

*Center for Advanced Studies in  
Measurement and Assessment*

*CASMA Monograph*

*Number 1*

## **Equating Recipes**

*Robert L. Brennan*

*Tianyou Wang*

*Seonghoon Kim*

*Jaehoon Seol*

With Contributions by  
Michael J. Kolen

Version 1.0  
September 2009

Center for Advanced Studies in  
Measurement and Assessment (CASMA)  
College of Education  
University of Iowa  
Iowa City, IA 52242  
Tel: 319-335-5439  
Fax: 319-384-0505  
Web: [www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma)

All rights reserved

Dedicated to  
*Bradley A. Hanson*

Intentionally blank page

## Preface

The first edition of the Kolen and Brennan book on *Test equating: Methods and practices* was published in 1995. It was coordinated with executable versions of six free computer programs that were written in either Fortran or C. The second edition of Kolen and Brennan was published in 2004. It was coordinated with nine computer programs, many of which are the same as those discussed in the first edition. These free programs are available in executable form on [www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma), which is the web-site for the Center for Advanced Studies in Measurement and Assessment (CASMA).<sup>1</sup>

There were two primary purposes for making these programs publicly available: to facilitate research and to serve as a resource to students studying equating. Although I believe these programs serve these functions quite well (especially the latter), they were never intended as programs for operational equating, and over time various problems, or at least challenges, were encountered. First, the earlier programs were all written for a Macintosh platform. As Macs became less used by psychometricians, the programs were converted to the PC/Windows platform, usually as console applications. Most programs are still available for both platforms, but it has become increasingly difficult to maintain versions of the programs for both platforms. Second, the programs generally use different conventions for input, and there is considerable redundancy in functions across programs. For example, nearly all programs involve computation of moments, but the functions for doing so vary across programs. Third, these programs are distributed in executable form, only, because they are written in different languages, by different persons, at different times, and usually not structured, commented, and documented in a manner that would make the code useful for persons other than the authors.

For these reasons (especially the last), these programs are not very viable candidates for open-source distribution, yet it has become blatantly clear that open source functions for equating are needed. As discussed more fully in Chapter 1, in effect, operational equating conducted by testing companies, as well as published research involving equating, is largely a black box because the details of the algorithms employed are often only crudely documented, and the code that operationalizes the algorithms is almost never available. In fact, testing companies typically treat equating code as highly proprietary, and only rarely are the algorithms well documented for the public. Proprietary claims are usually defended with business arguments, and lack of documentation is usually rationalized with explanations

---

<sup>1</sup>In alphabetical order, the authors of these computer programs are Robert L. Brennan, Yuehmei Chien, Zhongmin Cui, Bradley A. Hanson, Seonghoon Kim, Michael J. Kolen, and Lingjia Zeng. The major authors are Kolen and Hanson.

about lack of time. Such arguments are themselves arguable, but it seems unarguable that without access to code, or at least well documented algorithms, it is nearly impossible for others to assert objectively that equating has been done adequately and, hence, that reported scores for examinees are accurate.<sup>2</sup> With respect to equating research, code is sometimes made available by authors on an informal basis, but all too frequently those who use and study equating research must make a number of assumptions about how the equating was actually done.

The principal reason that *Equating Recipes* has been written is to address these concerns through making open-source code available for the vast majority of equating procedures. Hopefully, that the availability of *Equating Recipes* will remove, or at least mitigate, one large impediment to more transparency in equating.

It took over four years to develop and test the code (over 25,000 lines) and write the documentation for *Equating Recipes*. Most of the code is new, although there are some code segments that have their genesis in the programs cited in Kolen and Brennan (2004).

The authors of *Equating Recipes* are psychometricians; most are not professional programmers. For this reason, as well as many others, undoubtedly the code can be improved, especially when it is used in specific contexts with unique requirements. However, one of the distinct advantages of open source code is just that—it is open to anyone to examine, change, and/or improve, as desired. That fact alone is more than adequate justification for making the *Equating Recipes* code open source.

Most of the funding for the project that led to *Equating Recipes* was provided by the College Board via a contract for which I was the principal investigator. I am very grateful to the College Board, and particularly Wayne Camara, for supporting this project. Tianyou Wang was actively involved in almost all aspects of this project from the very beginning. He is the primary author of the code and chapters on kernel equating and continuized log-linear equating. Seonghoon Kim is the primary author of the code and chapters on IRT scale transformation and equating. Jaehoon Seol is the primary author of the cubic spline code and chapter, and he assisted with other aspects of this project, as well. Won-Chan Lee made various contributions throughout the history of this project.

I wish to acknowledge the special contribution of two persons. Technically, Michael J. Kolen is not an author of *Equating Recipes*, but it would be difficult to overestimate the extent to which his contributions to equating influenced *Equating Recipes*. For over 25 years, Michael and I have worked together on operational equating, equating research, writing about equat-

---

<sup>2</sup>If equating code is not made available by testing companies, then, in my opinion, there should be periodic, independent replication of operational equating results by an external party.

ing, and teaching equating. Our efforts have been so collaborative that it is sometimes difficult to recall which of us was responsible for what development, idea, or activity. The authors of *Equating Recipes* are very much indebted to him. Second, of the people I have worked with in my career, no one surpassed Bradley A. Hanson in terms of joint, superb skills in psychometrics (particularly equating) and computer programming. Also, Brad was a strong advocate of open source code. Without question, he inspired me to develop *Equating Recipes*, even though he died prematurely before the project was even conceived. *Equating Recipes* is dedicated to him, and I hope he would see it as part of his legacy to the field of measurement.

Iowa City, IA  
March 2009

Robert L. Brennan

Intentionally blank page



## Versions, Revisions, and Notes

- The Beta Version of *Equating Recipes* that was released in March 2009 required the following *Numerical Recipes* (Press, Teukolsky, Vetterling, & Flannery, 1992) functions: `ran2()`, `sort()`, `ludcmp()`, `lubksb()`, `qrdcmp()`, `genqrdcmp()`, `gaussj()`, `dfpmin()`, `lnsrch()`, and `rtsafe()`. Since these functions are not open-source and not in the public domain, users of *Equating Recipes* had to obtain these functions on their own. Version 1.0 of *Equating Recipes* includes open-source functions that have the same functionality as these *Numerical Recipes* functions. These open-source functions have names that begin with `er_`. (Note that `er_random()` is the replacement for the *Numerical Recipes* function `ran2()`, and `er_qrdcmp()` replaces both the *Numerical Recipes* functions `qrdcmp()` and `genqrdcmp()`.) These open-source functions are included in `ERutilities.c` with the prototypes in `ERutilities.h`. The files `NR.c` and `NR.h` are no longer required.
- The replacement of *Numerical Recipes* functions with open-source functions necessitated some modifications to the following files: `ERutilities.c`, `ERutilities.h`, `Bootstrap.c`, `LogLinear.c`, `matrix.c`, `matrix.h`, `kernelEquate.c`, `CLL_Equate.c`, `IRTst.c`, and `IRTeq.c`. These changes can be found in the code between the comments “Before version 1.0 update” and “End of version 1.0 update.”

Intentionally blank page

# Contents

<b>Preface</b>	<b>v</b>
<b>Versions, Revisions, and Notes</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Purposes . . . . .	2
1.2 Framework . . . . .	4
1.2.1 Categorization Schema . . . . .	4
1.2.2 Wrapper Functions . . . . .	6
1.2.3 Structures . . . . .	9
1.3 Use and Limitations . . . . .	10
1.4 Additional Comments and Conventions . . . . .	11
1.4.1 Typesetting Conventions . . . . .	12
1.4.2 Header Files . . . . .	12
1.4.3 Raw Scores . . . . .	12
1.4.4 Input Files . . . . .	13
1.4.5 Differences vis-a-vis other CASMA Computer Programs . . . . .	13
1.4.6 Differences vis-a-vis Kolen and Brennan (2004) . . .	13
1.4.7 Arrays . . . . .	13
1.4.8 Release of Memory . . . . .	14
1.4.9 Verbal Shortcuts . . . . .	14
1.5 Appendix: Open Source License . . . . .	14

<b>2</b>	<b>Utilities</b>	<b>17</b>
2.1	Descriptions of Utility Functions . . . . .	17
2.1.1	Raw Scores and Locations . . . . .	17
2.1.2	Reading and Storing Data . . . . .	18
2.1.3	Basic Distributional Characteristics . . . . .	20
2.1.4	Equipercentile Equating . . . . .	21
2.1.5	Scale Scores . . . . .	22
2.1.6	Alphanumeric Conversions . . . . .	24
2.1.7	Print Functions . . . . .	25
2.1.8	Miscellaneous . . . . .	27
2.2	Example . . . . .	27
2.2.1	USTATS . . . . .	29
2.2.2	BSTATS . . . . .	31
<b>3</b>	<b>Linear Equating with Random-Groups and Single-Group Designs</b>	<b>37</b>
3.1	Functions . . . . .	37
3.1.1	Wrapper_RN() . . . . .	38
3.1.2	Print_RN() . . . . .	38
3.1.3	Single-group Design Functions . . . . .	39
3.2	Random Groups Example . . . . .	39
3.2.1	Equated Raw Scores . . . . .	41
3.2.2	Equated Scale Scores . . . . .	41
<b>4</b>	<b>Linear Equating with the Common-item Nonequivalent Groups Design</b>	<b>47</b>
4.1	Observed-Score Equating . . . . .	47
4.1.1	Tucker Equating . . . . .	48
4.1.2	Levine Observed-Score Equating . . . . .	48
4.1.3	Chained Linear Equating . . . . .	48
4.2	Levine True-Score Equating . . . . .	49
4.3	Functions . . . . .	49
4.3.1	Wrapper_CN() . . . . .	50
4.3.2	CI_LinEq() . . . . .	51
4.3.3	CI_LinObsEq() . . . . .	52
4.3.4	Print_CN() . . . . .	53
4.4	Example . . . . .	53
<b>5</b>	<b>Equipercentile Equating with Random-Groups and Single-Group Designs</b>	<b>57</b>
5.1	Functions . . . . .	58
5.2	Random Groups Example . . . . .	58
5.3	Zero Frequencies . . . . .	59

