodes (optional)   |
| nNodLst           | INTEGER      | number of selected nodes (optional)   |
| <b>Output(s):</b> |              |   |
| nodValIPLst       | INTEGER(:)   | list of projected nodal values  |
| nNodValIPLst      | INTEGER      | number of values in the list  |

```

3221 SUBROUTINE MakeIPValLst (elmCode, IPValLst, nIPValLst,
3222 &
3223     INTEGER, INTENT(IN) :: elmCode
3224     REAL*8, ALLOCATABLE, DIMENSION(:,:), INTENT(OUT) :: IPValLst
3225     INTEGER, INTENT(OUT) :: nIPValLst
3226     INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: elmIDLst
3227     INTEGER, INTENT(IN), OPTIONAL :: nElmIDLst
3228
3229     INTEGER :: elmID, intElmID, i, nIP, IP
3230
3231
3232     IF (Present(elmIDLst) .AND. Present(nElmIDLst)) THEN
3233         ALLOCATE (IPValLst(nElmIDLst, nintbmx))
3234         IPValLst = 0.D0
3235
3236         DO i = 1, nElmIDLst
3237             elmID = elmIDLst(i)
3238             nIP = GetElmIPCount(elmID)
3239             DO IP = 1, nIP
3240                 IPValLst(i, IP) = GetIPVal(elmID, IP, elmCode)
3241             END DO
3242         END DO
3243
3244         nIPValLst = nElmIDLst
3245     ELSE
3246         ALLOCATE (IPValLst(numel, nintbmx))
3247         IPValLst = 0.D0
3248
3249         DO intElmID = 1, numel
3250             elmID = GetElmExtID(intElmID)
3251             nIP = GetElmIPCount(elmID)
3252             DO IP = 1, nIP
3253                 IPValLst(intElmID, IP) = GetIPVal(elmID, IP, elmCode)
3254             END DO
3255         END DO
3256
3257         nIPValLst = numel
3258     END IF
3259     RETURN
3260 END SUBROUTINE MakeIPValLst

```

This subroutine calculates the elemental quantities in the nodes and returns them as a list of values. It uses the `GetNodIPVal` to do the calculations for each node. If no node list is specified, it is done for all the nodes of the model. The elemental quantity and

the calculation method are selected using the `elmCode` and the `calcMeth` parameters, respectively.

### 5.4.31 *MakeNodValLst*

This subroutine makes a list of nodal values for all the nodes of the model or the specified list of nodes.

---

**Input(s):**

|                      |            |   |
|----------------------|------------|---|
| <code>nodCode</code> | INTEGER    | nodal post code<br>Example node post codes:<br>0 Coordinates<br>1 Displacement<br>2 Rotation<br>3 External force<br>4 External moment<br>5 Reaction force<br>6 Reaction moment<br>79 Total displacement |
| <code>nodLst</code>  | INTEGER(*) | list of nodes   |
| <code>nNodLst</code> | INTEGER    | number of nodes in the list   |

---

**Output(s):**

|                      |              |                                     |
|----------------------|--------------|-------------------------------------|
| <code>valLst</code>  | REAL*8(:, :) | list of values                      |
| <code>nValLst</code> | INTEGER      | number of values in the list        |
| <code>nComp</code>   | INTEGER      | number of components for each value |

---

```

405     SUBROUTINE MakeNodValLst (nodCode, valLst, nValLst, nComp,
406 &                               nodLst, nNodLst)
407
408     INTEGER, INTENT(IN) :: nodCode
409     INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: nodLst
410     INTEGER, INTENT(IN), OPTIONAL :: nNodLst
411     REAL*8, ALLOCATABLE, DIMENSION(:, :), INTENT(OUT) :: valLst
412     INTEGER, INTENT(OUT) :: nValLst, nComp
413
414     INTEGER :: nodID, intNodID, nTempValLst, i
415     REAL*8, ALLOCATABLE, DIMENSION(:) :: tempValLst
416
417     IF (Present(nodLst) .AND. Present(nNodLst)) THEN
418     DO i = 1, nNodLst
419         nodID = nodLst(i)
420         CALL CalcNodVal (nodID, nodCode, tempValLst, nTempValLst)
421
422         IF (nTempValLst .EQ. 0) THEN
423             CALL QUIT(1234)
424         ELSE
425             IF (Allocated(valLst) .EQV. .FALSE.)
426 &               ALLOCATE (valLst(nNodLst, nTempValLst))
427                 valLst(i, :) = tempValLst(:)
428             END IF
429     END DO

```

```

430      nComp = nTempValLst
431      nValLst = nNodLst
432      ELSE
433      DO intNodID = 1, numnp
434      nodID = GetNodExtID(intNodID)
435
436      CALL CalcNodVal (nodID, nodCode, tempValLst, nTempValLst)
437
438      IF (nTempValLst .EQ. 0) THEN
439      CALL QUIT(1234)
440      ELSE
441      IF (Allocated(valLst) .EQV. .FALSE.)
442      &      ALLOCATE(valLst(numnp, nTempValLst))
443      valLst(intNodID,:) = tempValLst(:)
444      END IF
445      END DO
446      nComp = nTempValLst
447      nValLst = numnp
448      END IF
449      RETURN
450
451      END SUBROUTINE MakeNodValLst

```

This subroutine creates a list of nodal values. Each nodal value is obtained using the CalcNodVal subroutine. If no node list is specified, the list will be comprised of all the nodes of the model.

### 5.4.32 *PrintElmIDGroupedLst*

This subroutine lists all of the elements of the model with the corresponding element groups.

---

#### Input(s):

elmID          INTEGER      external element ID

---

#### Output(s):

INTEGER      internal element ID

---

```

1970      SUBROUTINE PrintElmIDGroupedLst (outUnit)
1971      INTEGER, OPTIONAL, INTENT(IN) :: outUnit
1972
1973      INTEGER :: i, j, intElID, intElType, nElInGroup, ou
1974
1975      100      FORMAT (A30,1X,I4)
1976
1977      IF (Present(outUnit)) THEN
1978      ou = outUnit
1979      ELSE
1980      ou = 6
1981      END IF
1982
1983      WRITE (ou,*) '** PrintElmIDGroupedLst SUBROUTINE'
1984      WRITE (ou,100) 'No. of Element Group(s): ', nElGroups
1985      WRITE (ou,*)
1986      DO i = 1, nElGroups
1987      CALL Setup_EIGroups(i, nElInGroup, 0, 0, 0)
1988      WRITE (ou,100) 'Element Group No.:', i
1989      WRITE (ou,100) 'No. of nodes per element:', nnode
1990      WRITE (ou,100) 'No. of IPs per element:', intel

```

```

1991      WRITE (ou,100) 'Element class:', lclass
1992      WRITE (ou,100) 'Internal element type:', ityp
1993      WRITE (ou,100) 'Element type:', jtype
1994      WRITE (ou,100) 'No. of coordinates:', ncrdel
1995      WRITE (ou,100) 'No. of DOFs per node:', ndeg
1996      WRITE (ou,100) 'No. of DOFs:', ndegel
1997      WRITE (ou,100) 'No. of elements in the group:', nElInGroup
1998      WRITE (ou,*) 'List of elements in the group: '
1999      DO j = 1, nElInGroup
2000          intElID = iElGroup_EIEnum(j)
2001          intElType = iElType(intElID)
2002          WRITE (ou,100) 'Internal ID:', intElID
2003          WRITE (ou,100) 'User ID:', GetElmExtID(intElID)
2004          WRITE (ou,100) 'Internal Type:', intElType
2005      END DO
2006      WRITE (ou,*)
2007  END DO
2008  END SUBROUTINE PrintElmIDGroupedLst

```

This subroutine prints all the element groups and their element IDs. The Setup\_EIGroups utility subroutine is used to set the current element group (see Sect. 2.3.3). Additionally, some other properties of the elements are printed, e.g. the element class (lClass), number of integration points (intEl) etc.

### 5.4.33 *PrintElmIDLst*

This subroutine prints the list of user element IDs.

