The Mata Book

A Book for Serious Programmers and Those Who Want to Be

William W. Gould
STATACORP LLC

A Stata Press Publication
StataCorp LLC
College Station, Texas

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in LaTeX 2$_\varepsilon$
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Print ISBN-10: 1-59718-263-X
Print ISBN-13: 978-1-59718-263-8
ePub ISBN-10: 1-59718-264-8
ePub ISBN-13: 978-1-59718-264-5
Mobi ISBN-10: 1-59718-265-6
Mobi ISBN-13: 978-1-59718-265-2

Library of Congress Control Number: 2018933411

Contents

Acknowledgment **xvii**

1 Introduction **1**

1.1 Is this book for me? . . . 1

1.2 What is Mata? . . . 2

1.3 What is covered in this book . . . 3

1.4 How to download the files for this book . . . 6

2 The mechanics of using Mata **9**

2.1 Introduction . . . 9

2.2 Mata code appearing in do-files . . . 10

2.3 Mata code appearing in ado-files . . . 12

2.4 Mata code to be exposed publicly . . . 14

3 A programmer's tour of Mata **17**

3.1 Preliminaries . . . 17

3.1.1 Results of expressions are displayed when not stored . . . 18

3.1.2 Assignment . . . 20

3.1.3 Multiple assignment . . . 20

3.2 Real, complex, and string values . . . 22

3.2.1 Real values . . . 22

3.2.2 Complex values . . . 22

3.2.3 String values (ASCII, Unicode, and binary) . . . 22

3.3 Scalars, vectors, and matrices . . . 24

3.3.1 Functions rows(), cols(), and length() . . . 25

3.3.2 Function I() . . . 25

3.3.3 Function J() . . . 26

3.3.4 Row-join and column-join operators . . . 26
3.3.5 Null vectors and null matrices . . . 29
3.4 Mata's advanced features . . . 32
3.4.1 Variable types . . . 32
3.4.2 Structures . . . 34
3.4.3 Classes . . . 36
3.4.4 Pointers . . . 38
3.5 Notes for programmers . . . 40
3.5.1 How programmers use Mata's interactive mode . . . 40
3.5.2 What happens when code has errors . . . 42
3.5.3 The _error() abort function . . . 43
4 Mata's programming statements 45
4.1 The structure of Mata programs . . . 45
4.2 The program body . . . 47
4.2.1 Expressions . . . 47
4.2.2 Conditional execution statement . . . 49
4.2.3 Looping statements . . . 50
4.2.3.1 while . . . 51
4.2.3.2 for . . . 53
4.2.3.3 do while . . . 56
4.2.3.4 continue and break . . . 57
4.2.4 goto . . . 58
4.2.5 return . . . 59
4.2.5.1 Functions returning values . . . 59
4.2.5.2 Functions returning void . . . 60
5 Mata's expressions 61
5.1 More surprises . . . 62
5.2 Numeric and string literals . . . 64
5.2.1 Numeric literals . . . 64
5.2.1.1 Base-10 notation . . . 64

5.2.1.2 Base-2 notation 66
5.2.2 Complex literals 71
5.2.3 String literals 72
5.3 Assignment operator 73
5.4 Operator precedence 74
5.5 Arithmetic operators 75
5.6 Increment and decrement operators 76
5.7 Logical operators 77
5.8 (Understand this ? skip : read) Ternary conditional operator 79
5.9 Matrix row and column join and range operators 80
5.9.1 Row and column join 80
5.9.2 Comma operator is overloaded 81
5.9.3 Row and column count vectors 82
5.10 Colon operators for vectors and matrices 82
5.11 Vector and matrix subscripting 83
5.11.1 Element subscripting 84
5.11.2 List subscripting 86
5.11.3 Permutation vectors 88
5.11.3.1 Use to sort data 88
5.11.3.2 Use in advanced mathematical programming . . . 91
5.11.4 Submatrix subscripting 92
5.12 Pointer and address operators 94
5.13 Cast-to-void operator 97
6 Mata's variable types **99**
6.1 Overview 99
6.2 The forty variable types 103
6.2.1 Default initialization 105
6.2.2 Default eltype, orgtype, and therefore, variable type 106
6.2.3 Partial types 106
6.2.4 A forty-first type for returned values from functions 107