<b>6</b>	<b>Equipercentile Equating with the Common-item Nonequivalent Groups Design</b>	<b>65</b>
6.1	Frequency Estimation . . . . .	65
6.2	Modified Frequency Estimation . . . . .	66
6.3	Chained Equipercentile Equating . . . . .	67
6.4	Zero Frequencies . . . . .	68
6.5	Functions . . . . .	68
6.6	Example . . . . .	69
<b>7</b>	<b>Analytic Standard Errors</b>	<b>75</b>
7.1	Delta Method . . . . .	75
7.2	Functions . . . . .	76
7.3	Example . . . . .	76
<b>8</b>	<b>Bootstrap Standard Errors</b>	<b>81</b>
8.1	Functions . . . . .	82
8.1.1	Wrapper_Bootstrap() . . . . .	82
8.1.2	Print_Boot_se_era() . . . . .	85
8.1.3	Print_Boot_se_ess() . . . . .	85
8.2	Example . . . . .	86
8.2.1	Raw Scores . . . . .	86
8.2.2	Scale Scores . . . . .	89
<b>9</b>	<b>Beta-Binomial Presmoothing</b>	<b>93</b>
9.1	Four-parameter Beta Compound Binomial Model . . . . .	93
9.2	Special Cases of the Four-parameter Beta Compound Binomial Model . . . . .	94
9.3	Actual and Fitted Observed Score Densities . . . . .	95
9.4	The Beta-binomial Model Applied to Equating . . . . .	95
9.5	Functions . . . . .	96
9.5.1	Wrapper_Smooth_BB() and Print_BB() . . . . .	96
9.5.2	Wrapper_RB() . . . . .	97
9.6	Code for Wrapper_Bootstrap() . . . . .	98
9.7	Example . . . . .	98
9.7.1	Presmoothing . . . . .	100
9.7.2	Equipercentile Equating . . . . .	100
9.7.3	Bootstrap Standard Errors . . . . .	100
<b>10</b>	<b>Log-Linear Presmoothing</b>	<b>109</b>
10.1	General Discussion of Log-linear Smoothing . . . . .	109
10.1.1	Model . . . . .	110
10.1.2	Maximum Likelihood Estimation . . . . .	111
10.1.3	Newton-Raphson Method . . . . .	112
10.2	Univariate Smoothing in Equating . . . . .	113
10.2.1	Scaling the Design Matrix . . . . .	114

10.2.2	Raw and Scaled Moments . . . . .	114
10.2.3	Estimation Options . . . . .	115
10.3	Bivariate Smoothing in Equating . . . . .	116
10.3.1	Cross-products and their Moments . . . . .	117
10.3.2	Issues Specific to Equating . . . . .	117
10.4	Functions . . . . .	118
10.4.1	Univariate Smoothing: <code>Wrapper_Smooth_ULL()</code> and <code>Print_ULL()</code> . . . . .	118
10.4.2	Bivariate Smoothing: <code>Wrapper_Smooth_BLL()</code> and <code>Print_BLL()</code> . . . . .	119
10.4.3	Random Groups Design: <code>Wrapper_RL()</code> and <code>Print_RL()</code> . . . . .	120
10.4.4	Single Group Design: <code>Wrapper_SL()</code> and <code>Print_SL()</code> . . . . .	121
10.4.5	Common-item Nonequivalent Groups Design: <code>Wrapper_CL()</code> and <code>Print_CL()</code> . . . . .	122
10.5	Example: Random Groups . . . . .	124
10.5.1	Presmoothing . . . . .	124
10.5.2	Equipercetile Equating . . . . .	124
10.5.3	Bootstrap Standard Errors . . . . .	125
10.6	Example: Common-item Nonequivalent Groups . . . . .	137
10.6.1	Presmoothing . . . . .	137
10.6.2	Equipercetile Equating . . . . .	138
10.6.3	Bootstrap Standard Errors . . . . .	138
<b>11</b>	<b>Cubic Spline Postsmoothing</b> . . . . .	<b>151</b>
11.1	Smoothing Cubic Splines . . . . .	152
11.1.1	Solution Method for Smoothing Cubic Splines . . . . .	153
11.1.2	Description of the Algorithm . . . . .	156
11.2	Functions . . . . .	157
11.2.1	<code>Wrapper_Smooth_CubSpl()</code> . . . . .	157
11.2.2	<code>Print_CubSpl()</code> . . . . .	159
11.2.3	Other Functions . . . . .	160
11.3	Random Groups Example . . . . .	162
<b>12</b>	<b>Kernel Equating</b> . . . . .	<b>171</b>
12.1	The Kernel Equating Framework . . . . .	171
12.2	Functions for the Random Groups Design . . . . .	174
12.2.1	<code>Wrapper_RK()</code> . . . . .	174
12.2.2	<code>Print_RK()</code> . . . . .	175
12.2.3	Example . . . . .	175
12.3	Functions for the Single Group Design . . . . .	176
12.3.1	<code>Wrapper_SK()</code> . . . . .	176
12.3.2	<code>Print_SK()</code> . . . . .	177
12.3.3	Example . . . . .	177

12.4 Functions for the Common-Item Nonequivalent Groups Design	177
12.4.1 <code>Wrapper_CK()</code> . . . . .	178
12.4.2 <code>Print_CK()</code> . . . . .	179
12.4.3 Example . . . . .	179
<b>13 Standard Error of Kernel Equating</b>	<b>187</b>
13.1 Standard Error of Kernel Equating Based on the Delta Method	187
13.2 Functions for Random Groups Design . . . . .	189
13.2.1 <code>KernelEquateSEERG()</code> . . . . .	190
13.2.2 Example . . . . .	190
13.3 Functions for the Single Group Design . . . . .	193
13.3.1 <code>KernelEquateSEESG()</code> . . . . .	194
13.3.2 Example . . . . .	194
13.4 Functions for the Common-Item Nonequivalent Groups Design	194
13.4.1 <code>KernelEquateSEENEATPS()</code> . . . . .	195
13.4.2 <code>KernelEquateSEENEATChn()</code> . . . . .	195
13.4.3 Example . . . . .	196
<b>14 Continuized Log-Linear Equating</b>	<b>201</b>
14.1 Continuized Log-linear Method for the Random Groups Design	202
14.2 Single Group, Counter Balanced Design . . . . .	203
14.3 Common-Item Nonequivalent Groups Design . . . . .	204
14.4 Standard Errors for CLL Equating under the Random Groups Design . . . . .	205
14.5 Functions for the Random Groups Design . . . . .	207
14.5.1 <code>Wrapper_RC()</code> . . . . .	207
14.5.2 <code>Print_RC()</code> . . . . .	208
14.5.3 Example . . . . .	208
14.6 Functions for the Single Group Design . . . . .	208
14.6.1 <code>Wrapper_SC()</code> . . . . .	209
14.6.2 <code>Print_SC()</code> . . . . .	210
14.6.3 Example . . . . .	210
14.7 Functions for the Common-Item Nonequivalent Groups Design	210
14.7.1 <code>Wrapper_CC()</code> . . . . .	211
14.7.2 <code>Print_CC()</code> . . . . .	212
14.7.3 Example . . . . .	213
<b>15 Scale Transformation using Item Response Theory</b>	<b>223</b>
15.1 IRT Models . . . . .	224
15.1.1 Three-Parameter Logistic Model . . . . .	224
15.1.2 Graded Response Model . . . . .	225
15.1.3 Nominal Response Model . . . . .	225
15.1.4 Generalized Partial Credit Model . . . . .	226
15.2 IRT Scale Transformation Methods . . . . .	226
15.2.1 Overview of IRT Scale Transformation . . . . .	227

15.2.2	Moment Methods . . . . .	228
15.2.3	Characteristic Curve Methods . . . . .	230
15.3	Functions . . . . .	236
15.3.1	<code>Wrapper_IRTst()</code> . . . . .	236
15.3.2	Functions for Set-up and Clean-up . . . . .	238
15.3.3	Functions for the Moment Methods . . . . .	242
15.3.4	Functions for the Characteristic Curve Methods . . . . .	243
15.3.5	A Utility Function: <code>ScaleTransform</code> . . . . .	246
15.4	Examples . . . . .	247
15.4.1	Mixed Format . . . . .	247
15.4.2	3PL . . . . .	253
<b>16</b>	<b>Test Equating Using Item Response Theory</b>	<b>257</b>
16.1	Basic Concepts . . . . .	257
16.1.1	Scale Transformation as Equating . . . . .	258
16.1.2	Definition of True Scores in IRT . . . . .	258
16.1.3	Observed Score Distributions Generated by IRT . . . . .	259
16.2	Equating Methods . . . . .	260
16.2.1	IRT True Score Equating . . . . .	261
16.2.2	IRT Observed Score Equating . . . . .	262
16.3	Functions . . . . .	263
16.3.1	<code>Wrapper_IRTeq()</code> . . . . .	264
16.3.2	<code>Print_IRTeq()</code> . . . . .	266
16.3.3	<code>Print_ESS_QD()</code> . . . . .	266
16.3.4	Set-up and Clean-up . . . . .	267
16.3.5	<code>trueScoreEq()</code> . . . . .	269
16.3.6	<code>IRTmixObsEq()</code> . . . . .	270
16.4	Examples . . . . .	271
16.4.1	Mixed Format . . . . .	272
16.4.2	3PL . . . . .	281
	<b>References</b>	<b>285</b>
	<b>Index of Wrapper and Print Functions</b>	<b>291</b>