---

#### Input(s):

outUnit    INTEGER    output unit (optional)

---

#### Output(s):

none

---

```

1902      SUBROUTINE PrintElmIDLst (outUnit)
1903          INTEGER, OPTIONAL, INTENT(IN) :: outUnit
1904
1905          INTEGER, ALLOCATABLE, DIMENSION(:) :: elmIDLst
1906          INTEGER :: ou, i
1907
1908      100      FORMAT (A19,1X,A19)
1909      200      FORMAT (I19,1X,I19)
1910
1911
1912      IF (Present (outUnit)) THEN
1913          ou = outUnit
1914      ELSE
1915          ou = 6
1916      END IF
1917
1918      CALL MakeElmIDLst (elmIDLst)
1919
1920      DO i = 1, numel
1921          WRITE (ou,100) 'Element Internal ID', 'Element External ID'
1922          WRITE (ou,200) i, elmIDLst(i)
1923      END DO
1924      RETURN
1925  END SUBROUTINE PrintElmIDLst

```

This subroutine prints the list of user element IDs.



### 5.4.34 *PrintIPCoordLst*

This subroutine prints the coordinates of the integration points for either all elements or a selected list of elements.

---

#### Input(s):

|           |            |   |
|-----------|------------|---|
| outUnit   | INTEGER    | output unit (optional)                      |
| elmIDLst  | INTEGER(:) | list of external/user element ID (optional) |
| nElmIDLst | INTEGER    | number of elements in the list (optional)   |

---

#### Output(s):

none

---

```

2152 SUBROUTINE PrintIPCoordLst (outUnit, elmIDLst, nElmIDLst)
2153   INTEGER, INTENT(IN), OPTIONAL :: outUnit
2154   INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: elmIDLst
2155   INTEGER, INTENT(IN), OPTIONAL :: nElmIDLst
2156
2157   INTEGER :: ou, elmID, intElmID, IP, nIP, nIPCoordLst, i
2158   REAL*8, ALLOCATABLE, DIMENSION(:,:,:) :: IPCoordLst
2159
2160 100   FORMAT (A7,X,A2,X,3(A15,X))
2161 200   FORMAT (I7,X,I2,X,3(F15.4,X))
2162 300   FORMAT (59(' - '))
2163
2164   IF (Present (outUnit)) THEN
2165     ou = outUnit
2166   ELSE
2167     ou = 6
2168   END IF
2169
2170   WRITE (ou,*) 'Integration Point Coordinates'
2171   WRITE (ou,300)
2172   WRITE (ou,100) 'Element', 'IP', 'X', 'Y', 'Z'
2173   WRITE (ou,300)
2174
2175   IF (Present (elmIDLst) .AND. Present (nElmIDLst)) THEN
2176     CALL MakeIPCoordLst (IPCoordLst, nIPCoordLst,
2177 &      elmIDLst, nElmIDLst)
2178
2179     DO i = 1, nIPCoordLst
2180       elmID = elmIDLst(i)
2181       nIP = GetElmIPCount (elmID)
2182       DO IP = 1, nIP
2183         WRITE (ou, 200) elmID, IP, IPCoordLst(i,IP,:)
2184       END DO
2185       WRITE (ou,300)
2186     END DO
2187   ELSE
2188
2189     CALL MakeIPCoordLst (IPCoordLst, nIPCoordLst)
2190
2191     DO intElmID = 1, nIPCoordLst
2192       elmID = GetElmExtID (intElmID)
2193       nIP = GetElmIPCount (elmID)
2194       DO IP = 1, nIP
2195         WRITE (ou, 200) elmID, IP, IPCoordLst(intElmID, IP,:)
2196       END DO
2197       WRITE (ou,300)
2198     END DO
2199   END IF
2200 END SUBROUTINE PrintIPCoordLst

```

This subroutine prints the coordinates of the integration points. This can be done either for all the elements of the model or for a selected list of elements (elmIDLst).

### 5.4.35 PrintIPValLst

This subroutine prints the elemental quantities in the integration points.

---

|                  |            |   |
|------------------|------------|---|
| <b>Input(s):</b> |            |   |
| elmCode          | INTEGER    | element post code<br>Example elemental post codes:<br>1-6 Components of strain<br>11-16 Components of stress<br>17 von Mises stress |
| outUnit          | INTEGER    | output unit (optional)  |
| elmIDLst         | INTEGER(:) | list of external/user element ID (optional)   |
| nElmIDLst        | INTEGER    | number of elements in the list (optional)   |

---

|                   |  |  |
|-------------------|--|--|
| <b>Output(s):</b> |  |  |
| none              |  |  |

---

```

2054 SUBROUTINE PrintIPValLst (elmCode, outUnit, elmIDLst, nElmIDLst)
2055     INTEGER, INTENT(IN) :: elmCode
2056     INTEGER, INTENT(IN), OPTIONAL :: outUnit
2057     INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: elmIDLst
2058     INTEGER, INTENT(IN), OPTIONAL :: nElmIDLst
2059
2060     INTEGER :: ou, i, elmID, intElmID, IP, nIP, nIPValLst
2061     REAL*8, ALLOCATABLE, DIMENSION(:,:) :: IPValLst
2062
2063     100 FORMAT (A13,X,I4)
2064     200 FORMAT (A7,X,A2,X,A16,X)
2065     300 FORMAT (I7,X,I2,X,F16.4,X)
2066     400 FORMAT (28(' - '))
2067
2068     IF (Present (outUnit)) THEN
2069         ou = outUnit
2070     ELSE
2071         ou = 6
2072     END IF
2073
2074     WRITE (ou, 100) 'Increment No.', inc
2075     WRITE (ou, 100) 'Element Post Code', elmCode
2076     WRITE (ou, 400)
2077     WRITE (ou, 200) 'Element', 'IP', 'Value'
2078     WRITE (ou, 400)
2079
2080     IF (Present(elmIDLst) .AND. Present(nElmIDLst)) THEN
2081         CALL MakeIPValLst (elmCode, IPValLst, nIPValLst,
2082             & elmIDLst, nElmIDLst)
2083
2084     DO i = 1, nIPValLst
2085         elmID = elmIDLst(i)
2086         nIP = GetElmIPCount (elmID)
2087         DO IP = 1, nIP
2088             WRITE (ou, 300) elmID,IP, IPValLst(i,IP)
2089         END DO
    
```

```

2090         WRITE (ou,400)
2091     END DO
2092 ELSE
2093     CALL MakeIPValLst (elmCode, IPValLst, nIPValLst)
2094     DO intElmID = 1, nIPValLst
2095         elmID = GetElmExtID (intElmID)
2096         nIP = GetElmIPCount (elmID)
2097         DO IP = 1, nIP
2098             WRITE (ou, 300) elmID, IP, IPValLst(intElmID, IP)
2099         END DO
2100     WRITE (ou,400)
2101 END DO
2102 END IF
2103
2104 END SUBROUTINE PrintIPValLst

```

This subroutine prints the elemental quantities in the integration points of elements. The elements can be a selected list (elmIDLst) or otherwise, all the elements of the model are considered. The output unit is also optional and its default value is 6.

### 5.4.36 *PrintNodCoordLst*

This subroutine prints the coordinates of the nodes.

---

#### Input(s):

|          |            |   |
|----------|------------|---|
| nodState | INTEGER    | the state of the node (optional):<br>1 undeformed state (default)<br>2 deformed state |
| outUnit  | INTEGER    | output unit (optional)  |
| nodLst   | INTEGER(*) | list of nodes   |
| nNodLst  | INTEGER    | number of nodes in the list   |

---

#### Output(s):

none

---

```

2235 SUBROUTINE PrintNodCoordLst (nodState, outUnit, nodLst, nNodLst)
2236     INTEGER, INTENT(IN), OPTIONAL :: nodState
2237     INTEGER, INTENT(IN), OPTIONAL :: outUnit
2238     INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: nodLst
2239     INTEGER, INTENT(IN), OPTIONAL :: nNodLst
2240
2241     INTEGER :: i, ou, ns, nodID, nNodCoordLst
2242     REAL*8, ALLOCATABLE, DIMENSION(:,:) :: nodCoordLst
2243
2244 100     FORMAT (A10,X,3(A15,X))
2245 200     FORMAT (I10,X,3(F15.4,X))
2246 300     FORMAT (59(' - '))
2247
2248
2249     IF (Present (outUnit)) THEN
2250         ou = outUnit
2251     ELSE
2252         ou = 6
2253     END IF
2254
2255     IF (Present (nodState)) THEN