6.3 Appropriate use of transmorphic 109
6.3.1 Use transmorphic for arguments of overloaded functions . . 109
6.3.2 Use transmorphic for output arguments 110
6.3.2.1 Use transmorphic for passthru variables 111
6.3.3 You must declare structures and classes if not passthru . . . 112
6.3.4 How to declare pointers 112
7 Mata's strict option and Mata's pragmas 115
7.1 Overview 115
7.2 Turning matastrict on and off 117
7.3 The messages that matastrict produces, and suppressing them 117
8 Mata's function arguments 121
8.1 Introduction 121
8.2 Functions can change the contents of the caller's arguments 121
8.2.1 How to document arguments that are changed 123
8.2.2 How to write functions that do not unnecessarily change arguments 125
8.3 How to write functions that allow a varying number of arguments . . 125
8.4 How to write functions that have multiple syntaxes 127
9 Programming example: n_choose_k() three ways 129
9.1 Overview 129
9.2 Developing n_choose_k() 130
9.3 n_choose_k() packaged as a do-file 134
9.3.1 How I packaged the code: n_choose_k.do 134
9.3.2 How I could have packaged the code 137
9.3.2.1 n_choose_k.mata 139
9.3.2.2 test_n_choose_k.do 141
9.3.3 Certification files 144
9.4 n_choose_k() packaged as an ado-file 145
9.4.1 Writing Stata code to call Mata functions 145
9.4.2 nchooseki.ado 147

9.4.3 test_nchooseki.do 150
9.4.4 Mata code inside of ado-files is private 153
9.5 n_choose_k() packaged as a Mata library routine 153
9.5.1 Your approved source directory 154
9.5.1.1 make_lmatabook.do 156
9.5.1.2 test.do 157
9.5.1.3 hello.mata 157
9.5.1.4 n_choose_k.mata 158
9.5.1.5 test_n_choose_k.do 158
9.5.2 Building and rebuilding libraries 158
9.5.3 Deleting libraries 159
10 Mata's structures **161**
10.1 Overview 161
10.2 You must define structures before using them 164
10.3 Structure jargon 165
10.4 Adding variables to structures 166
10.5 Structures containing other structures 166
10.6 Surprising things you can do with structures 167
10.7 Do not omit the word scalar in structure declarations 167
10.8 Structure vectors and matrices and use of the constructor function . 168
10.9 Use of transmorphic with structures 170
10.10 Structure pointers 172
11 Programming example: Linear regression **177**
11.1 Introduction 177
11.2 Self-threading code 181
11.3 Linear-regression system lr*() version 1 187
11.3.1 lr*() in action 187
11.3.2 The calculations to be programmed 192
11.3.3 lr*() version-1 code listing 194
11.3.4 Discussion of the lr*() version-1 code 197

11.3.4.1 Getting started 198
11.3.4.2 Assume subroutines 199
11.3.4.3 Learn about Mata's built-in subroutines 200
11.3.4.4 Use of built-in subroutine cross() 202
11.3.4.5 Use more subroutines 204
11.4 Linear-regression system lr*() version 2 205
11.4.1 The deviation from mean formulas 206
11.4.2 The lr*() version-2 code 208
11.4.3 lr*() version-2 code listing 211
11.4.4 Other improvements you could make 211
11.5 Closeout of lr*() version 2 213
11.5.1 Certification 213
11.5.2 Adding lr*() to the lmatabook.mlib library 218
12 Mata's classes 221
12.1 Overview 221
12.1.1 Classes contain member variables 222
12.1.2 Classes contain member functions 222
12.1.3 Member functions occult external functions 224
12.1.4 Members—variables and functions—can be private 225
12.1.5 Classes can inherit from other classes 227
12.1.5.1 Privacy versus protection 229
12.1.5.2 Subclass functions occult superclass functions . . . 229
12.1.5.3 Multiple inheritance 230
12.1.5.4 And more 231
12.2 Class creation and deletion 231
12.3 The this prefix 233
12.4 Should all member variables be private? 234
12.5 Classes with no member variables 236
12.6 Inheritance 238
12.6.1 Virtual functions 240