# List of Tables

1.1	Page References for Wrapper and Associated Print Functions	8
2.1	Main() Code to Illustrate Utilities	28
2.2	Output for struct USTATS x (ACT Math)	30
2.3	Output for $X$ in struct BSTATS xv using mondatx	32
2.4	Output for $V$ in struct BSTATS xv using mondatx	33
2.5	Output for Bivariate Frequency Distribution for struct BSTATS xv Using mondatx	34
2.6	Output for Bivariate Relative Frequency Distribution for struct BSTATS xv Using mondatx	35
3.1	Main() Code to Illustrate Linear Equating with Random Groups Design	40
3.2	Linear Equating Raw-Score Results for the Random Groups Design	42
3.3	Scale Score Conversion Table for Old Form Y	43
3.4	Equated Unrounded Scale Scores for Linear Equating with the Random Groups Design	44
3.5	Equated Rounded Scale Scores for Linear Equating with the Random Groups Design	45
4.1	Main() Code to Illustrate Linear Equating with Common- item Nonequivalent Groups Design	55

4.2	Linear Equating Raw-Score Results for the Common-items Nonequivalent Groups Design . . . . .	56
5.1	Main() Code to Illustrate Equipercentile Equating with Ran- dom Groups Design . . . . .	59
5.2	Equipercentile Equating Raw-Score Results for the Random Groups Design . . . . .	61
5.3	Equated Unrounded Scale Scores for Equipercentile Equat- ing with the Random Groups Design . . . . .	62
5.4	Equated Rounded Scale Scores for Equipercentile Equating with the Random Groups Design . . . . .	63
6.1	Main() Code to Illustrate Equipercentile Equating with Common- item Nonequivalent Groups Design . . . . .	70
6.2	Equipercentile Equating Raw-Score Results for the Common- item Nonequivalent Groups Design . . . . .	71
6.3	Synthetic Densities of $X$ and $Y$ Under Frequency Estimation	72
6.4	Synthetic Densities of $X$ and $Y$ Under Modified Frequency Estimation . . . . .	73
7.1	Main() Code to Illustrate Estimating Analytic Raw-Score Standard Errors for Equipercentile Equating with Random Groups Design . . . . .	78
7.2	Estimated Analytic Raw-Score Standard Errors for Equiper- centile Equating with Random Groups Design . . . . .	79
8.1	Main() Code to Illustrate Estimating Bootstrap Standard Errors for Equipercentile Equating with Random Groups Design . . . . .	87
8.2	Estimated Bootstrap Raw Score Standard Errors for Equiper- centile Equating with Random Groups Design . . . . .	88
8.3	Estimated Bootstrap Unounded Scale Score Standard Errors for Equipercentile Equating with Random Groups Design .	91
8.4	Estimated Bootstrap Rounded Scale Score Standard Errors for Equipercentile Equating with Random Groups Design .	92
9.1	Main() Code to Illustrate Beta-binomial Presmoothing for Equipercentile Equating with Random Groups Design . . .	99
9.2	Output for Beta-binomial Smoothing for Form X . . . . .	101
9.3	Output for Beta-binomial Smoothing for Form Y . . . . .	102
9.4	Output for Equated Raw Scores using Beta-binomial Smooth- ing . . . . .	103
9.5	Output for Unrounded Equated Scale Scores using Beta- binomial Smoothing . . . . .	104

9.6	Output for Rounded Equated Scale Scores using Beta-binomial Smoothing . . . . .	105
9.7	Output for Bootstrap Estimates of Standard Errors for Equated Raw Scores using Beta-binomial Smoothing . . . . .	106
9.8	Output for Bootstrap Estimates of Standard Errors for Unrounded Equated Scale Scores using Beta-binomial Smoothing	107
9.9	Output for Bootstrap Estimates of Standard Errors for Rounded Equated Scale Scores using Beta-binomial Smoothing . . . .	108
10.1	Main() Code to Illustrate Log-linear Presmoothing for Equipercentile Equating with Random Groups Design . . . . .	126
10.2	RG Output for Log-linear Presmoothing for Form X: General Information and Design Matrices . . . . .	127
10.3	RG Output for Log-linear Fitted Frequencies and Proportions for Form X . . . . .	128
10.4	RG Output for Log-linear Presmoothing for Form Y: General Information and Design Matrices . . . . .	129
10.5	RG Output for Log-linear Fitted Frequencies and Proportions for Form Y . . . . .	130
10.6	RG Output for Equated Raw Scores using Log-linear Smoothing . . . . .	131
10.7	RG Output for Unrounded Equated Scale Scores using Log-linear Smoothing . . . . .	132
10.8	RG Output for Rounded Equated Scale Scores using Log-linear Smoothing . . . . .	133
10.9	RG Output for Bootstrap Estimates of Standard Errors for Equated Raw Scores using Log-linear Smoothing . . . . .	134
10.10	RG Output for Bootstrap Estimates of Standard Errors for Unrounded Equated Scale Scores using Log-linear Smoothing	135
10.11	RG Output for Bootstrap Estimates of Standard Errors for Rounded Equated Scale Scores using Log-linear Smoothing	136
10.12	Main() Code to Illustrate Log-linear Presmoothing for Equipercentile Equating with Common-item Nonequivalent Groups Design . . . . .	139
10.13	CG Output for Log-linear Presmoothing for Form X: General Information . . . . .	140
10.14	CG Output for Log-linear Presmoothing for Form X: Observed Frequencies . . . . .	141
10.15	CG Output for Log-linear Presmoothing for Form X: Fitted Frequencies . . . . .	142
10.16	CG Output for Log-linear Presmoothing for Form X: Moments	143
10.17	CG Output for Log-linear Presmoothing for Form X: Fitted Bivariate Frequency Distribution . . . . .	144
10.18	CG Output for Log-linear Presmoothing for Form X: Fitted Marginal Distributions of $X$ and $V$ in Population 1 . . . . .	145

10.19CG Output for Log-linear Presmoothing for Form Y: Fitted Marginal Distributions of $Y$ and $V$ in Population 2 . . . . .	146
10.20CG Output for Log-linear Raw-score Equating: General In- formation . . . . .	147
10.21CG Output for Log-linear Raw-score Equating . . . . .	148
10.22CG Output for Bootstrap Estimates of Standard Errors for Log-linear Raw-score Equating . . . . .	149
11.1 Main() Code to Illustrate Cubic Spline Postsmoothing with Random Groups Design . . . . .	165
11.2 Output for Equated Raw Scores for Cubic Spline Smoothing that puts $X$ on the Scale of $Y$ . . . . .	166
11.3 Output for Equated Raw Scores for Cubic Spline Smoothing that puts $Y$ on the Scale of $X$ . . . . .	167
11.4 Output for Equated Raw Scores for Cubic Spline Smoothing	168
11.5 Output for Unrounded Equated Scale Scores using Cubic Spline Smoothing . . . . .	169
12.1 Main() Code to Illustrate Kernel Equating with the Random Groups Design . . . . .	180
12.2 Output for Equated Raw Scores using Kernel Equating Un- der a RG Design . . . . .	181
12.3 Main() Code to Illustrate Kernel Equating with the Single Group Design . . . . .	182
12.4 Output for Equated Raw Scores using Kernel Equating Un- der a SG Design . . . . .	183
12.5 Main() Code to Illustrate Kernel Equating with the CINEG Design . . . . .	184
12.6 Output for Equated Raw Scores using Kernel Equating Un- der a CINEG Design . . . . .	185
13.1 Main() Code to Illustrate Kernel Equating with the Random Groups Design . . . . .	191
13.2 Output for Kernel Equating SEE Under a RG Design . . .	192
13.3 Main() Code to Illustrate Kernel Equating SEE with the Single Group Design . . . . .	197
13.4 Main() Code to Illustrate Kernel Equating SEE with the CINEG Design . . . . .	198
13.5 Analytical and Bootstrap SEE for Frequency Estimation and Chained Equipercntile Kernel Equating under a CINEG Design . . . . .	199
14.1 Main() Code to Illustrate CLL Equating with the Random Groups Design . . . . .	214

14.2	Output for Equated Raw Scores using CLL Equating Under a RG Design . . . . .	215
14.3	Main() Code to Illustrate CLL Equating with the Single Group Design . . . . .	217
14.4	Output for Equated Raw Scores using CLL Equating Under a SG Design . . . . .	218
14.5	Main() Code to Illustrate CLL Equating with the CINEG Design . . . . .	219
14.6	Output for Equated Raw Scores using CLL Equating Under a CINEG Design . . . . .	220
15.1	Main() Code to Illustrate IRT Scale Transformation for Three Mixed-Format Dummy Data Examples . . . . .	248
15.2	Output Illustrating IRT Scale Transformation for Three Mixed-Format Dummy Data Examples . . . . .	251
15.3	Contents of Files Created for Mixed-Format Dummy Data Example . . . . .	252
15.4	Main() Code to Illustrate IRT Scale Transformation for Kolen and Brennan (2004, chap. 6) Example . . . . .	254
15.5	Output Illustrating IRT Scale Transformation for Kolen and Brennan (2004, chap. 6) Example . . . . .	255
15.6	Contents of Files Created for Kolen and Brennan (2004, chap. 6) Example . . . . .	256
16.1	Main() Code to Illustrate IRT True-Score and Observed-Score Equating for a Mixed-Format Dummy Data Example) Example . . . . .	274
16.2	Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example . . . . .	275
16.3	Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: IRT Fitted Distributions and Moments . . . . .	276
16.4	Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Scale Score Conversion Table . . . . .	277
16.5	Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Unrounded Scale Scores . . . . .	278
16.6	Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Rounded Scale Scores . . . . .	279
16.7	Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Scale Score Moments Based on IRT Fitted Distributions . . . . .	280