```

```

2256         ns = nodState
2257     ELSE
2258         ns = 1
2259     END IF
2260
2261     IF (ns .EQ. 1) THEN
2262         WRITE(ou,*) 'Original Nodal Coordinates '
2263     ELSE
2264         WRITE(ou,*) 'Deformed Nodal Coordinates '
2265     END IF
2266
2267     WRITE (ou,300)
2268     WRITE (ou,100) 'Node ID', 'X', 'Y', 'Z'
2269     WRITE (ou,300)
2270
2271     IF (Present(nodLst) .AND. Present(nNodLst)) THEN
2272         CALL MakeNodCoordLst(nodCoordLst, nNodCoordLst, ns,
2273 &                             nodLst, nNodLst)
2274
2275         DO i = 1, nNodCoordLst
2276             nodID = nodLst(i)
2277             WRITE (ou, 200) nodID, nodCoordLst(:,i)
2278         END DO
2279     ELSE
2280
2281         CALL MakeNodCoordLst(nodCoordLst, nNodCoordLst, ns)
2282
2283         DO i = 1, nNodCoordLst
2284             nodID = GetNodExtID(i)
2285             WRITE (ou, 200) nodID, nodCoordLst(:,i)
2286         END DO
2287
2288     END IF
2289     WRITE (ou,300)
2290 END SUBROUTINE PrintNodCoordLst

```

This subroutine uses the `MakeNodCoordLst` subroutine to make a list of all the nodes in the model and prints their coordinates. Alternatively, the same procedure can be done for a selected group of nodes (`nodLst`). The coordinates can be selected via the `nodState` flag to be the original or deformed ones. The output unit is unit 6 by default but other file units can be used as the output as well.

### 5.4.37 *PrintNodIDLst*

This subroutine prints all of the nodes of the model along with their essential properties.

---

#### Input(s):

|         |         |                                       |
|---------|---------|---------------------------------------|
| setID   | INTEGER | ID of the set                         |
| outUnit | INTEGER | output unit (optional)<br>default = 6 |

---

#### Output(s):

none

---

```

2322 SUBROUTINE PrintNodIDLst ()
2323   INTEGER :: i
2324 100   FORMAT (A35, I4)
2325
2326   WRITE (6,*) '** PrintNodIDLst SUBROUTINE'
2327   WRITE (6,100) 'Total no. of nodes:', numnp
2328   WRITE (6,100) 'Max. No. of DoF per node:', ndegmx
2329   WRITE (6,100) 'Max. No. of nodes per element:', nnodmx
2330   WRITE (6,100) 'Max. No. of coordinates per node:', ncrdmx
2331   WRITE (6,100) 'Max. No. of connections to a node:', maxnp
2332   DO i = 1, numnp
2333     WRITE (6,100) 'Internal node ID:', i
2334     WRITE (6,100) 'User node ID:', GetNodExtID(i)
2335   END DO
2336 END SUBROUTINE PrintNodIDLst

```

This subroutine lists all the nodes of the model along with some respective properties, e.g. the maximum number of degrees of freedom (ndegmx), maximum number of nodes per element (nnodmx). The subroutine can be used for debugging purposes or to store the output for later uses.

### 5.4.38 *PrintNodValIPLst*

This subroutine prints all the nodal values obtained from the integration points.

---

#### Input(s):

|         |            |   |
|---------|------------|---|
| elmCode | INTEGER    | element post code<br>Example elemental post codes:<br>1-6 Components of strain<br>11-16 Components of stress<br>17 von Mises stress |
| outUnit | INTEGER    | output unit (optional)<br>default = 6   |
| nodLst  | INTEGER(:) | list of node user IDs   |
| nNodLst | INTEGER    | number of nodes in the list   |

---

#### Output(s):

none

---

```

2597 SUBROUTINE PrintNodValIPLst (elmCode, outUnit, nodLst, nNodLst)
2598   INTEGER, INTENT(IN) :: elmCode
2599   INTEGER, INTENT(IN), OPTIONAL :: outUnit
2600   INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: nodLst
2601   INTEGER, INTENT(IN), OPTIONAL :: nNodLst
2602
2603   INTEGER :: ou, nodID, i, nVal
2604   REAL*8, ALLOCATABLE, DIMENSION(:) :: uTransVal, uExtraVal,
2605   & uAveraVal, wTransVal, wExtraVal, wAveraVal
2606
2607 100   FORMAT (A13, X, I4)
2608 200   FORMAT (A7, X, 6(A16, X))
2609 400   FORMAT (110(' - '))
2610 500   FORMAT (8X, A50, 1X, A50)
2611
2612   IF (Present(outUnit)) THEN
2613     ou = outUnit

```

```

2614 ELSE
2615     ou = 6
2616 END IF
2617
2618 WRITE (ou, 100) 'Increment No.', inc
2619 WRITE (ou, 100) 'Element Post Code', elmCode
2620 WRITE (ou, 400)
2621 WRITE (ou, 500) 'Unweighted', 'Weighted'
2622 WRITE (ou, 200) 'Node', 'Translated', 'Extrapolated', 'Average',
2623 & 'Translated', 'Extrapolated', 'Average'
2624 WRITE (ou, 400)
2625
2626 IF (Present(nodLst) .AND. Present(nNodLst)) THEN
2627     CALL MakeNodVallPLst (elmCode, 1,
2628 &     uTransVal, nVal, nodLst, nNodLst)
2629     CALL MakeNodVallPLst (elmCode, 2,
2630 &     uExtraVal, nVal, nodLst, nNodLst)
2631     CALL MakeNodVallPLst (elmCode, 3,
2632 &     uAveraVal, nVal, nodLst, nNodLst)
2633     CALL MakeNodVallPLst (elmCode, 4,
2634 &     wTransVal, nVal, nodLst, nNodLst)
2635     CALL MakeNodVallPLst (elmCode, 5,
2636 &     wExtraVal, nVal, nodLst, nNodLst)
2637     CALL MakeNodVallPLst (elmCode, 6,
2638 &     wAveraVal, nVal, nodLst, nNodLst)
2639
2640     DO i = 1, nVal
2641         nodID = nodLst(i)
2642         CALL PrintValue
2643     END DO
2644
2645 ELSE
2646
2647     CALL MakeNodVallPLst (elmCode, 1, uTransVal, nVal)
2648     CALL MakeNodVallPLst (elmCode, 2, uExtraVal, nVal)
2649     CALL MakeNodVallPLst (elmCode, 3, uAveraVal, nVal)
2650     CALL MakeNodVallPLst (elmCode, 4, wTransVal, nVal)
2651     CALL MakeNodVallPLst (elmCode, 5, wExtraVal, nVal)
2652     CALL MakeNodVallPLst (elmCode, 6, wAveraVal, nVal)
2653
2654     DO i = 1, nVal
2655         nodID = GetNodExtID(i)
2656         CALL PrintValue
2657     END DO
2658 END IF
2659
2660 WRITE (ou, 400)
2661 RETURN
2662
2663 CONTAINS
2664
2665 SUBROUTINE PrintValue ()
300     FORMAT (17 ,X,6(F16.4,X))
2667
2668     WRITE (ou, 300) nodID, uTransVal(i), uExtraVal(i),
2669 &     uAveraVal(i), wTransVal(i), wExtraVal(i), wAveraVal(i)
2670     END SUBROUTINE PrintValue
2671
2672 END SUBROUTINE PrintNodVallPLst

```

This subroutine prints all the elemental values which are evaluated in the nodes using various methods. The output of all the methods used in the MakeNodVallPLst subroutine are obtained and printed in the output unit. The default value for the output is 6.

### 5.4.39 *PrintNodValLst*

This subroutine prints the nodal quantities of the nodes.