12.6.2 Final functions 243
12.6.3 Polymorphisms 245
12.6.4 When to use inheritance 246
12.7 Pointers to class instances 247
13 Programming example: Linear regression 2 **249**
13.1 Introduction 249
13.2 LinReg in use 250
13.3 LinReg version-1 code 252
13.4 Adding OPG and robust variance estimates to LinReg 253
13.4.1 Aside on numerical accuracy: Order of addition 257
13.4.2 Aside on numerical accuracy: Symmetric matrices 258
13.4.3 Finishing the code 259
13.5 LinReg version-2 code 260
13.6 Certifying LinReg version 2 260
13.7 Adding LinReg version 2 to the lmatabook.mlib library 261
14 Better variable types **263**
14.1 Overview 263
14.2 Stata's macros 264
14.3 Using macros to create new types 264
14.4 Macroed types you might use 266
14.4.1 The boolean type 268
14.4.2 The Code type 269
14.4.3 Filehandle 271
14.4.4 Idiosyncratic types, such as Filenames 272
14.4.5 Macroed types for structures 272
14.4.6 Macroed types for classes 273
14.4.7 Macroed types to avoid name conflicts 273
15 Programming constants **277**
15.1 Problem and solution 277
15.2 How to define constants 278

15.3 How to use constants . . . 279
15.4 Where to place constant definitions . . . 279
16 Mata's associative arrays 281
16.1 Introduction . . . 281
16.2 Using class AssociativeArray . . . 282
16.3 Finding out more about AssociativeArray . . . 284
17 Programming example: Sparse matrices 285
17.1 Introduction . . . 285
17.2 The idea . . . 286
17.3 Design . . . 287
17.3.1 Producing a design from an idea . . . 287
17.3.2 The design goes bad . . . 293
17.3.3 Fixing the design . . . 295
17.3.3.1 Sketches of R_*x*() and S_*x*() subroutines . . . 299
17.3.3.2 Sketches of class's multiplication functions . . . 303
17.3.4 Design summary . . . 308
17.3.5 Design shortcomings . . . 311
17.4 Code . . . 312
17.5 Certification script . . . 321
18 Programming example: Sparse matrices, continued 323
18.1 Introduction . . . 324
18.2 Making overall timings . . . 325
18.2.1 Timing T1, Mata R=RR . . . 327
18.2.2 Timing T2, SpMat R=RR . . . 327
18.2.3 Timing T3, SpMat R=SR . . . 328
18.2.4 Timing T4, SpMat R=RS . . . 328
18.2.5 Timing T5, SpMat R=SS . . . 329
18.2.6 Call a function once before timing . . . 329
18.2.7 Summary . . . 330
18.3 Making detailed timings . . . 330

18.3.1 Mata's timer() function . . . 331
18.3.2 Make a copy of the code to be timed . . . 331
18.3.3 Make a do-file to run the example to be timed . . . 331
18.3.4 Add calls to timer_on() and timer_off() to the code . . . 332
18.3.5 Analyze timing results . . . 336
18.4 Developing better algorithms . . . 338
18.4.1 Developing a new idea . . . 338
18.4.2 Aside . . . 340
18.4.2.1 Features of associative arrays . . . 341
18.4.2.2 Advanced use of pointers . . . 344
18.5 Converting the new idea into code sketches . . . 348
18.5.0.3 Converting the idea into a sketch of R_SxS() . . . 349
18.5.0.4 Sketching subroutine cols_of_row() . . . 352
18.5.1 Converting sketches into completed code . . . 354
18.5.1.1 Double-bang comments and messages . . . 357
18.5.1.2 // NotReached comments . . . 358
18.5.1.3 Back to converting sketches . . . 358
18.5.2 Measuring performance . . . 360
18.6 Cleaning up . . . 360
18.6.1 Finishing R_SxS() and cols_of_row() . . . 361
18.6.2 Running certification . . . 365
18.7 Continuing development . . . 366
19 The Mata Reference Manual 369
A Writing Mata code to add new commands to Stata 373
A.1 Overview . . . 373
A.2 Ways to structure code . . . 375
A.3 Accessing Stata's data from Mata . . . 379
A.4 Handling errors . . . 384
A.5 Making the calculation and displaying results . . . 387
A.6 Returning results . . . 389

A.7 The Stata interface functions . . . 393
A.7.1 Accessing Stata's data . . . 393
A.7.2 Modifying Stata's data . . . 394
A.7.3 Accessing and modifying Stata's metadata . . . 394
A.7.4 Changing Stata's dataset . . . 396
A.7.5 Accessing and modifying Stata macros, scalars, matrices . . . 396
A.7.6 Executing Stata commands from Mata . . . 397
A.7.7 Other Stata interface functions . . . 399