16.8	Main() Code to Illustrate IRT True-Score and Observed-Score Equating for Kolen and Brennan (2004, chap. 6) Example . . . . .	282
16.9	Output Illustrating IRT True-Score and Observed-Score Equating for Kolen and Brennan (2004, chap. 6) Example . . . .	283
16.10	Output Illustrating IRT True-score and Observed-score Equating for Kolen and Brennan (2004, chap. 6) Example: IRT Fitted Distributions and Moments . . . . .	284

# List of Figures

1.1	<i>Equating Recipes</i> wrapper functions. . . . .	6
8.1	Estimated standard errors for delta method and bootstrap. . . . .	89
11.1	Cubic spline algorithm. . . . .	157
12.1	Kernel and percentile-rank based equipercentile equating under a RG Design. . . . .	175
12.2	Kernel and percentile-rank based equating using the frequency estimation and chained equipercentile methods under a CINEG design. . . . .	186
13.1	SEEs of Kernel and Percentile-Rank Based Equipercentile Equating under a RG Design . . . . .	193
14.1	CLL, Kernel and Percentile-Rank Based Equipercentile Equating Functions under a RG Design . . . . .	216
14.2	Kernel, percentile-rank based, and CLL equating functions using the frequency estimation method under a CINEG design. . . . .	221
14.3	Kernel, percentile-rank based, and CLL equating functions using the chained equipercentile method under a CINEG design. . . . .	221





# 1

## Introduction

*Equating Recipes* provides a set of open-source functions written in ANSI C to perform all types of equating discussed by Kolen and Brennan (2004), as well as some other equating methods. The *Equating Recipes* code and this monograph are freely available on [www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma), which is the website for the Center for Advanced Studies in Measurement and Assessment (CASMA) at the University of Iowa. Typical copyright rules apply. The code is available under the open-source license conditions discussed in Section 1.5. Older versions of some of the *Equating Recipes* functions are used in equating programs that have been distributed by Kolen and Brennan for many years—see the CASMA website.

The C language (Kernighan & Ritchie, 1988) has been chosen because of its simplicity, flexibility, and likely longevity. The *Equating Recipes* code is copyrighted, but the code itself is open source in the sense discussed in the appendix to this chapter. *Equating Recipes* is provided in a folder named **ER for distribution (date)**. This folder contains: (a) source files and header files; (b) the Microsoft Visual Studio C++ project;<sup>1</sup> (c) this monograph and a **ReadMe** file; and (d) main functions, data files, conversion tables, and output files for the examples in this monograph. There are over 25,000 lines of code in *Equating Recipes*, including comments. To our knowledge, nothing comparable to *Equating Recipes* currently exists.

---

<sup>1</sup>Microsoft Visual Studio does not distinguish between C and C++ projects. Of course, other compilers could be used.

The *Equating Recipes* C functions and this monograph are highly coordinated with Kolen and Brennan (2004). Users of these functions will need to be relatively proficient in C and very familiar with Kolen and Brennan (2004). Those who wish to modify or extend these functions will need to be quite proficient in C.

Unless noted otherwise, *Equating Recipes* uses the notation in Kolen and Brennan (2004), and the notation is not always fully described each time it is used. It is particularly important to recall that in Kolen and Brennan (2004)  $X$  refers to the raw-score random variable for the new form of a test that was administered to the new population (or group) 1, and  $Y$  refers to the raw-score random variable for the old form that was administered to the old population (or group) 2. This is only one of a number of ways in which *Equating Recipes* is coordinated with Kolen and Brennan (2004). There are other excellent treatments of equating (particularly Holland & Dorans, 2006), but they are not always as closely coordinated with the notation and terminology used in *Equating Recipes*.

Immediately following the preface on page ix is a section entitled “Versions, Revisions, and Notes.” This section provides brief explanations of the revisions associated with each version of *Equating Recipes*. Any significant changes in the *Equating Recipes* code also will also listed in this section, even if they do not cause changes in the content of this monograph.

## 1.1 Purposes

*Equating Recipes* has been developed for three primary purposes—to enhance what might be called “equating transparency,” to provide tools for researchers, and to provide a way to estimate bootstrap standard errors for raw and scale scores for most equating procedures.<sup>2</sup> Secondly, with some important caveats, the functions in *Equating Recipes* might be used as part of an equating system for a specific testing program.

Far too frequently operational equating is essentially a black box whose contents are not accessible to those who wish to understand at a deep level exactly how equating was performed in a specific context. In this sense, operational equating is almost always opaque. *Equating Recipes* seeks to help make equating more transparent by providing computer code, along with explanations and examples, that operationalizes equating procedures, as discussed in Kolen and Brennan (2004) and elsewhere, and that is freely available to any person or entity.<sup>3</sup>

---

<sup>2</sup>The bootstrap wrapper function discussed in Chapter 8 could be adapted for use with any equating procedure, but doing so with IRT procedures would involve extraordinarily extensive computations.

<sup>3</sup>Users should not assume that the equating procedures used by any specific company match those in *Equating Recipes*. Virtually all testing companies treat their equating

This opaque feature of almost all operational equating may be more problematic than the public perceives. In particular, in almost all large-scale testing programs, the accuracy of an examinee's reported score is very directly based on the accuracy of equating. Yet, when an examinee challenges the accuracy of a reported score and requests a re-score, that re-score almost never involves a reconsideration of the equating, let alone a replication of it. Indeed, it is relatively rare for equating to be replicated by an independent party or entity under any circumstances. The existence of *Equating Recipes* will not solve this problem, per se, but it does provide tools (both open-source code and this monograph) that could be used to help make equating more transparent.

The last two decades have witnessed a tremendous increase in the amount of research on equating. However, the computer code actually used to perform equating is often not available to other researchers who seek to understand, replicate, or extend such research. Consequently, in many cases, the users of equating research must assume that the reported equating results are defensible, without being able to verify them independently. Furthermore, researchers who seek to study or extend some particular equating procedure must “start from scratch” or use “canned” programs that hide the actual computer code. Clearly, this is a considerably less than optimal situation for conducting research. Hopefully, the availability of *Equating Recipes* will help mitigate these problems for researchers.

There are numerous papers that have been written about standard errors of equating (see, for example, Kolen & Brennan, 2004, chap. 7). Generally speaking, each such paper provides formulas for estimating standard errors for equated raw scores for a particular procedure, a particular design, and a particular set of assumptions. It is important to note that almost always these procedures provide standard error estimates for raw scores, rather than the scale scores (often rounded) actually reported to examinees and used as a basis for decisions. As discussed in Kolen and Brennan (2004, pp. 235–245), in principle the bootstrap can be used to estimate standard errors for raw scores or scale scores for any equating procedure. *Equating Recipes* provides functions to facilitate doing so.<sup>4</sup>

*Equating Recipes* is not an equating system, per se, although the functions provided certainly could be used as part of an operational equating system or to replicate equating results obtained from an operational system. Typically, an equating system is highly specific to a particular testing program. Obviously, the system must include functions that compute re-

---

systems as proprietary. Companies are generally unwilling to share their code used for equating, and they are often unwilling to share even the details of the algorithms used.

<sup>4</sup>Again, the bootstrap wrapper function discussed in Chapter 8 could be adapted for use with any equating procedure, but doing so with IRT procedures would involve extraordinarily extensive computations.

sults for particular equating procedures, but, in addition, answers to many questions must be considered, such as:

- For which methods and types of smoothing (if any) will equating results be obtained?
- What criteria will be used to select a particular method and type of smoothing?
- How many links will be used?
- How will results be “averaged” over links?
- What kinds of tabular and/or graphical capabilities will be needed to examine results?
- What, if any, comparisons will be made between current equating results and past results?
- What files need to be produced as input to the score reporting system?
- What files need to be archived for future use?

The goal of an equating system is to facilitate answering these questions and any others deemed relevant. Since almost every testing program tends to deal with these (and other) questions in a somewhat different way, it necessarily follows that a generic equating system, applicable to numerous testing programs, is virtually impossible to develop without creating an extraordinarily complex “entity.” It is believed that *Equating Recipes* has a kind of modularity and “grain size” that could facilitate creating various parts of an equating system for a specific program.

## 1.2 Framework

*Equating Recipes* employs a framework that has three components: (a) categorization schema used to characterize equating procedures; (b) wrapper functions that provide relatively easy access to the equating functions; and (c) structures that facilitate passing data to and from the wrappers. Each of these components is described next.

### 1.2.1 Categorization Schema

Most of the functions in *Equating Recipes* generate equating results for specific designs (D), methods (M), and types of smoothing (S) (if any), which we sometimes abbreviate D/M/S. The types of designs are:

- random groups (R or RG);
- single group (S or SG); and
- common-item nonequivalent groups (C or CG).

The types of methods are:

- mean (M);
- linear (L);
- equipercentile (various versions labelled E, F, G, C, H, and A as described later); and
- item response theory (IRT).

The types of smoothing are:

- no smoothing (N);
- two or -four parameter beta (compound) binomial presmoothing (B);
- log-linear presmoothing (L);
- cubic-spline postsmoothing (S);
- kernel (K); and
- continuized log-linear (Z).

The abbreviations (especially the single-letter abbreviations) are often used in this monograph and the functions.<sup>5</sup>

In essence this D/M/S categorization schema provides a kind of advanced organizer for most of the functions in *Equating Recipes*. There are other functions that facilitate data input, that organize data input, or that provide output.