---

#### Input(s):

|         |            |   |
|---------|------------|---|
| nodCode | INTEGER    | nodal post code<br>Example node post codes:<br>0 Coordinates<br>1 Displacement<br>2 Rotation<br>3 External force<br>4 External moment<br>5 Reaction force<br>6 Reaction moment<br>79 Total displacement |
| outUnit | INTEGER    | output unit (optional)<br>default = 6   |
| nodLst  | INTEGER(:) | list of node user IDs   |
| nNodLst | INTEGER    | number of nodes in the list   |

---

#### Output(s):

none

---

```

2488 SUBROUTINE PrintNodValLst (nodCode, nodLst, nNodLst, outUnit)
2489   INTEGER, INTENT(IN) :: nodCode
2490   INTEGER, DIMENSION(*), INTENT(IN), OPTIONAL :: nodLst
2491   INTEGER, INTENT(IN), OPTIONAL :: nNodLst, outUnit
2492
2493   INTEGER :: nodID, nValLst, i, j, nComp, ou
2494   REAL*8, ALLOCATABLE, DIMENSION(:,:) :: valLst
2495
2496 100   FORMAT (A15,X,I2)
2497 500   FORMAT (I7,X,I10(F16.4,X,:))
2498
2499   IF (Present (outUnit)) THEN
2500     ou = outUnit
2501   ELSE
2502     ou = 6
2503   END IF
2504
2505   WRITE (ou, 100) 'Increment No.', inc
2506   WRITE (ou, 100) 'Node Post Code', NodCode
2507
2508   IF (Present(nodLst) .AND. Present(nNodLst)) THEN
2509     CALL MakeNodValLst (nodCode, valLst, nValLst, nComp,
2510                       & nodLst, nNodLst)
2511     CALL PrintTableHeader
2512     DO i = 1, nValLst
2513       WRITE (ou, 500) nodLst(i), [(valLst(i,j), j = 1, nComp)]
2514     END DO
2515   ELSE
2516     CALL MakeNodValLst (nodCode, valLst, nValLst, nComp)
2517     CALL PrintTableHeader
2518     DO i = 1, nValLst
2519       WRITE (ou, 500) GetNodExtID(i),[(valLst(i,j), j = 1, nComp)]
2520     END DO
2521   END IF

```

```

2522      CALL PrintLine
2523      RETURN
2524
2525      CONTAINS
2526
2527      SUBROUTINE PrintTableHeader ( )
2528          INTEGER :: j
2529
2530 200      FORMAT ( 'Node ID ',10(5X, 'Component', I3 ,:))
2531
2532          CALL PrintLine
2533          WRITE (ou, 200) [(j, j = 1, nComp)]
2534          CALL PrintLine
2535
2536      END SUBROUTINE PrintTableHeader
2537
2538      SUBROUTINE PrintLine ( )
2539          INTEGER :: j
2540
2541 400      FORMAT (8(' - '),10(17A1 ,:))
2542
2543          WRITE (ou, 400) [( ' - ', j = 1, nComp*17)]
2544      END SUBROUTINE PrintLine
2545
2546      END SUBROUTINE PrintNodValLst

```

This subroutine uses the MakeNodValLst subroutine to create a list of nodal values and prints them. The PrintTableHeader and PrintLine subroutines are used to print the header of the table and horizontal lines, respectively.

### 5.4.40 *PrintSetItemLstID*

This subroutine prints the information regarding a specific set ID to a file.

---

**Input(s):**

|         |         |                                       |
|---------|---------|---------------------------------------|
| setID   | INTEGER | ID of the set                         |
| outUnit | INTEGER | output unit (optional)<br>default = 6 |

---

**Output(s):**

none

---

```

2700      SUBROUTINE PrintSetItemLstID (setID, outUnit)
2701          INTEGER, INTENT(IN) :: setID
2702          INTEGER, INTENT(IN), OPTIONAL :: outUnit
2703
2704          INTEGER :: ou
2705
2706 100      FORMAT (A28, 1X, I32)
2707 101      FORMAT (A28, 1X, L32)
2708 102      FORMAT (A28, 1x, A32)
2709
2710          IF (Present (outUnit)) THEN
2711              ou = outUnit
2712          ELSE
2713              ou = 6
2714          END IF
2715

```



```

2716 WRITE (ou,*) '** PrintSetItemLstID SUBROUTINE'
2717 WRITE (ou,100) 'Set no.:', setID
2718 WRITE (ou,102) 'Set name:', setnam(setid)
2719 WRITE (ou,100) 'Set type:', isetdat(1,setid)
2720 WRITE (ou,100) 'No. of item(s):', isetdat(2,setid)
2721 WRITE (ou,101) 'Sorted list:', isetdat(3,setid) .EQ. 1
2722 WRITE (ou,101) 'B.C. flag:', isetdat(4,setid) .EQ. 1
2723 WRITE (ou,100) 'No. of item(s) (full mode):', isetdat(5,setid)
2724 WRITE (ou,101) 'Updates in remeshing:', isetdat(6,setid) .EQ. 1
2725 WRITE (ou,*)
2726 RETURN
2727 END SUBROUTINE PrintSetItemLstID

```

The PrintSetItemLstID subroutine prints the various information regarding a set to an output file. The information is extracted from the spaceset common block of MARC (see Table 2.8). The optional outUnit argument is used to indicate the output of the information. By default, the unit number 6 is used if no output unit is specified. The same approach is used in other subroutines which handle a printing task.

#### 5.4.41 *PrintSetItemLstName*

This subroutine prints the information regarding a specific set name to a file.

---

##### Input(s):

|         |              |                                       |
|---------|--------------|---------------------------------------|
| setName | CHARACTER(*) | the name of the set                   |
| outUnit | INTEGER      | output unit (optional)<br>default = 6 |

---

##### Output(s):

none

---

```

2758 SUBROUTINE PrintSetItemLstName (setName, outUnit)
2759 CHARACTER(LEN=*) , INTENT(IN) :: setName
2760 INTEGER, INTENT(IN), OPTIONAL :: outUnit
2761
2762 INTEGER :: i, ou
2763 LOGICAL :: notFound
2764
2765 IF (Present (outUnit)) THEN
2766   ou = outUnit
2767 ELSE
2768   ou = 6
2769 END IF
2770
2771 notFound = .TRUE.
2772 i = 1
2773 DO WHILE ((i .LE. ndSet) .AND. (notFound))
2774   IF (setName .EQ. setNam(i)) then
2775     CALL PrintSetItemLstID (i, ou)
2776     notFound = .FALSE.
2777   ELSE
2778     i = i + 1
2779   END IF
2780 END DO
2781 IF (notFound .EQV. .TRUE.) WRITE(ou,*) setName, ' is not found.'
2782 RETURN
2783 END SUBROUTINE PrintSetItemLstName

```

The PrintSetByName subroutine uses a DO WHILE loop to find a set by its name. Then, it prints the details of the set by using the PrintSetItemLstID subroutine. Note that a set name is a variable of type CHARACTER with a length of 32. However, in this listing, an assumed size character is used to cover every length of the setNam argument.

### 5.4.42 PrintSetLst

This subroutine prints the general information regarding all the sets of the model to a file.

---

**Input(s):**

|         |              |                                       |
|---------|--------------|---------------------------------------|
| setName | CHARACTER(*) | the name of the set                   |
| outUnit | INTEGER      | output unit (optional)<br>default = 6 |

---

**Output(s):**

none

---

2813  
2814  
2815  
2816  
2817  
2818  
2819  
2820  
2821  
2822  
2823  
2824  
2825  
2826  
2827  
2828  
2829  
2830  
2831  
2832  
2833  
2834

```

SUBROUTINE PrintSetLst (outUnit)
  INTEGER, INTENT(IN), OPTIONAL :: outUnit

  INTEGER :: i, ou
100  FORMAT (A27,1X,I4)

  IF (Present (outUnit)) THEN
    ou = outUnit
  ELSE
    ou = 6
  END IF

  WRITE (ou,100) 'Total sets:', ndset
  WRITE (ou,100) 'Max. no. of sets:', nsetmx
  WRITE (ou,100) 'Max. set name characters:', nchnam
  WRITE (ou,100) 'Max. no. of set items:', mxitmsset
  WRITE (ou,*)
  DO i = 1, ndset
    CALL PrintSetItemLstID (i, ou)
  END DO
  RETURN
END SUBROUTINE PrintSetLst
    
```

The PrintSetLst subroutine prints the general information regarding the sets of the model to a file. The output file is optional and if it is not specified, the default value of 6 is used. The information is extracted from the spaceset MARC common block (see Table 2.8). Then, the specific data for each set of the model is printed using the PrintSetItemLstID subroutine.