B Mata's storage type for complex numbers 401

B.1 Complex values . . . 401
B.2 Complex values and literals . . . 403
B.3 Complex scalars, vectors, and matrices . . . 404
B.4 Real, complex, and numeric eltypes . . . 406
B.5 Functions Re(), Im(), and C() . . . 408
B.6 Function eltype() . . . 408

C How Mata differs from C and C++ 411

C.1 Introduction . . . 411
C.2 Treatment of semicolons . . . 411
C.3 Nested comments . . . 412
C.4 Argument passing . . . 412
C.5 Strings are not arrays of characters . . . 412
C.6 Pointers . . . 413
C.6.1 Pointers to existing objects . . . 413
C.6.2 Pointers to new objects, allocation of memory . . . 413
C.6.3 The size and even type of the object may change . . . 415
C.6.4 Pointers to new objects, freeing of memory . . . 415
C.6.5 Pointers to subscripted values . . . 415
C.6.6 Pointer arithmetic is not allowed . . . 416
C.7 Lack of switch/case statements . . . 416
C.8 Mata code aborts with error when C would crash . . . 416

D Three-dimensional arrays (advanced use of pointers) 417

D.1 Introduction 417

D.2 Creating three-dimensional arrays 417

References 419

Author index 421

Subject index 423

(Pages omitted)

1 Introduction

1.1	Is this book for me?	1
1.2	What is Mata?	2
1.3	What is covered in this book	3
1.4	How to download the files for this book	6

1.1 Is this book for me?

This book is for you if you have tried to learn Mata by reading the *Mata Reference Manual* and failed. You are not alone. Though the manual describes the parts of Mata, it never gets around to telling you what Mata is, what is special about Mata, what you might do with Mata, or even how Mata's parts fit together. This book does that.

This is an applied book. It will teach you the modern way to write programs, which is to say, it will teach you about structures, classes, and pointers. And the book will show you some programming techniques that may be new to you. In short, in this book, we are going to use Mata to write programs that are good enough that StataCorp could distribute them.

This book is for "serious programmers and those who want to be". Fifteen years ago, the subtitle would have referenced professional rather than serious programmers, and yet I would have written the same book. These days, the distinction is evaporating. I meet researchers who do not program for a living but are most certainly serious. And I meet the other kind, too.

A serious programmer is someone who has a serious interest in sharpening their programming skills and broadening their knowledge of programming tools. There is an easy test to determine whether you are serious. If I tell you that I know of a new technique for programming interrelated equations and your response is "Tell me about it," then you are serious.

Being serious is a matter of attitude, not current skill level or knowledge.

Still, I made assumptions in writing this book. I assumed that you have some experience with at least one programming language, be it Stata's ado, Python, Java, C++, Fortran, or any other language you care to mention. I also assumed that you already know that programs contain conditional statements and loops. If you need a first introduction to

programming, you could look at the introductory section of the Mata manual or at the Mata chapters in Baum's friendly text *An Introduction to Stata Programming* (2016).

The examples in this book are statistical and mathematical. Formulas are provided, but the formulas are of secondary importance. They just provide the examples of something for us to program.

In this book, I will show you a language aimed at programming statistical and data management applications that has all the usual features and some unique ones, too. And I will show you programming techniques that might be new to you.

As I said, being serious is a matter of attitude. New techniques and languages are continually being developed, and you need to learn them, just as I still learn them. I have been programming for 45 years as a professional. I have a lot of experience and knowledge, but I have not stopped learning new techniques. I may be a professional programmer, but more importantly, I am a serious one.

1.2 What is Mata?

Many Stata users would describe Mata as a matrix language. StataCorp itself markets Mata that way. Mata would be more accurately described, however, as an across-platform portable-code compiled programming language that happens to have matrix capabilities. Just as important as its matrix capabilities are Mata's structures, classes, and pointers.

We at StataCorp designed and wrote Mata to be the development language that we would use. Nowadays, we write most new features of Stata in Mata. Before Mata existed, we used C. Compared with C, Mata code is easier to write, less error prone, easier to debug, and easier to maintain.