Clearly, one of the elements of the D/M/S categorization schema is “method” (M), as described above. Usually, the word “procedure” is used to characterize any single instance of the D/M/S categorization schema. In this sense, for example, with the common-item non-equivalent groups design, the equipercentile method with no smoothing is a “procedure.” Sometimes, however, “method” is used in the sense of “procedure.” The context makes the intent clear.

---

<sup>5</sup>It is somewhat imprecise to characterize the kernel method as a smoothing method, but it does involve smoothing, as discussed in Chapter 12. A similar statement applies to the continuized log-linear method.

	No Smoothing			Presmoothing	
Design	M	L	E	Beta-Binomial	Log-Linear
RG	Wrapper_RN()			Wrapper_RB()	Wrapper_RL()
SG	Wrapper_SN()				Wrapper_SL()
CG	Wrapper_CN()				Wrapper_CL()
Wrapper_Smooth_BB():			beta-binomial smoothing		
Wrapper_Smooth_ULL():			univariate log-linear smoothing		
Wrapper_Smooth_BLL():			bivariate log-linear smoothing		
Wrapper_Smooth_CubSpl():			cubic-spline postsmoothing		
Wrapper_ESS();			equated scale scores		
Wrapper_Bootstrap();			raw- and scale-score bootstrap standard errors		
Design	Kernel			Continuized Log-Linear	
RG	Wrapper_RK()			Wrapper_RC()	
SG	Wrapper_SK()			Wrapper_SC()	
CG	Wrapper_CK()			Wrapper_CC()	
Wrapper_IRTst():			IRT scale transformation		
Wrapper_IRTeq():			IRT equating		

FIGURE 1.1. *Equating Recipes* wrapper functions.

### 1.2.2 Wrapper Functions

The principal functions that perform equating often have lengthy and complex argument lists involving variables, vectors, and matrices (or pointers to such elements). To simplify calling these functions *Equating Recipes* uses the wrapper functions listed in Figure 1.1. The argument lists for the wrapper functions generally consist of structures (discussed later) and a few input variables.

The wrapper functions in rectangles in Figure 1.1 have design-specific information in their names. For the wrapper functions at the top of Figure 1.1, the first letter after the underscore indicates the design, and the second letter indicates the type of presmoothing, if any. For the wrapper functions in rectangles near the bottom of Figure 1.1, the second letter indicates kernel equating or continuized log-linear equating.

The first three wrapper functions not in rectangles perform presmoothing without any equating. In operational equating contexts, usually different types and/or degrees of presmoothing are examined before selecting one particular presmoothing to be used for equating. These wrapper functions are design specific in the sense that `Wrapper_Smooth_BB()` and `Wrapper_Smooth_ULL()` are associated with the random groups design, and `Wrapper_Smooth_BLL()` is associated with both the single group design and the common-item nonequivalent groups design.

The next three wrapper functions in the middle of Figure 1.1 are not design specific. For example, `Wrapper_ESS()` computes equated scale scores for any D/M/S procedure using equated raw scores as input along with the raw-to-scale score conversion table for the old Form Y.

The last two wrapper functions are specific to IRT. The first does IRT scale transformation, which is employed (usually) with the common-item non-equivalent groups design. The second performs IRT true-score and observed-score equating assuming item parameter estimates and ability distributions are on the same scale. The scale transformation functions can handle multiple IRT models as encountered, for example, with mixed-format tests. Note that *Equating Recipes* does *not* provide functions to estimate item parameters.

The wrapper functions have the distinct advantage of making it relatively easy to use the *Equating Recipes* functions, since the wrapper functions hide a great deal of complexity from the user. However, the wrapper functions are at least one step removed from the principal functions that perform the equating computations. So, those who want to study computational algorithms used by *Equating Recipes* will need to drill down through the wrapper functions.

A distinguishing feature of the wrapper functions is their extensive use of structures. By contrast, the functions that perform most of the computations do not make nearly as much use of structures. So, users who wish to modify computational methods or add a new method will not need to grapple too much with structures.

The use of wrapper functions makes the *Equating Recipes* organizational structure somewhat resemble procedures (as in SAS or SPSS). The numerous functions for implementing particular equating methods makes *Equating Recipes* somewhat resemble a library of equating functions. This dual perspective is intentional and hopefully helpful for different users. Although we believe the wrapper functions will be helpful in many contexts, we suspect that some users will prefer to write their own code to call *Equating Recipes* functions that directly implement particular equating procedures.

There are print functions associated with each wrapper function. Table 1.1 lists the page numbers where the wrapper and print functions are described. One purpose of the print functions is to display detailed results

TABLE 1.1. Page References for Wrapper and Associated Print Functions

Wrapper	page	Print	page
Wrapper_RN()	38	Print_RN()	38, 58
Wrapper_SN()	39	Print_SN()	39, 58
Wrapper_CN()	50	Print_CN()	53, 68
		Print_SynDens()	69
Wrapper_RB()	97	Print_RB()	98
Wrapper_RL()	121	Print_RL()	121
Wrapper_SL()	122	Print_SL()	122
Wrapper_CL()	123	Print_CL()	124
		Print_SynDens()	69
Wrapper_ESS()	23	Print_ESS()	25
Wrapper_Bootstrap()	82	Print_Boot_se_eraw()	85
		Print_Boot_se_ess()	85
Wrapper_Smooth_BB()	96	Print_BB()	96
Wrapper_Smooth_ULL()	118	Print_ULL()	119
Wrapper_Smooth_BLL()	119	Print_BLL()	120
Wrapper_Smooth_CubSpl()	157	Print_CubSpl()	159
Wrapper_RK()	174	Print_RK()	175
Wrapper_SK()	176	Print_SK()	177
Wrapper_CK()	178	Print_CK()	179
Wrapper_RC()	207	Print_RC()	208
Wrapper_SC()	209	Print_SC()	210
Wrapper_CC()	211	Print_CC()	212
Wrapper_IRTst()	237	wrapper does own printing	
Wrapper_IRTeq()	264	Print_IRTeq()	266
		Print_ESS_QD()	266

for the extensive set of examples in this monograph.<sup>6</sup> In almost all cases, these examples are the same ones used in Kolen and Brennan (2004). It is likely that most users of *Equating Recipes* will want to produce their own print functions tailored to their own needs. For such users, the provided print functions may still be of use since they illustrate how to access just about every input and output variable, vector, etc.

<sup>6</sup>Sometimes, because of space limitations, in *Equating Recipes* the output is displayed in a slightly different format from how it is actually printed by the print function.



### 1.2.3 Structures

Use of the *Equating Recipes* equating functions is greatly facilitated through structures. A structure in C is a collection of variables (called members), possibly of different types, grouped together under a single name. Structures are used extensively in *Equating Recipes* to pass sets of data from one function to another, especially when wrapper functions are involved. In *Equating Recipes* the structure declarations are in `ERutilities.h`. As illustrated by the examples in this monograph, the user provides the variable names associated with the structures.

The declarations in `ERutilities.h` provide a commented description of each of the members in each of the structures. Provided next is a list of most of the non-IRT structures along with a brief description of the structures:

- **USTATS**: raw-score statistics for a univariate distribution
- **BSTATS**: raw-score statistics for a bivariate distribution
- **PDATA**: input for a particular D/M/S schema as well as other data passed among functions
- **ERAW\_RESULTS**: equated raw score results
- **ESS\_RESULTS**: equated scale score results
- **BB\_SMOOTH**: input and output for beta-binomial smoothing
- **ULL\_SMOOTH**: input and output for univariate log-linear smoothing
- **BLL\_SMOOTH**: input and output for bivariate log-linear smoothing
- **CS\_SMOOTH**: input and output for cubic spline postsMOOTHing
- **BOOT\_ERAW\_RESULTS**: equated raw score results for bootstrap
- **BOOT\_ESS\_RESULTS**: equated scale score results for bootstrap

See Chapters 15 and 16 for additional structures used with IRT.

The structures **USTATS** and **BSTATS** play particularly important roles in any wrapper function that is design specific (see Figure 1.1). A wrapper function involving the random groups (R) design requires two **USTATS** structures—one for the new Form X and one for the old Form Y. A wrapper function involving the single group design (S) requires one **BSTATS** structure for new Form X and old Form Y. A wrapper function involving the common-item nonequivalent groups design (C) requires two **BSTATS** structures—one for new Form X and its associated common-items V in population 1, and one for the old Form Y and its associated V in population 2.

After a wrapper function completes execution, just about all input data, input parameters, and summary statistics are contained in a **PDATA** structure, all equating results are in an **ERAW\_RESULTS** structure, and all equated

scale score results are in an `ESS_RESULTS` structure. This means that the user has direct access to not only final equating results but also most intermediate results that are likely to be of interest. The price for this benefit is, of course, extensive use of storage, but with today's computers this is a relatively minor concern.

It is *strongly* advised that users *not* reuse any particular structure. So, for example, if a particular wrapper function populates a `PDATA` structure `aaa` and an `ERAW_RESULTS` structure `bbb`, then some other wrapper function should *not* populate `PDATA` and `ERAW_RESULTS` structures with the names `aaa` and `bbb`, respectively.

Generally, each wrapper function has an associated `PDATA` structure that is associated with an `ERAW_RESULTS` structure. The `PDATA` structure also may be associated with other structures such as an `ESS_RESULTS` structure (if scale score results are requested). It is advisable to choose structure names that are similar enough to reflect such "linkages." This advice is followed in the examples found in the various chapters of this monograph.

### 1.3 Use and Limitations

No warranties are made, expressed or implied, that the *Equating Recipes* computer code is free of errors, that it is consistent with any particular standard, or that it will meet the requirements of any particular application. The authors disclaim any direct or consequential damages resulting from use of the code. Similarly, the authors make no claim that the *Equating Recipes* monograph is free of errors.