## 5.5 FileTools Module

### 5.5.1 AutoFilename

This subroutine updates the chosen name of an existing file name.

---

**Input(s):**

fn CHARACTER(\*) file name

---

**Output(s):**

fn CHARACTER(\*) file name

---

59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80

```

SUBROUTINE AutoFilename (fName)
  CHARACTER (LEN=*), INTENT(INOUT) :: fName
  CHARACTER (LEN = LEN(fName)+8) :: tempfName
  CHARACTER (LEN = 3) :: digit
  INTEGER :: i, j
  LOGICAL :: fileExist
90   FORMAT (i3.3)
  fileExist = .TRUE.
  i = 0
  DO WHILE (fileExist .EQV. .TRUE.)
    i = i + 1
    WRITE (digit, 90) i
    tempfName = trim(fName)// '_'// digit// '.txt'
  INQUIRE (FILE = tempfName, EXIST=fileExist)
  END DO
  fName = tempfName
  RETURN
END SUBROUTINE AutoFilename

```

The AutoFilename subroutine provides the user with a mechanism to prevent overwriting the output files. This case usually happens when the result of a run is not saved elsewhere and then rerunning the corresponding job overwrites the old results. This subroutine avoids overwriting the existing files by adding a unique 3-digit trailing number to the file name.

### 5.5.2 FindFreeUnit

This subroutine returns the first free file unit.

---

**Input(s):**

none

---

**Output(s):**

fn CHARACTER(\*) file name

---

```

18      SUBROUTINE FindFreeUnit (fName)
19          INTEGER, INTENT(OUT) :: fName
20          INTEGER, PARAMETER :: MINUNIT = 10, MAXUNIT = 200
21          INTEGER :: i
22          LOGICAL :: uCon, found
23
24          found = .FALSE.
25          i = MINUNIT
26          DO WHILE ((i .LT. MAXUNIT) .AND. (found .EQV. .FALSE.))
27              INQUIRE (UNIT = i, OPENED = uCon)
28              IF (uCon .EQV. .FALSE.) THEN
29                  found = .TRUE.
30              ELSE
31                  i = i + 1
32              END IF
33          END DO
34          IF ((i .EQ. MAXUNIT) .AND. (found .EQV. .FALSE.)) THEN
35              fName = -1
36          ELSE
37              fName = i
38          END IF
39      END SUBROUTINE FindFreeUnit

```

The FindFreeUnit subroutine assigns the first free file unit number to its only argument. It starts checking from the unit number MINUNIT up to MAXUNIT to find a free one. If the search was unsuccessful, the subroutine returns -1 to state the error. With this simple subroutine, any attempts to connect an already engaged file unit is easily prevented. Such attempts will result in the Exit Message 7001 which indicates a program crash due to the FORTRAN code. In this case, the INTEL® compiler issues a severe error 40 with the following message:

```
forrtl: severe (40): recursive I/O operation, unit 20, file unknown
```

The message indicates that the FORTRAN code is trying to connect a file unit, unit 20 in this case, which is already occupied in another process.

### 5.5.3 DeleteFile

This subroutine deletes a file.

---

#### Input(s):

fn            CHARACTER(\*)    file name

---

#### Output(s):

none

---

```

43      SUBROUTINE DeleteFile (fName)
44          CHARACTER(LEN=*), INTENT (IN) :: fName
45
46          INTEGER :: fUnit
47          LOGICAL :: fe
48
49          CALL FindFreeUnit (fUnit)
50          INQUIRE (FILE = fName, EXIST=fe)
51          IF (fe .EQV. .TRUE.) THEN
52              OPEN (UNIT = fUnit, FILE = fName, STATUS = 'OLD')

```

```

53      CLOSE (UNIT = fUnit , STATUS = 'DELETE')
54      END IF
55      END SUBROUTINE DeleteFile

```

The DeleteFile subroutine accepts one character argument (fn) containing the name of an external file. The file, if existing, is deleted by the subroutine.

## 5.6 MiscTools Module

### 5.6.1 DelRepeated

This subroutine removes all the recurring items of a list but preserves one instance.

---

#### Input(s):

|          |            |                              |
|----------|------------|------------------------------|
| itemLst  | INTEGER(*) | list of items to be searched |
| nItemLst | INTEGER    | number of items in the list  |

---

#### Output(s):

|             |            |                                    |
|-------------|------------|------------------------------------|
| outItemLst  | INTEGER(:) | the output item list               |
| nOutItemLst | INTEGER    | number of items in the output list |

---

```

135      SUBROUTINE DelRepeated (itemLst , nItemLst , outItemLst , nOutItemLst)
136          INTEGER, DIMENSION(*) , INTENT(IN) :: itemLst
137          INTEGER, INTENT(IN) :: nItemLst
138          INTEGER, DIMENSION(:) , ALLOCATABLE, INTENT(OUT) :: outItemLst
139          INTEGER, INTENT(OUT) :: nOutItemLst
140
141          LOGICAL , ALLOCATABLE, DIMENSION(:) :: lstMask
142          INTEGER, ALLOCATABLE, DIMENSION(:) :: lstIndex
143          INTEGER :: i
144
145          ALLOCATE (lstMask(nItemLst))
146          lstMask = .TRUE.
147
148          DO i = nItemLst , 2 , -1
149              lstMask(i) = .NOT. (Any (itemLst(:i-1) .EQ. itemLst(i)))
150          END DO
151
152          ALLOCATE(lstIndex , source = PACK([(i , i=1 , nItemLst)] , lstMask))
153          ALLOCATE(outItemLst , source = itemLst(lstIndex))
154          nOutItemLst = Size (outItemLst)
155      END SUBROUTINE DelRepeated

```

The DelRepeated subroutine checks the 1D array (itemLst) for repeated items and removes all, keeping only one instance. The refined list is returned via the outItemLst allocatable array. The subroutine uses the Any intrinsic function to create a logical array (lstMask) containing the occurrence status of every item. The final value of the mask will be .FALSE. for a recurring element. In line 18, an index array (lstIndex) is created containing the indices corresponding to the unique elements. The index is used to create the final list.

### 5.6.2 DelRepeated2D

This subroutine removes all the recurring items of a 2D array but preserves one instance.

**Input(s):**

|          |              |                              |
|----------|--------------|------------------------------|
| itemLst  | INTEGER(2,*) | list of items to be searched |
| nItemLst | INTEGER      | number of items in the list  |

**Output(s):**

|             |              |                                    |
|-------------|--------------|------------------------------------|
| outItemLst  | INTEGER(:,:) | the output item list               |
| nOutItemLst | INTEGER      | number of items in the output list |

```

190 SUBROUTINE DelRepeated2D (itemLst, nItemLst,
191 &                          outItemLst, nOutItemLst)
192
193     INTEGER, DIMENSION(2,*), INTENT(IN) :: itemLst
194     INTEGER, INTENT(IN) :: nItemLst
195     INTEGER, DIMENSION(:, :), ALLOCATABLE, INTENT(OUT) :: outItemLst
196     INTEGER, INTENT(OUT) :: nOutItemLst
197
198     LOGICAL, ALLOCATABLE, DIMENSION(:) :: lstMask
199     INTEGER, ALLOCATABLE, DIMENSION(:) :: lstIndex
200     INTEGER :: i
201
202     ALLOCATE (lstMask(nItemLst))
203     lstMask = .TRUE.
204
205     DO i = nItemLst, 2, -1
206         lstMask(i) = .NOT.(Any (itemLst(1,:i-1) .EQ. itemLst(1,i)
207 &                          .AND. itemLst(2,:i-1) .EQ. itemLst(2,i)))
208     END DO
209
210     ALLOCATE(lstIndex, source=PACK([(i, i=1, nItemLst)], lstMask))
211
212     ALLOCATE(outItemLst, source=itemLst(:, lstIndex))
213     nOutItemLst = Size (outItemLst, 2)
214
215 END SUBROUTINE DelRepeated2D

```

The DelRepeated2D subroutine works in a similar way to its 1D version but considers each pair of the 2D arrays.

### 5.6.3 ExtractIntersectLst

Returns the list of intersecting items for two lists (1D arrays).