It is important that Mata is compiled. Being compiled means that programs run fast. Stata's other programming language, ado, is interpreted. Interpreted languages are slow in comparison with compiled languages. Mata code runs 10–40 times faster than ado.

Mata looks a lot like C and C++. In *The C Programming Language*, Kernighan and Ritchie (1978) introduced what has become perhaps the most famous first program:

```
main()
{
        printf("hello, world\n") ;
}
```

To convert the program to Mata, we need to add `void` in front of `main()`:

```
: void main()
> {
>         printf("hello, world\n") ;
> }
: main()
hello, world
```

Most Mata users would not bother typing the semicolon at the end of `printf("hello, world\n")`. Semicolons are optional in Mata. There are other differences between the languages, too. Those differences are covered in appendix C.

1.3 What is covered in this book

The programs we will write in this book are

Filename	Contents
`hello.mata`	First program, function `hello()`
`n_choose_k.mata`	Serious but short function, packaged as library function
`lr1.mata`	Linear regression, ver. 1 (structures)
`lr2.mata`	Linear regression, ver. 2 (structures)
`earthdistance.mata`	An aside concerning classes
`linreg1.mata`	Linear regression take 2, ver. 1 (classes)
`linreg2.mata`	Linear regression take 2, ver. 2 (classes)
`spmat1.mata`	Sparse matrices, ver. 1
`spmat2.mata`	Sparse matrices, ver. 2
`spmat3.mata`	Sparse matrices, ver. 3

The first serious program we will write is `n_choose_k()`. It will have just 47 lines including comments and white space.

We will then work our way to a nearly complete implementation of linear regression, starting with `lr1.mata` and ending with `linreg2.mata`. There will be only 388 lines in the final code in `linreg2.mata`! We will use structures for the first two implementations and use classes after that.

The `earthdistance.mata` program merely illustrates a point about class programming.

Finally, we will undertake a large project, namely, the implementation of sparse matrices. Sparse matrices are matrices in which most elements are 0. The project will concern storing the matrices efficiently—there is no reason to store all those 0s—and writing code to add and multiply them just as if they were regular matrices. File `spmat3.mata` will contain 937 lines.

We will do all that, but we will not start until chapter 9. There is a lot to tell you first.

Chapter 2 covers the mechanics of using Mata. You may know that Mata can be used interactively, but that is not how we will be using it except when we want to experiment before committing an idea to code.

Chapter 3 takes you on a tour of Mata. It will show you ordinary features, such as assignment; surprising features, such as 0×0 matrices and 0×1 and 1×0 vectors; and advanced features, such as structures, classes, and pointers. Pointers, by the way, are not nearly as difficult to understand as you might fear. Later, we will use pointers when we write `lr1.mata`, our first implementation of linear regression, and we will use them in an advanced way when we write `spmat3.mata` to implement sparse matrices.

Chapter 4 explains Mata's programming statements, all nine of them. There may be only nine, but they fit together in remarkable ways.

Chapter 5 provides details about Mata's expressions, such as `y = sqrt(2)`. Expressions are one of the nine programming statements, but that understates their importance because they comprise the bulk of programs. Just calling a subroutine is an expression. Chapter 5 also discusses programming for numerical accuracy. Do not skip section 5.2.1.2 even though its title is *Base-2 notation*.

Chapter 6 describes Mata's 40 variable types. One of them is `transmorphic`, and the chapter enumerates its proper and improper uses.

Chapter 7 is about Mata's `strict` option. `strict` tells Mata to flag questionable constructs in programs. Bugs hide inside questionable constructs.

Chapter 8 is about function arguments. Mata passes arguments by reference, but you may not yet know what that means. The chapter also shows how to write functions that allow a varying number of arguments.

In chapter 9, we finally turn to programming. The chapter is entitled *n_choose_k() three ways*. We will write the new function `n_choose_k()` and use it in three ways. We will use the function in an analysis do-file, as the computational engine inside an ado-file, and as a function to be added to a Mata library so that it can be used anywhere and anyplace.

We will start programming in chapter 9, and we will not stop. A few chapters after 9 will explain Mata features that we will need for the programs we will write. Chapters 10 and 11 deeply explain structures. Chapters 12 and 13 do the same for classes. Chapter 14 shows how to create new variable types so you can declare a variable to be `boolean` instead of `real` or an `SpMat` instead of a `class SpMat scalar`. Chapter 15 shows a better way to deal with constants that appear in code. Chapter 16 explains Mata's associative arrays.