The authors do not claim that the functions provided in *Equating Recipes* are necessarily the best that could be developed. However, they have the distinct virtues of being publicly available, free, and coordinated with Kolen and Brennan (2004). We encourage others to implement improvements in these functions as they see fit, consistent with the terms of the license in the appendix to this chapter.

The *Equating Recipes* code has only a limited amount of built-in error checking. In our experience, this is usually adequate for research purposes, but considerably more error checking could be incorporated and may be warranted in certain circumstances.

We have made no great efforts to write code that is optimal in the sense of maximizing speed and/or minimizing use of storage. With today's computers, for the types of computations done in equating, such optimality issues are usually relatively unimportant from a practical point of view.

Each of the authors of the *Equating Recipes* code has substantial programming experience, but most of us are not professional programmers, primarily; rather, we are psychometricians for whom computer programming is a tool. Also, although we have attempted to impose a degree of

uniformity across chapters and code, the *Equating Recipes* functions and chapters written by different authors tend to have different “flavors.”

A question frequently addressed to us is, “Why did you pick C rather than some other language?” As noted earlier, the C language has been chosen because of its simplicity, flexibility, and likely longevity. Importantly, it is our guess (perhaps time will prove us wrong!) that no matter how programming languages evolve in the near future, the *Equating Recipes* functions will still work—or can be made to work—with C-type compilers, or it will be relatively easy to convert the *Equating Recipes* C functions to other languages. We encourage such conversion efforts now or in the future, consistent with the conditions noted in the first two paragraphs of this section.

It is our plan to revise the computer code and the *Equating Recipes* monograph on a periodic basis. We invite anyone using *Equating Recipes* to suggest revisions, but we implore such persons to be reasonable in their expectations about if/when such revisions will be made. Since the code is open source, any one is free to modify it; there is no need to depend on revisions by the authors.

## 1.4 Additional Comments and Conventions

It is strongly recommended that all users of *Equating Recipes* study, or at least skim, the content of Chapter 2, which discusses utility functions used in *Equating Recipes*. Even a superficial understanding of these utility functions greatly facilitates understanding how the *Equating Recipes* code implements various equating procedures. In particular, users are strongly advised to study the example in Section 2.2 on page 27.

For the most part, starting with Chapter 3, each chapter follows the same general outline. There is a brief introduction of the procedure (i.e., an instance of the D/M/S schema), including a reference to relevant parts of Kolen and Brennan (2004). This is followed by a description of the procedure that is often more detailed (and sometimes less detailed) than that provided by Kolen and Brennan (2004). Then the wrapper and print function for the procedure are described. In some cases, functions called by wrapper functions are described, as well. Just about all functions in the *Equating Recipes* code are quite heavily commented; so, there is considerable redundancy in the comments and the descriptions provided in *Equating Recipes*. Finally, at least one sample `main()` function is provided that usually uses data from Kolen and Brennan (2004) to illustrate use of the function(s).

### 1.4.1 Typesetting Conventions

In this monograph the *Equating Recipes* names of functions, files, headers, and variables are all typeset in typewriter style (the `\ttfamily` in  $\text{\LaTeX}$ ). Also, with minor exceptions C code and output are provided in typewriter style, whether the code is “hard-coded” or user provided. In the description of examples, file names and character strings are put in quotes, but the file names and character strings themselves do *not* contain the quotation marks. Single characters are surrounded by single quotes such as `'A'`. It would look better to use `'A'`, but when code is displayed in a text window `'A'` is usually shown rather than `'A'`.

### 1.4.2 Header Files

There are some public domain *Numerical Recipes* (Press et al., 1992) functions used in *Equating Recipes*. They are in `NRutilities.c` with an associated header file `NRutilities.h`.

The *Equating Recipes* open-source utility functions discussed in Chapter 2 are contained in a file named `ERutilities.c` which has an associated header file `ERutilities.h`. This header file also includes the declarations<sup>7</sup> of the structures listed on page 9, as well as the standard library headers `<stdio.h>`, `<stdlib.h>`, `<math.h>`, and `<string.h>`.

The other (i.e., non-utilities) *Equating Recipes* open-source functions are organized into files, each of which has an associated header file. These header files themselves are all included in a single header file named `ER.h`.

Therefore, to be safe, a user-defined `main()` function might include the following statements:

```
#include "ER.h"
#include "ERutilities.h"
#include "NRutilities.h"
```

For most `main()` functions of the type used in the examples in this monograph, only `ER.h` and `ERutilities.h` are actually needed.

### 1.4.3 Raw Scores

Virtually all of the discussions and examples in Kolen and Brennan (2004) implicitly or explicitly assume that raw scores are non-negative integers (usually number-correct scores). Unless otherwise noted, however, the C

---

<sup>7</sup>In *Equating Recipes* we usually follow the typical terminological convention that a “definition” involves memory allocation, while a “declaration” merely announces the properties of a variable (primarily its type). Note, however, that it is common practice to say that local variables for a function are declared at the top of file, even when doing so actually allocates storage as well.

functions in *Equating Recipes* are more general in that raw scores can be any set of real numbers (both positive and negative) provided that, in the ordered set of scores, adjacent scores differ by the same constant (not necessarily 1, although that is almost always the case).

#### 1.4.4 Input Files

The lines of all input files *including the last* should end with a carriage-return/linefeed (a newline character in the terminology of C). This is easily done by hitting the “Enter” key on the keyboard. In our experience with *Equating Recipes* and many other programs, a number of users save a file without typing “Enter” after the last entry in the last line. This may not always cause an execution error, but sometimes it does.

#### 1.4.5 Differences vis-a-vis other CASMA Computer Programs

Occasionally, there are minor differences between the results produced by the *Equating Recipes* functions and output from one or more of the equating programs on the CASMA website ([www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma)). Generally, the *Equating Recipes* results are more accurate, more easily defensible, or more consistent with the conventions in Kolen and Brennan (2004).<sup>8</sup>

#### 1.4.6 Differences vis-a-vis Kolen and Brennan (2004)

There is not a one-to-one correspondence between chapter numbers in *Equating Recipes* and chapter numbers in Kolen and Brennan (2004). This should not cause confusion as long as the reader pays attention to the context of comments in *Equating Recipes*.

#### 1.4.7 Arrays

Unless otherwise noted, all arrays (i.e., vectors) are zero-offset; i.e., they start at 0 rather than 1, which is traditional in some languages other than C, such as Fortran. Similarly, all dimensions of all multi-dimensional arrays (i.e., matrices) start at 0, unless otherwise noted. The principal exception occurs when certain *Numerical Recipes* functions are called.

For almost all vectors, space is allocated dynamically using the public domain utilities in *Numerical Recipes*, and almost always allocation occurs before any function is called that uses the vector. In such cases, the vector itself is not passed to the function; rather a pointer to the vector is passed. Similar conventions apply to matrices.

---

<sup>8</sup>The one slight exception to this statement is discussed in Section 16.4.2.

### 1.4.8 Release of Memory

Dynamic allocation of memory occurs frequently in *Equating Recipes*, especially in structures that are passed from one function to another. Dynamic allocation is performed using one of the *Numerical Recipes* public domain utilities in `NRutilities.c` or, occasionally, `allocate_matrix()` in `ERutilities.c`. (Sometimes `malloc()` is used directly.) Functions for releasing memory are also in `NRutilities.c` and `ERutilities.c`. However, often memory is not released in the *Numerical Recipes* functions themselves, because doing so might impede one or more of the multiple uses for these functions. The basic principle, of course, is that memory should be deallocated only after its final use, but the final use may occur at different points depending on the application. In most equating circumstances, deallocation does not matter because today's computers almost always have more than enough memory for a typical equating. However, when functions in *Equating Recipes* are used for simulations, it may be important to deallocate memory in the appropriate part of the code.

### 1.4.9 Verbal Shortcuts

To simplify descriptions of variables, particularly in the argument lists of functions, sometimes the role of pointers is verbally suppressed in this monograph and in the comments for the *Equating Recipes* functions. For example, if a variable in a function prototype is `FILE *fp`, it is often described as file `fp` rather than a pointer to file `fp`. Similarly, if `vec` is the name of an integer vector with space dynamically allocated as discussed previously, then `int *vec` in a function prototype might be described as the integer vector `vec[]`, rather than the technically more correct description: "a pointer to the first element of a vector named `vec`." Similarly, `double **matrix` might be described as a double matrix `matrix[][]`, rather than the technically more correct description: "a pointer to the set of pointers to the rows of a double matrix named `matrix`" (for those matrices whose space is dynamically allocated using *Numerical Recipes* conventions).

## 1.5 Appendix: Open Source License

The *Equating Recipes* code is distributed under the GNU Lesser General Public License, version 3 (LGPLv3), June 29, 2007, which grants certain permissions beyond those in the GNU General Public License, version 3 (GPLv3), June 29, 2007. The `ReadMe` file distributed with the *Equating Recipes* code contains the full text of the license. The following discussion provides a shorter, simplified version of prominent license conditions. This summary should not be interpreted as limiting, expanding, or in any way altering the license conditions in LGPLv3.

This license applies to the software’s source and object code and comes with any rights that the authors have in it (other than trademarks). You agree to the the terms of this license by copying, distributing, or making a derivative work of the software. You get the royalty free right to:

- use the software for any purpose;
- make derivative works of it (this is called a “Derived Work”); and
- copy and distribute it and any Derived Work.

If you distribute the software or a Derived Work, you must give back to the community by:

- prominently noting the date of any changes you make;
- leaving other people’s copyright notices, warranty disclaimers, and license terms in place;
- providing the source code in a form that is easy to get and modify;
- licensing it to everyone under the LGPLv3 license, without adding further restrictions to the rights provided; and
- conspicuously announcing that it is available under that license.