**Input(s):**

|           |            |                                    |
|-----------|------------|------------------------------------|
| itemLst1  | INTEGER(*) | first list of items                |
| nItemLst1 | INTEGER    | number of items in the first list  |
| itemLst2  | INTEGER(*) | second list of items               |
| nItemLst2 | INTEGER    | number of items in the second list |

**Output(s):**

|           |            |  |
|-----------|------------|--|
| interLst  | INTEGER(:) | the list of intersecting items           |
| nInterLst | INTEGER    | number of items in the intersecting list |

```

76      SUBROUTINE ExtractIntersectLst (itemLst1 , nItemLst1 ,
77      & itemLst2 , nItemLst2 , interLst , nInterLst)
78
79          INTEGER, DIMENSION(*) , INTENT(IN) :: itemLst1 , itemLst2
80          INTEGER, INTENT(IN) :: nItemLst1 , nItemLst2
81          INTEGER, ALLOCATABLE, INTENT(OUT) :: interLst(:)
82          INTEGER, INTENT(OUT) :: nInterLst
83
84          INTEGER :: i , nLst
85          LOGICAL, ALLOCATABLE, DIMENSION(:) :: lstMask
86          INTEGER, ALLOCATABLE, DIMENSION(:) :: tLst
87
88          ALLOCATE (lstMask(nItemLst1))
89
90          DO i = 1, nItemLst1
91              lstMask(i) = Any (itemLst1(i) .EQ. itemLst2(:nItemLst2))
92          END DO
93
94          nLst = Count(lstMask)
95
96          IF (nLst .NE. 0) THEN
97              ALLOCATE (tLst , SOURCE = Pack (itemLst1(1:nItemLst1) ,
98      &          MASK=lstMask))
99              CALL DelRepeated (tLst , nLst , interLst , nInterLst)
100             ELSE
101                 nInterLst = nLst
102             END IF
103
104     END SUBROUTINE ExtractIntersectLst

```

This subroutine selects the intersecting items in two lists. To do so, every item of the first list is compared to all the items of the second list and the Any intrinsic function is used to obtain any .TRUE. values. The recurring items are deleted using the DelRepeated subroutine. Therefore, the output list is created without any recurring items.

### 5.6.4 *GetDistance*

This function returns the distance between two points.

**Input(s):**

|        |           |                                 |
|--------|-----------|---------------------------------|
| pointA | REAL*8(3) | coordinates of the first point  |
| pointB | REAL*8(3) | coordinates of the second point |

**Output(s):**

|        |                             |
|--------|-----------------------------|
| REAL*8 | distance between two points |
|--------|-----------------------------|

```

243     FUNCTION GetDistance (pointA , pointB)
244     REAL*8, DIMENSION(3) , INTENT(IN) :: pointA , pointB
245     REAL*8 :: GetDistance
246
247     GetDistance = SQRT ((pointA(1)-pointB(1))**2.D0 +

```

248  
249  
250

```

&          (pointA(2) - pointB(2))**2.D0 +
&          (pointA(3) - pointB(3))**2.D0)
END FUNCTION GetDistance
    
```

The GetDistance function returns the distance between two geometrical coordinates, i.e. the coordinates of the pointA and pointB points.

### 5.6.5 GetIndex

This function returns the index of an item in a list.

**Input(s):**

---

|            |            |                             |
|------------|------------|-----------------------------|
| itemLst    | INTEGER(*) | list of items               |
| nItemLst   | INTEGER    | number of items in the list |
| searchItem | INTEGER    | item to be searched         |

---

**Output(s):**

|         |  |
|---------|--|
| INTEGER | the index of the searchItem in the itemLst |
|---------|--|

---

282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297

```

FUNCTION GetIndex (itemLst, nItemLst, searchItem)
  INTEGER, INTENT(IN), DIMENSION(*) :: itemLst
  INTEGER, INTENT(IN) :: nItemLst, searchItem
  INTEGER :: GetIndex

  INTEGER :: i

  GetIndex = 0

  DO i = 1, nItemLst
    IF (itemLst(i) .EQ. searchItem) THEN
      GetIndex = i
      EXIT
    END IF
  END DO
END FUNCTION GetIndex
    
```

The GetIndex function returns the index of an item in a 1D array. If the item cannot be found, the function returns a zero.

### 5.6.6 GetRandNum

This function returns a random number between -1 and +1.

**Input(s):**

none

**Output(s):**

---

|        |                 |
|--------|-----------------|
| REAL*8 | a random number |
|--------|-----------------|

---



```

282 FUNCTION GetIndex (itemLst, nItemLst, searchItem)
283   INTEGER, INTENT(IN), DIMENSION(*) :: itemLst
284   INTEGER, INTENT(IN) :: nItemLst, searchItem
285   INTEGER :: GetIndex
286
287   INTEGER :: i
288
289   GetIndex = 0
290
291   DO i = 1, nItemLst
292     IF (itemLst(i) .EQ. searchItem) THEN
293       GetIndex = i
294       EXIT
295     END IF
296   END DO
297 END FUNCTION GetIndex

```

The GetRandNum function generates a random number with a random sign. It combines the random\_number and random\_seed intrinsic functions to produce a random number between 0 and 1. The random\_seed function is required to just run once to generate the first random number. After that, every subsequent number is generated based on the previous one. Two random numbers are generated within the function (randNum1 and randNum2) and their subtraction will be the return value of the function. This is done to generate a random sign as well as the random number.

### 5.6.7 *PrintElapsedTime*

This subroutine prints the elapsed time between its two executions.

---

#### Input(s):

outUnit    INTEGER    output unit (optional)

---

#### Output(s):

none

---

```

449 SUBROUTINE PrintElapsedTime (outUnit)
450   INTEGER, INTENT(IN), OPTIONAL :: outUnit
451
452   INTEGER :: ou
453   REAL :: startTime, stopTime, elapsedTime
454   SAVE :: startTime
455   LOGICAL, SAVE :: firstRun = .TRUE.
456
100  FORMAT (A22, X, F8.3, X, 'seconds')
458
459   IF (Present (outUnit)) THEN
460     ou = outUnit
461   ELSE
462     ou = 6
463   END IF
464
465   IF (firstRun .EQV. .TRUE.) THEN
466     firstRun = .FALSE.
467     CALL Cpu_time (startTime)
468     WRITE (ou, '(A22)') 'Stopwatch started...'
469   ELSE
470     CALL Cpu_time (stopTime)

```

```

471         elapsedTime = stopTime - startTime
472         WRITE (ou, 100) 'Elapsed time:', elapsedTime
473         firstRun = .TRUE.
474     END IF
475 END SUBROUTINE PrintElapsedTime

```

The PrintElapsedTime subroutine is used to calculate the approximate duration of a process. Since the subroutine returns the elapsed time between its two executions, it must be obviously executed twice; first at the beginning of the process and then at its end. It uses the Cpu\_time intrinsic function to record the time at each run and then calculates the elapsed time in seconds. The output of the function is of type character so it can be readily printed in the output. The default value for the output unit is 6.

### 5.6.8 PutSmallFirst

This subroutine compares the pairs of a 2D array, and removes the smaller component to the first position of the array.

---

#### Input(s):

|          |              |                             |
|----------|--------------|-----------------------------|
| itemLst  | INTEGER(2,*) | list of items               |
| nItemLst | INTEGER      | number of items in the list |

---

#### Output(s):

|         |            |                      |
|---------|------------|----------------------|
| itemLst | INTEGER(*) | sorted list of items |
|---------|------------|----------------------|

---

```

507     SUBROUTINE PutSmallFirst (itemLst, nItemLst)
508         INTEGER, DIMENSION(2,*) , INTENT(INOUT) :: itemLst
509         INTEGER, INTENT(IN) :: nItemLst
510
511         INTEGER :: i
512
513         DO i = 1, nItemLst
514             IF (itemLst(1,i) .GT. itemLst(2,i)) THEN
515                 CALL SwapInt (itemLst(1,i), itemLst(2,i))
516             END IF
517         END DO
518     END SUBROUTINE PutSmallFirst

```

This subroutine compares the first and the second component of the array. If the first one is larger, it is replaced by the second one.

### 5.6.9 SwapInt

This subroutine swaps the value of two integer numbers.