The chapters of this book are about Mata, not Stata. All but one example is about writing Mata programs to be called from other Mata programs. And yet, the purpose of Mata is to add new features to Stata. In appendix A, we will finally discuss programming for Stata. Because you will have read the chapters, we will be able to discuss the subject as one serious and knowledgeable programmer with another. There will be three issues for us to discuss.

The first issue is how code should be structured. Stata's ado language is how new commands are added to Stata, and Mata does not change that. The question is whether you should write one line of ado-code calling Mata so that the entire program is written in Mata, or you should parse in Stata and then call Mata, or you should leave the ado-code in charge and use Mata to provide the occasional subroutine for the ado-code to call.

The second issue is how to access Stata objects such as variables, observations, macros, and the like. Mata provides functions to do this.

The third issue is how to handle errors caused by mistakes by the users of our code. By default, Mata aborts with error and issues a traceback log. That is acceptable behavior when we write subroutines for use by other serious programmers, but it is not acceptable when writing code for direct use by Stata users. Mata has functions that will issue informative error messages and stop execution with a nonzero return code so that we can write code that handles errors as gracefully as Stata users expect.

The book covers more, too. A thorough treatment of programming requires discussion about workflow. Workflow is jargon for how to organize your work from the time you write the first line of code to the time the program is ready to ship or be put in use. Workflow is also about how you will later fix the program's first reported bug, and its second, and the substantive expansion of capabilities that you will make two years from now.

The workflow discussion begins in chapter 2, becomes more detailed in chapter 9, and continues in every programming example thereafter. Earlier, I mentioned the programs we will be writing: `hello.mata`, `n_choose_k.mata`, `lr1.mata`, and so on. When we write `lr1.mata`, we will also write file `test_lr1.do`, a Stata do-file to certify that the code in `lr1.mata` produces correct results. We will store the certified code and its test file in our Approved Source Directory. We will develop an automated procedure for creating and updating Mata libraries that recompiles all the code in all the `*.mata` files, runs all the `test_*.do` files, and rebuilds libraries from scratch.

(Pages omitted)

2 The mechanics of using Mata

2.1 Introduction 9
2.2 Mata code appearing in do-files 10
2.3 Mata code appearing in ado-files 12
2.4 Mata code to be exposed publicly 14

2.1 Introduction

I showed the Mata function for `hello()` in the last chapter. Here it is again, although this time I have changed the function's name from `main()` to `hello()` and I execute it:

```
. mata:
------------------------------------- mata (type end to exit) -----
: void hello()
> {
>         printf("hello, world\n") ;
> }

: hello()
hello, world

: end
--------------------------------------------------------------------------------

. _
```

Just the act of entering the program caused Mata to compile it. Mata compiled `hello()`, discarded the original source code, and left the compiled code in memory. That is why I can execute the function by typing `hello()`.

This interactive approach can be useful in teaching, but it is useless for serious applications. There are three ways Mata code is used more seriously.

Mata code can be placed in do-files. The functions you define there can be used interactively and by other do-files.

Mata code can be placed in ado-files. The functions you define there can be used inside the ado-file.

Mata code can be compiled and placed in libraries. The functions you place in them may be used anytime, anywhere. They can be used interactively, in do-files, in ado-files, and in other functions that appear in the same or different libraries.

2.2 Mata code appearing in do-files

I do not recommend putting Mata code straight into analysis do-files, although I have done that when the code was simple enough. Complicated code will need debugging, and debugging is easier when the code can be worked on in isolation. That argues for putting the code in its own do-file. Doing that also makes it easier to use the Mata code in other analyses.

I recommend that you place the code in its own do-file with the file extension `.mata`, such as

hello.mata
```
version 15

mata:
void hello()
{
        printf("hello, world\n")
}
end
```
hello.mata

Additional functions can appear in the same file:

hello.mata
```
version 15

mata:
void hello()
{
        printf("hello, world\n")
}

void goodbye()
{
        printf("good-bye, world\n")
}
end
```
hello.mata

Functions in the same file should be related. `hello()` and `goodbye()` are related; in the unlikely event you want to use one of them, you will probably need the other. Related use is a fine reason for functions to appear in the same file. Usually, however, the functions are even more related in that they call one another.