Furthermore,

- you get NO WARRANTIES—none of any kind; and
- if the software damages you in any way, you may only recover direct damages up to the amount you paid for it (that is zero if you did not pay anything). You may not recover any other damages, including those called “consequential damages.” (The state or country where you live may not allow you to limit your liability in this way, so this may not apply to you).

This license continues perpetually, except that your license rights end automatically if you do not abide by the “give back to the community” terms (your licensees get to keep their rights if they abide by the terms of this license).

Note that LGPLv3 permits use of the code in proprietary programs, whereas GPLv3 does not.





# 2

## Utilities

The code for the *Equating Recipes* utility functions is in `ERutilities.c`. The prototypes are in `ERutilities.h`, which also includes the declarations of almost all structures in *Equating Recipes*.

### 2.1 Descriptions of Utility Functions

In this section, the utility functions for *Equating Recipes* are categorized and described. For many functions, the input parameters and return values are described in more detail in the comments in the code for the functions.

#### 2.1.1 Raw Scores and Locations

With minor exceptions raw scores are *not* limited to number-correct scores. Rather, raw scores can be any real numbers as long as there is a constant difference (called the increment, here) between adjacent scores in the ordered sequence of scores.<sup>1</sup> Usually *Equating Recipes* does not store the raw scores; it computes them whenever necessary. The functions for determining raw scores, the number of raw scores, and the location of each raw score in a vector are as follows:

---

<sup>1</sup>Sometimes, of course, the context requires that raw scores be consecutive non-negative integers (e.g., number-correct scores for a test consisting solely of dichotomously-scored items).

- `double score(int loc, double min, double inc)`

returns the raw score associated with location `loc` in a zero-offset vector, where the minimum raw score is `min`, and the increment between raw scores is `inc`.

- `int nscores(double max, double min, double inc)`

returns the number of raw scores, say `ns`, where the minimum raw score is `min`, the maximum raw score is `max`, and the increment between raw scores is `inc`. If the raw scores are associated with a zero-offset vector (which is almost always the case in *Equating Recipes*), then the index of the vector ranges from 0 to `ns-1`.

- `int loc(double x, double min, double inc)`

returns the location (in a vector) associated with a raw score of `x` when the minimum raw score is `min` and the increment between raw scores is `inc`.

### 2.1.2 Reading and Storing Data

An important and frequently used convention in *Equating Recipes* is that input data are stored as elements in structures. The two primary structures are `USTATS` and `BSTATS`, which are declared in `ERutilities.h`. The random groups design requires two `USTATS` structures, the single group design requires one `BSTATS` structure, and the common-item nonequivalent groups design requires two `BSTATS` structures. The primary functions described next read raw data or frequency distributions and store statistics and other information in `USTATS` or `BSTATS`. A first two functions merely compute certain statistics. (Note that *Equating Recipes* does not score examinee response vectors.)

- `int ReadRawGet_mind_maxd(char fname[], int scol,  
double *mind, double *maxd)`

gets the lowest score `mind` and the highest score `maxd` in whitespace-delimited column `scol` in a file named `fname`. The function returns the number of examinees.

- `int ReadRawGet_moments(char fname[], int scol,  
double *moments, double *mind,  
double *maxd)`

computes the first four moments of raw scores in whitespace-delimited column `scol` in a file named `fname`. The function also determines the lowest score `mind` and highest score `maxd`, and returns the number of examinees.

- `void ReadRawGet_USTATS(char fname[], int scol, double min, double max, double inc, char id, struct USTATS *s)`

reads raw data from column `scol` in a whitespace-delimited file named `fname` and gets or assigns all elements of `USTATS` structure `s`. Input to the function includes the minimum raw score (`min`), the maximum raw score (`max`), the raw score increment (`inc`), and a single-character identifier (`id`) for the variable in column `scol`. If anything other than 'X' (for new form) or 'Y' (for old form) is used for `id`, the user should exercise great care in interpreting the output generated by the *Equating Recipes* print functions.

Note that `min` can be lower than the lowest raw score in column `scol` (`mind`), and `max` can be higher than the highest score (`mind`). For example, with a test of  $K$  dichotomously-scored items with `inc` = 1, this feature might be used to force the raw-score frequency distribution to range from a raw score of 0 to a raw score of  $K$  even though the lowest actual raw score in the data is greater than 0.

When `inc` is a fraction, particularly a repeating decimal (e.g.,  $\bar{3}$ ), raw scores in file `fname`, `min`, `max`, and `inc` should be specified with as much precision as is necessary for the function `score()` to return a sufficiently accurate raw score. When `inc` is a repeating decimal, at least eight decimal digits are suggested.

- `void ReadFdGet_USTATS(char fname[], int scol, int fcol, double min, double max, double inc, char id, struct USTATS *s)`

reads a raw-score frequency distribution from column `scol` (for scores) and column `fcol` (for frequencies) in a whitespace-delimited file named `fname`, and gets or assigns all elements of `USTATS` structure `s`. (See discussion of `ReadRawGet_USTATS()` for a description of other arguments.)

The frequency column must follow the raw score column. Raw scores with zero frequencies need not be in the file, and raw scores need not be ordered in any particular manner. See the previous description of `ReadRawGet_USTATS()` for additional details.

- `void ReadRawGet_BSTATS(char fname[], int rows, int cols, double rmin, double rmax, double rinc, double cmin, double cmax, double cinc, char rid, char cid, struct BSTATS *s)`

reads raw data for two variables in a whitespace-delimited file and gets or assigns all elements for a `BSTATS` bivariate structure `s`. The following is a description of the input variables:

- ▷ **fname** = name of file;
- ▷ **rows** = column (in file) for score for rows of bivariate frequency distribution matrix;
- ▷ **cols** = column (in file) for score for columns of bivariate frequency distribution matrix;
- ▷ **rmin** = minimum value for row scores;
- ▷ **rmax** = maximum value for row scores;
- ▷ **rinc** = increment for row scores;
- ▷ **cmin** = minimum value for column scores;
- ▷ **cmax** = maximum value for column scores;
- ▷ **cinc** = increment for column scores;
- ▷ **rid** = single-character id for rows; and
- ▷ **cid** = single-character id for columns.

If the data are based on the common-item nonequivalent groups design, then almost always '**rid**' should be '**X**' or '**Y**' (for full-length form), and '**cid**' should be '**V**' (for common items). If the data are based on the single-group design, then almost always '**rid**' should be '**X**' and '**cid**' should be '**Y**'. If these conventions are not followed, then the user should exercise great care in interpreting the output generated by the *Equating Recipes* print functions.

Almost always **rinc** = **cinc**. It is not necessary that **rows** < **cols**; this permits rows or columns to be associated with either variable. Minimum, maximum, and increment values should be specified with sufficient precision, as discussed on page 19. Similarly, minimum and maximum values can be lower and/or smaller, respectively than the minimum and maximum scores in the data, as discussed on page 19.

### 2.1.3 Basic Distributional Characteristics

Discussed next are a set of functions that are related to, or used to obtain, certain characteristics of raw score distributions. These include functions for obtaining moments, performing interpolation, and obtaining cumulative relative frequencies. Several of these functions use the variables **min**, **max**, and **inc**, which are described on page 19.

- **int MomentsFromFD(double min, double max, double inc, double \*scores, int \*fd, double \*moments)**

computes the first four moments of a frequency distribution, **fd[]**, with frequencies stored as integers. Scores are either (i) computed using **score()** or (ii) stored in **scores[]**. If **scores == NULL**, then (i) is used; otherwise (ii) is used.

- `void MomentsFromRFD(double min, double max, double inc,  
double *scores, double *rfd,  
double *moments)`

computes the first four moments of a relative frequency distribution, `rfd[]`. Scores are either (i) computed using `score()` or (ii) stored in `scores[]`. If `scores = NULL`, then (i) is used; otherwise (ii) is used.

- `double interpolate(double x, int ns, double *f)`

returns the interpolated value of `x` given the relative frequency distribution `f[]` assuming there are `ns` score categories in the zero-offset vector `f[]`.

- `void cum_rel_freqs(double min, double max, double inc,  
double *rfd, double *crfd)`

computes the cumulative relative frequency distribution (`crfd[]`) from the relative frequency distribution (`rfd[]`).

#### 2.1.4 *Equipercentile Equating*

Equipercentile equating, which requires obtaining percentile ranks and percentile points, is central to many equating procedures. Therefore, these functions are treated as utility functions in *Equating Recipes*.

- `double perc_rank(double min, double max, double inc,  
double *crfd, double x)`

returns the percentile rank for a score of `x` given a cumulative relative frequency distribution `crfd[]`. `x` can be any real number between `min` and `max`. In particular, `x` need not be an “achievable” score in the sense of a score that results from the function `score()`. (Recall that `min`, `max`, and `inc` are described on page 19).

- `double perc_point(int ns, double min, double inc,  
double *crfd, double pr)`

which, for a given percentile rank (`pr`) and cumulative relative frequency distribution (`crfd[]`), returns the percentile point, which is defined as the average of the “upper” and “lower” percentile points (see Kolen & Brennan, 2004, Equation 2.15 and 2.16, p. 45). This function is the inverse of the `perc_rank()` function.

- `void EquiEquate(int nsy, double miny, double incy,  
double *crfdy, int nsx,  
double *prdx, double *eraw)`

which, given percentile ranks for scores on  $X$  (`prdx[]`) and the cumulative relative frequency distribution for  $Y$  (`crfd[]`), computes equipercentile equivalents that transform raw scores on  $X$  to the scale of  $Y$  (see Kolen & Brennan, 2004, Equation 2.18, p. 46). The equivalents are stored in `eraw[]`. This function makes direct use of the functions `perc_rank()` and `perc_point()`.