**Input(s):**

|      |         |               |
|------|---------|---------------|
| num1 | INTEGER | first number  |
| num2 | INTEGER | second number |

**Output(s):**

|      |         |               |
|------|---------|---------------|
| num1 | INTEGER | second number |
| num2 | INTEGER | first number  |

```

550      SUBROUTINE SwapInt (num1, num2)
551          INTEGER, INTENT(INOUT) :: num1, num2
552
553          INTEGER :: tNum
554
555          tNum = num1
556          num1 = num2
557          num2 = tNum
558      END SUBROUTINE SwapInt

```

This subroutine simply swaps two integer numbers.

### 5.6.10 *SwapReal*

This subroutine swaps the value of two double precision numbers.

**Input(s):**

|      |        |               |
|------|--------|---------------|
| num1 | REAL*8 | first number  |
| num2 | REAL*8 | second number |

**Output(s):**

|      |        |               |
|------|--------|---------------|
| num1 | REAL*8 | second number |
| num2 | REAL*8 | first number  |

```

550      SUBROUTINE SwapInt (num1, num2)
551          INTEGER, INTENT(INOUT) :: num1, num2
552
553          INTEGER :: tNum
554
555          tNum = num1
556          num1 = num2
557          num2 = tNum
558      END SUBROUTINE SwapInt

```

This subroutine simply swaps two real numbers.

# References

1. Adams JC (2009) The Fortran 2003 handbook: the complete syntax, features and procedures. Springer, London
2. Anandarajah A (2010) Computational methods in elasticity and plasticity: solids and porous media. Springer, New York
3. Blevins J (2014) Fortran wiki organization. <http://fortranwiki.org/fortran/show/Keywords>. Accessed 28 May 2016
4. Boukabara SA, van Delst P (2010) Standards, guidelines and recommendations for writing Fortran 90/95 code. [http://projects.osd.noaa.gov/SPSRB/standards\\_docs/fortran95\\_v2.1.docx](http://projects.osd.noaa.gov/SPSRB/standards_docs/fortran95_v2.1.docx). Accessed 10 Oct 2015
5. Chapman SJ (2008) Fortran 95/2003 for scientists and engineers, 3rd edn. McGraw-Hill, Boston
6. Chen WF, Zhang H (1991) Structural plasticity: theory, problems, and CAE software. Springer, New York
7. Chepurniy N (2014) Using cray pointers in a Fortran 90 program. [https://www.sharcnet.ca/help/index.php/Using\\_CRAY\\_POINTERS\\_in\\_a\\_FORTRAN\\_90\\_PROGRAM](https://www.sharcnet.ca/help/index.php/Using_CRAY_POINTERS_in_a_FORTRAN_90_PROGRAM). Accessed 12 Oct 2015
8. Chivers I, Sleightholme J (2015) Introduction to programming with Fortran, 3rd edn. Springer, Switzerland
9. Clerman NS, Spector W (2012) Modern Fortran: usage and style. Cambridge University Press, Cambridge
10. Crisfield MA (1991) Non-linear finite element analysis of solids and structures. Essentials, vol 1. Wiley, Chichester
11. Crisfield MA (1997) Non-linear finite element analysis of solids and structures. Advanced topics, vol 2. Wiley, Chichester
12. de Souza Neto EA, Perić D, Owen DRJ (2008) Computational methods for plasticity: theory and applications. Wiley, Chichester
13. Dunne F, Petrinic N (2005) Introduction to computational plasticity. Oxford University Press, Oxford
14. Farrell J (2010) Just enough programming logic and design. Course Technology Cengage Learning, Boston
15. Hinton E, Campbell JS (1974) Local and global smoothing of discontinuous finite element functions using a least squares method. *Int J Numer Methods Eng* 8(1):461–480
16. Horowitz E, Sahni S (1978) Fundamentals of computer algorithms. Computer software engineering series. Computer Science Press, Potomac
17. IEEE (2008) IEEE standard for floating-point arithmetic
18. Intel Corporation (2005) Intel Fortran language reference. Santa Clara, California

19. Intel Corporation (2012) Intel Fortran compiler XE 13.0 user and reference guides: Windows OS. Santa Clara, California
20. ISO/IEC (2010) ISO/IEC 1539-1:2010 Fortran 2008
21. Markus A, Metcalf M (2012) Modern Fortran in practice. Cambridge University Press, Cambridge
22. Metcalf M, Reid JK, Cohen M (2011) Modern Fortran explained. Numerical mathematics and scientific computation. Oxford University Press, Oxford and New York
23. Misfeldt T, Bumgardner G, Gray A (2004) The elements of C++ style. Cambridge University Press, Cambridge
24. MSC Software Corporation (2014) Marc 2014.2: element library, vol B. MSC Software Corporation, Newport Beach, California
25. MSC Software Corporation (2014) Marc 2014.2: program input, vol C. Marc 2014.2 Manual, MSC Software Corporation, Newport Beach, California
26. MSC Software Corporation (2014) Marc 2014.2: theory and user information, vol A. MSC Software Corporation, Newport Beach, California
27. MSC Software Corporation (2014) Marc 2014.2: user subroutines and special routines, vol D. MSC Software Corporation, Newport Beach, California
28. MSC Software Corporation (2014) Marc Python 2014.2: tutorial and reference manual. MSC Software Corporation, Newport Beach, California
29. Öchsner A (2014) Elasto-plasticity of frame structure elements: modeling and simulation of rods and beams. Springer, New York
30. Öchsner A (2016) Computational statics and dynamics: an introduction based on the finite element method. Springer, Singapore
31. Öchsner A, Öchsner M (2016) The finite element analysis program MSC Marc/Mentat. Springer, Singapore
32. Oualline S (1992) C elements of style: the programmers style manual for elegant C and C++ programs. M & T Books, San Matteo
33. Owen DRJ, Hinton E (1980) Finite elements in plasticity: theory and practice, 1st edn. Pineridge Press Limited, Swansea
34. Seidl H, Wilhelm R, Hack S (2012) Compiler design: analysis and transformation. Springer, New York
35. Simo JC, Hughes TJR (1998) Computational inelasticity. Springer, New York

# Index

## A

Access violation, 156  
Action statement, 10  
Actual argument, 42, 72  
Advance specifier, 102, 114  
Algorithm, 17, 20, 155  
    bubble sort, 177  
    Newton-Raphson, 241, 245, 247  
Allocation status, 52  
Anisotropic material, 192, 195  
Approximation  
    chopping, 61  
    rounding, 61  
Argument, 39, 42  
Argument association, 42, 48, 51  
Argument keyword, 7  
Array  
    constructor, 81  
    element, 75  
    element order, 79  
    extent, 75  
    implied-DO loop, 81  
    lower bound, 75  
    rank, 52, 75, 76, 78  
    shape, 75, 76  
    size, 76  
    specifier, 76  
    subscript, 76  
    upper bound, 75  
Assumed shape array, 51, 78  
Assumed size array, 80  
Asynchronous I/O, 96  
Attribute, 54  
    ALLOCATABLE, 72, 78, 90  
    DATA, 88  
    DIMENSION, 75, 77  
    INTENT, 40, 48, 83, 89

OPTIONAL, 89

PARAMETER, 46, 83, 84, 87

POINTER, 78, 90

PRIVATE, 43, 85

PUBLIC, 43, 85

SAVE, 44, 83, 86

TARGET, 51, 90

Attribute-oriented declaration, 46, 85

Automatic array, 77

## B

Base, *see* Radix

Batch file

    include\_win64.bat, 127, 161

    mentat.bat, 127

    run\_marc.bat, 126, 127, 161

    submit1.bat, 127

    submit2.bat, 127

    submit3.bat, 127

Bias, 62

Biased exponent, 59

Binary representation, 105, 116

Blank block, 86

Block data, 36, 37, 87

## C

Cancellation of host association, 48

C/C++, 162, 177

Characteristic, *see* Exponent

Column-major order, 79

Comment, 4, 13, 24

Common block, 43, 44, 86, 87, 126

    concom, 147, 270

    creeps, 147

    ctable, 137

- dimen, 147
- elemdata, 151
- iautcr, 147
- marc\_usdadm, 197
- matdat, 270
- prepro, 143
- spaceset, 147
- Compiler directive, 11, 145, 162
- Complement representation, 56
- Component keyword, 7
- Component selector, 74
- Conformable array, 76
- Continuation line, 13, 72
- CRAY pointer, 92