To use the functions in your analysis do-file, code `do` *filename*`.mata` in the do-file before using them:

```
------------------------------------------------------ analysis.do -------
version 15
clear all
.
.
.
do hello.mata
mata: hello()
mata: goodbye()
.
.
.
------------------------------------------------------ analysis.do -------
```

The line `do hello.mata` appears in boldface only for emphasis. When the `analysis.do` do-file executes the line, the `hello.mata` do-file will be executed, which will define the functions `hello()` and `goodbye()`. You could execute `analysis.do` by typing

```
. do analysis
  (output omitted)
```

File `analysis.do` begins with the line `version 15`. Version control is a hallmark of Stata. Every Stata do-file and ado-file since Stata 1 (in 1985) still works even though Stata's programming language looks nothing like it did originally. I included `version 15` in this file so that it will continue to work in the future.

In this book, we will use `.mata` for files containing Mata code. Those files should start with a version number, too. Look back and you will see that `version 15` appears at the top of file `hello.mata`. Version control serves the same purpose in `.mata` files that it does in do-files and ado-files. If some Mata language feature should change in the future, that feature will be backdated to have its old meaning. The version number does not preclude the use of features added later; it merely handles backdating for changes in syntax.

If there seems to be a profusion of `version 15` statements in these two files, imagine that it is now two-and-a-half years later and you are using Stata 16. I also need you to imagine that `analysis.do` is a real analysis do-file and that `hello()` and `goodbye()` do something useful. Typing `do analysis` will obviously reproduce the original results, but that is not what you want to do. You want to add a second analysis using a new Stata 16 feature. You create file `analysis2.do` and it starts, naturally enough, with `version 16`. You also want to use `hello()` and `goodbye()` in the new file, so you include `do hello.mata` in new file `analysis2.do`. The `version 15` in `mata.do` will assure that the code in `hello()` and `goodbye()` is given the Stata 15 interpretation when the functions are compiled. Thus, even in this new Stata 16 do-file, old functions `hello()` and `goodbye()` will work as originally intended.

(Pages omitted)

4 Mata's programming statements

4.1 The structure of Mata programs 45
4.2 The program body 47
4.2.1 Expressions 47
4.2.2 Conditional execution statement 49
4.2.3 Looping statements 50
4.2.3.1 while 51
4.2.3.2 for 53
4.2.3.3 do while 56
4.2.3.4 continue and break 57
4.2.4 goto 58
4.2.5 return 59
4.2.5.1 Functions returning values 59
4.2.5.2 Functions returning void 60

4.1 The structure of Mata programs

Individual programs are formally called functions in Mata, but that will not stop us from calling them programs, routines, or subroutines. A program is a chunk of code. Here are some examples.

Function `speed_of_light()` takes no arguments and returns a value. It would be useful if you were an astrophysicist.

```
real scalar speed_of_light()
{
        return(299792458 /* m/sec */)
}
```

Function `show()` takes arguments but returns nothing. Functions returning nothing are common when displaying results or writing results to a file.

```
void show(real scalar a)
{
        printf("a = %f\n", a)
}
```

Function `n_choose_k()` and its subroutine `nfactorial_over_kfactorial()` really are functions in the mathematical sense because they accept arguments and return results.

```
real scalar n_choose_k(real scalar n, real scalar k)
{
        return( n-k > k ?
                nfactorial_over_kfactorial(n, n-k) /
                nfactorial_over_kfactorial(k, 1)
                :
                nfactorial_over_kfactorial(n, k) /
                nfactorial_over_kfactorial(n-k, 1)
              )
}

real scalar nfactorial_over_kfactorial(real scalar n,
                                       real scalar k)
{
        real scalar     result, i

        if (n<0 | n>1.0x+35 | n!=trunc(n)) return(.)
        if (k<0 | k>1.0x+35 | k!=trunc(k)) return(.)

        result = 1
        for (i=n; i>k; --i) result = result*i

        return(result)
}
```

I want you to focus on the physical structure of the programs. That structure is

returnedtype *name*(*arguments*)
{
        declarations

        program body
}

All Mata functions have this structure.

Most functions require *arguments* and return something. *returnedtype* specifies what is returned, such as a `real scalar` or `complex matrix`.

Functions that return nothing are said to return `void`.

You can omit the *declarations* of the variables used in the body of the program, but we will not omit them in this book. Omitting the declarations increases the chances of mistakes, and programs without declarations sometimes run slower. They run slower when the compiler—not knowing the type—needs to produce more general code that can handle all the possibilities.

4.2 The program body

There are nine statements that can be used in the program body. They are as follows:

Conditional execution statements:

```
if (expr) ... else ...
```

Looping statements:

```
for (expr; expr; expr) ...
while (expr) ...
do ... while (expr)
continue            (continue with next iteration of loop)
break               (break out of loop)
```

Go-to statements (useful when translating Fortran programs):

```
goto stmt
```

Exit and exit-and-return-value statements:

```
return  and  return(expr)
```

Assignment, subroutine calls, and the like:

```
expr
```

expr is an abbreviation for expressions.

4.2.1 Expressions

We will discuss expressions deeply in the next chapter, and anyway, you already know what expressions are. Examples of expressions include

```
i = i + 1
y = myfcn(x)
mysubroutine(a, b)
```

This last example may not look like an expression to you, but it is. It is an expression that returns `void`.

There is a lot I could tell you about expressions, but as I said, you mostly know what they are. I do need to tell you about three surprising features of expressions, however.

The first surprising feature is that mathematical expressions such as

```
(-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

and logical expressions such as

```
a>1 & b<2
```

are, despite appearances, both numerical expressions. They are numerical because they both return numerical results. Logical expressions return 1 or 0, where 1 means true and 0 means false. Mathematical and logical expressions may differ in the operators used, but they do not differ in the type of results they produce. Because they do not differ, mathematical and logical operators can be combined in surprising and useful ways.

For instance, say you have three numerical variables, `a`, `b`, and `c`. How many are negative? Answer: `(a<0) + (b<0) + (c<0)` are negative.

Arithmetic expressions can substitute for logical expressions, too. A condition is deemed to be true if the expression evaluates to any value except 0 because 0 means false. This means you can code

```
if ( (-b + sqrt(b^2 - 4*a*c)) / (2*a) ) ...
```

and what follows the `if` will be executed when `(-b + sqrt(b^2 - 4*a*c)) / (2*a)` is not 0.

The equivalency of numeric and logical expressions is Mata's first surprising feature. The second is that `=` means assignment and `==` means equality. Do not code

```
if (x=2) ...
```

when you mean

```
if (x==2) ...
```

The first is not an error; it is a bug. Mata will not complain when you code `if (x=2)`, but the code will not do what you expect. The code will treat `x=2` as assignment, meaning `x` will be changed to be 2. If that is not bad enough, assignment leaves behind the value, so the expression will be treated as true.

Coding `x==2` is how you ask whether `x` is equal to 2.

Coding `x!=2` is how you ask whether `x` is not equal to 2.

Finally, I need to tell you about Mata's `++` and `--` operators. Coding `i++` increments `i` by 1. You can think of it as a shorthand for `i = i + 1`. By the same token, coding `i--` decrements `i` by 1.

Later in this chapter, I will show you examples of `i++`, such as

```
for (i=0; i<=n; i++) ...
```

I could just as well present the example as

```
for (i=0; i<=n; i=i+1) ...
```

Most programmers type `i++` instead of `i = i + 1`.

You can code the `++` operator after the variable name or before it: `i++` or `++i`. When coded as a standalone statement, which you code makes no difference. Coded in the midst of an expression, there is a distinction. Look at the following two statements:

```
z = v[i++] + x

z = v[++i] + x
```

`v[i++]` means obtain `v[i]` and then increment `i`.

`v[++i]` means increment `i` and then obtain `v[i]`.

For instance, if `i` were 2 before the statements were executed, then

`v[i++]` accesses `v[2]`, whereas

`v[++i]` accesses `v[3]`,

and either way, `i` is incremented to be 3.

`i--` and `--i` work the same way.

4.2.2 Conditional execution statement

The syntax of `if (`*expr*`) ... else ...` is

`if (`*expr*`)` *stmt1*

and

`if (`*expr*`)` *stmt1*
`else` *stmt2*

stmt1 is executed if *expr* evaluates to true (nonzero).

When `else` is coded, *stmt2* is executed if *expr* evaluates to false (zero).

You can code

```
if (x==2) y = myfcn(z)
```

and you can code

```
if (x==2) y =  myfcn(z)
else      y = altfcn(z)
```