**D**

- Data
  - block, 43
  - dictionary, 14
  - entity, 48, 52
  - environment, 8, 13, 36, 38, 44, 47, 51, 53
  - representation, 55
- Debugging, 17, 20, 24, 33, 36, 46, 122, 130, 144, 154, 155, 158
- Debugging mode, 161, 163, 165
- Deferred-shape array, 78
- Defintion status, 52
- Delimiter, 5
- Denormalized number, 59, 62, 63
- Derived data type, 44, 51, 53, 73
- Derived type definition, 13, 51, 74
- Direct access, 97
- Divided-by-zero exception, 62
- Double precision, 70
- Dummy argument, 6, 40, 42, 51, 72, 78, 87
- Dynamic data entity, 90

**E**

- Effective item list, 109
- Encapsulation, 43, 50, 85
- Encoding length, 59, 60
- End-of-File control specifier, 102
- End-of-File record, 97
- End-of-Record control specifier, 102
- End specifier, 117
- Entity-oriented declaration, 46, 84
- Error control specifier, 103
- Exception, 60
- Explicit
  - array, 77
  - formatting, 108
  - initialization, 88

- interface, 95
  - type delcaration, 46
- Exponent, 58
- Expression, 72
- External
  - file, 97, 99
  - function, 41
  - subprogram, 36, 43, 95
  - subroutine, 41

**F**

- File position, 97
  - advancing, 97
  - non-advancing, 97
- File specifier, 105
- File unit, 96
- Fixed format, 10, 13, 132, 134, 162
- Fixed point number, 55, 58
- Flag, 27
- Floating point number, 55
- Floating point representation, 59
- Flowchart, 18, 20
  - block, 26
- Format specification, 105
- Format specifier, 102, 108
- Formatted record, 97
- Fraction, *see* Mantissa
- Free format, 2, 10, 132, 134
- Function
  - result, 6, 38, 52

**G**

- Global
  - entity, 47
  - scope, 51
- GOTO-less program, *see* Structured program

**H**

- History definition
  - CONTINUE, 131
  - CONTROL, 131
  - LOADCASE, 131
  - NEW, 130
  - PARAMETERS, 131
- Host association, 48, 51

**I**

- Identifier, *see* Name
- Implicit
  - declaration, 95

- formatting, 108
- interface, 94
- Inexact exception, 60, 61
- Initial point, 97
- Initializing, 54
- Interface, 39
  - block, 94
- Internal
  - file, 97
  - subprogram, 39, 51
- Interoperability, 177
- Intinsic function, 81
- Intrinsic
  - data type, 53, 66
    - logical, 72
    - real, 68
  - procedure, 41
- Invalid exception, 62
- Isotropic hardening, 224, 244

**K**

- Keyword, 7, 13, 42

**L**

- Label, 6
- Least significant bit (LSB), 57
- Lexical token, 5
- Literal constant, 52, 84
- Local
  - entity, 39, 47
  - scope, 51
  - variable, 16, 48, 51, 86
- Logical planning, 18

**M**

- Machine Epsilon, 61
- Mantissa, 58
- Marc subroutine
  - ELEVAR, 237
  - FORCDT, 146, 182
  - FORCEM, 184
  - HOOKLW, 192
  - IMPD, 230
  - MOTION, 199
  - ORIENT, 193
  - ORIENT2, 193
  - PLOTV, 189
  - SEPFOR, 199
  - UACTIVE, 197
  - UBGINC, 217, 219, 238
  - UBREAKGLUE, 205

- UEDINC, 213, 217, 219, 223, 226, 234, 237
- UFXORD, 217
- UINSTR, 201
- USDATA, 145, 197
- USELELM, 254, 256, 261, 266
- USHELL, 208
- USPLIT\_MESH, 220
- USPRNG, 145, 213
- UVSCPL, 241, 245, 247
- WKSPL, 136, 186

- Mixed mode expression, 69

- Model definition

- CONNECTIVITY, 130, 254
- COORDINATES, 130, 132
- DEFINE, 130, 141
- END OPTION, 130
- EXCLUDE, 143
- EXTENDED, 133
- FIXED DISP, 143
- FORCDT, 183
- GEOMETRY, 130
- INCLUDE, 130
- ISOTROPIC, 195, 242, 254
- NO PRINT, 130
- OPTIMIZE, 130
- ORTHOTROPIC, 195, 242
- POST, 130
- PRINT, 130
- PRINT ELEMENT, 130
- PRINT NODE, 130
- RESTART, 130
- SOLVER, 125
- SUMMARY, 130, 134
- TABLE, 137
- TYING, 143
- UDUMP, 230
- UFXORD, 217
- USDATA, 145, 197
- WORK HARD, 189

- Modularization, 25

- Modular programming, 34

- Module, 43, 44

- procedure, 36

- reference, 45

- subprogram, 36, 44, 95

- Most significant bit (MSB), 56, 60

- Multiplication factor, 137

**N**

- Name, 5, 15, 16

- abbreviation, 16, 273



- conflict, 45
- Name association, 51
- Named common block, 86
- Named constant, 13, 52, 72, 84
- Namelist specifier, 102
- Nonlinear spring, 215
- Normalized number, 60, 62, 63
- Normalized scientific notation, 58
- Not-a-Number (NaN), 64

**O**

- Obsolescent feature, 1, 6
- Operator, 13, 73
- Out-of-bound exception, 76
- Overflow exception, 62

**P**

- Parameter
  - ALLOCATE, 130
  - DIST LOADS, 185
  - ELEMENTS, 129, 254
  - END, 130
  - EXTENDED, 130
  - LARGE STRAIN, 224
  - SIZING, 130
  - TABLE, 137, 187
  - TITLE, 129
  - VERSION, 130
- Partial record, 114
- Pointee, 93
- Pointee array, 93
- Pointer association, 51
- Pointer status
  - associated, 91
  - defined, 91
  - disassociated, 91
  - undefined, 91
- Pre-defined common block, 147
- Procedural programming, 34
- Procedure, *see* Subprogram
- Procedure file, 122
- Program database file, 161
- Programming convention, 12
- Pseudocode, 18, 20, 26

**R**

- Radix, 55, 58
- Record number specifier, 103
- Required input/output, 146
- Reserved word, 6, 53

**S**

- Scientific notation, 58
- Scope, 13, 47, 50
- Scoping unit, 51, 95
- Script
  - PyMentat, 176
  - PyPost, 176
- Semantic error, 17
- Sequence association, 51
- Sequential access, 97
- Side effect, 39
- Sign bit, 56
- Sign-magnitude representation, 56
- Significant, *see* Mantissa
- Significant figure, 58, 61, 62, 69
- Size control specifier, 103
- Size specifier, 115
- Spaghetti code, 20
- Specification expression, 77
- Specifier, 96
- Statement, 7
- Statement keyword, 7
- Static data entity, 90
- Storage association, 51, 87
- Structure
  - block, 21
    - nesting, 24
    - repetition, 22
    - selection, 22
    - sequence, 21
    - stacking, 24
- Structured program, 20, 25, 112
- Subobject, 52, 54
- Subprogram, 35, 51
- Subscript triplet, 83
- Syntax error, 17

**T**

- Table-driven input, 135
- Terminal point, 97
- Type
  - declaration, 17, 45, 46, 53, 66, 74
  - parameter, 53, 54
    - keyword, 7
    - kind, 66, 71
    - length, 66, 71

**U**

- Unbiased exponent, 59
- Underflow exception, 62
- Unformatted record, 97
- Unit specifier, 102, 105

Unnamed entity, [47](#)  
Use association, [51](#)  
User-defined element, [255](#)  
User defined subprogram, [95](#)  
Utility subroutine  
  ELMVAR, [160](#), [234](#)  
  GMPRD, [270](#)  
  GMTRA, [270](#)  
  NODVAR, [160](#), [224](#), [226](#), [230](#)  
  QUIT, [223](#)  
  SETEL, [154](#)

  SETUP\_ELGROUPS, [151](#)  
  TABVA2, [137](#), [188](#)

**V**

Vector subscript, [83](#)

**Z**

Zero-increment solution, [130](#)  
Zero-size array, [75](#)