The “tail” conventions employed in the code for `perc_point()` are such that the lowest possible percentile point is `min-inc/2` and the highest possible percentile point is `max+inc/2`. For a dichotomously-scored test of  $K$  items, this means that the possible range is  $[-.5, K + .5]$  (see Kolen & Brennan, p. 45). It follows that equated scores obtained using `EquiEquate()` are constrained to be in the same range.

### 2.1.5 Scale Scores

The equating process typically involves two major steps: (i) equating raw scores on Form  $X$  to the scale of raw scores for Form  $Y$ , and (ii) converting the equated raw scores for Form  $X$  to the scale scores for Form  $Y$ . The second step is entirely independent of the first and, therefore, the scale-score functions in *Equating Recipes* are independent of the equating method used.

- `void ReadSSConvTableForY(char nameyct[], double minp, double maxp, double inc, double **yct)`

reads the raw-to-scale score conversion table (`yct[][]`) for Form  $Y$  from a white-space-delimited file named `nameyct`. Raw scores must be in the first column, and scale scores must be in the second column. Almost always unrounded scale scores should be provided.

The remaining input variables are:

- ▷ `minp` = minimum raw score for Form  $y$  conversion, not simply the minimum score in a particular data set (should be set to 0 for number-correct scores);
- ▷ `maxp` = maximum raw score for Form  $Y$  conversion, not simply the maximum score in a particular data set (should be set to number of items for number-correct scores); and
- ▷ `inc` = raw score increment for Form  $Y$  conversion.

It is important to note that the number of records in `yct` should be equal to the number of score categories plus two—that is `nscores(maxp,minp,inc)+2`, where

- ▷ first record is for `minp-(inc/2)`,

- ▷ second record is for `min`,
- ▷ third record is for `min+inc`,
- ...
- ▷ 3rd last record is for `max-inc`,
- ▷ 2nd last record is for `max`, and
- ▷ last record is for `maxp+(inc/2)`.

In effect, the first record provides the lower-limit for scale scores, and the last record provides the upper limit for scale scores. These are the conventions followed by Kolen and Brennan (2004, p. 57). Users should type “Enter” after the scale score in the last record (see Section 1.4.4).

The scale scores in `yct[] []` are used by `Equated_ss()` to get equated scale scores, which means that the `yct[] []` scale scores should be specified with considerable precision.

The lines of all input files *including the last* should end with a carriage-return/linefeed (a newline character in the terminology of C). This is easily done by hitting the “Enter” key on the keyboard. In our experience with *Equating Recipes* and many other programs, a number of users save a file without typing “Enter” after the last entry in the last line. This may not always cause an execution error, but sometimes it does.

- `void Wrapper_ESS(struct PDATA *inall, struct ERAW_RESULTS *r, double minp, double maxp, double incp, char *nameyct, int round, int lprss, int hprss, struct ESS_RESULTS *s)`

is a wrapper function used for getting equated scale scores (both unrounded and rounded) and then storing them in the `ESS_RESULTS` structure `s`, along with scale-score moments (both rounded and unrounded). The input consists of:

- ▷ several elements from the `PDATA` structure `inall`;
- ▷ equated raw scores from the `ERAW_RESULTS` structure `r`;
- ▷ `minp` = minimum raw score for Form Y conversion;
- ▷ `maxp` = maximum raw score for Form Y conversion;
- ▷ `incp` = raw score increment for Form Y conversion;
- ▷ `nameyct` = name of file containing raw-to-scale score conversion for Y;

- ▷ **round**: if **round** = 0, then there is no rounding of scale scores; otherwise, if **round** = *i*, rounded scale scores are rounded to *i*-th digit before the decimal point (e.g., if **round** = 1, 123.45 rounds to 123, and if **round** = 2, 123.45 rounds to 120);
- ▷ **lprss** = lowest possible rounded scale score; and
- ▷ **hprss** = highest possible rounded scale score.

The principal function called by `Wrapper_ESS()` is `Equated_ss()`, which is discussed next.

- `void Equated_ss(double min, double max, double inc,  
double minp, double maxp, double incp,  
double *eraw, double **yct, int round,  
int lprss, int hprss,  
double *essu, double *essr)`

gets rounded and unrounded scale-scores (`essu[]` and `essr[]`, respectively) given equated raw scores (`eraw[]`) and the raw-to-scale-score conversion table for *Y* (`yct[][]`). It is assumed that the increment for *X* (`inc`) is the same as the increment for *Y* (`incp`). See the description of `Wrapper_ESS()` for definitions of the other input variables.

### 2.1.6 Alphanumeric Conversions

Sometimes characters need to be converted to integers, and sometimes characters need to be converted to a double format. Furthermore, for reading data, the default assumption in *Equating Recipes* is that data are whitespace delimited. Therefore, when data are in fixed-format, they need to be converted to whitespace-delimited data. Functions for these purposes are:

- `int atoiinteger(FILE *fp, char *str, int begin, int end,  
int *x)`

finds the string `str` in positions `begin` to `end` (inclusive) in file `fp` and converts the string to an integer variable `x`. (The function returns the string length or EOF if end-of-file or an error occurs.)

- `int atodouble(FILE *fp, char *str, int begin, int end,  
double *x)`

finds the string `str` in positions `begin` to `end` (inclusive) in file `fp` and converts the string to a double variable `x`. (The function returns the string length or EOF if end-of-file or an error occurs.)

- `void convertFtoW(char fname[], int nv, int fields[][3],  
char tname[])`



converts **nv** selected variables in a fixed format file named **fname** to **nv** tab-delimited variables in a file named **tname** that can be read directly using other utility functions. The location of the variables in the file named **fname** is provided by **fields[] [3]**, where there are as many rows (specified by the user) as there are fields to convert. The three columns of **fields[] [3]** are as follows:

- ▷ col 1 : 0 means write variable as integer; 1 means write variable as double;
- ▷ col 2: beginning column in file **fp** for variable; and
- ▷ col 3: ending column in file **fp** for variable.

An example is provided later in this chapter.

### 2.1.7 *Print Functions*

The utility print functions are listed below. In a sense, the word “print” is a misnomer; what actually happens is that results are written to a file specified by the user. As noted previously, these functions were created primarily to illustrate output for the purposes of this monograph; they are not necessarily optimal for particular research or operational equating purposes.

- **void Print\_USTATS(FILE \*fp, char tt[], struct USTATS \*s)**

writes results to file **fp** for the USTATS structure **s**. A user supplied title **tt[]** is provided at the beginning of the file.

- **void Print\_BSTATS(FILE \*fp, char tt[], struct BSTATS \*s, int pbfd)**

writes results to file **fp** for the BSTATS structure **s**. A user supplied title **tt[]** is provided at the beginning of the file. If **pbfd = 1** then the bivariate frequency distribution and bivariate relative frequency distribution are printed. If **pbfd = 0** then the bivariate distributions are not printed.

- **void Print\_ESS(FILE \*fp, char tt[], struct PDATA \*inall, struct ESS\_RESULTS \*s)**

writes the raw-to-scale score conversion table for Form Y (**yct[] []**) to file **fp**, as well as equated scale scores (unrounded and rounded) for Form X. The input variables are:

- ▷ **fp** = file pointer for output;
- ▷ **tt[]** = user supplied text identifier that is printed at the top of the file;

- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ variables in the `ESS_RESULTS` structure `s` including
  - \* `essu[]` = unrounded equated scale scores,
  - \* `essr[]` = rounded equated scale scores,
  - \* `mtsu[]` = moments for equated unrounded scale scores, and
  - \* `mtsr[]` = moments for equated rounded scale scores.

The raw-to-scale score conversion table for Form Y is always printed first, with each scale score followed by `|`, which make the table look like it terminates with a horizontal line. Scale scores associated with the equated Form X raw scores are *not* printed with this terminating line, which makes them readily distinguishable from the base Form Y scale scores.

- `void Print_vector(FILE *fp, char label[], double *vector, int nrow, char rowhead[], char colhead[])`

writes a double vector named `vector` to file `fp`. The additional input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `label[]` = user supplied text identifier that is printed at the top of the file;
- ▷ `nrow` = number of rows (i.e., elements) in the vector;
- ▷ `rowhead` = heading for rows (often "Raw Score"); and
- ▷ `colhead` = heading for column (contains the vector elements).

Data are printed in columns of width 12. So, to make `rowhead` and `colhead` align with printed data, the headings should be 12 characters long. For 40 rows, an example is:

```
Print_vector(outf, "Dummy output", vector, 40,
             "    Raw Score",
             "    Column")
```

- `void Print_matrix(FILE *fp, char label[], double **matrix, int nrow, int ncol, char rowhead[], char colheads[])`

writes a double matrix named `matrix` to file `fp`. The additional input variables are:

- ▷ `fp` = file pointer for output;

- ▷ `label[]` = user supplied text identifier that is printed at the top of the file;
- ▷ `nrows` = number of rows in matrix;
- ▷ `ncols` = number of columns in matrix;
- ▷ `rowhead` = heading for rows (often "Raw Score"); and
- ▷ `colheads` = headings for each of columns.

Data are printed in columns of width 18. So, to make `rowhead` and `colheads` align with printed data, the headings should be 18 characters long. An example is for 40 rows and two columns is:

```
Print_matrix(outf, "Dummy output", matrix, 40, 2,
             "          Raw Score",
             "          First Col          Second Col")
```

- `void Print_file(char FileName[], FILE *fp)`

prints the contents of file named `FileName` to the file pointed to by `fp`.

### 2.1.8 Miscellaneous

There are various miscellaneous functions including:

- `int skipcols(FILE *fp, int k)`

skips `k` characters from file `fp`; returns `k` or `EOF` if end-of-file is encountered or an error occurs.

- `void flushline(FILE *fp)`

deletes all characters in file `fp` from current file position to newline or return character.

- `void runerror(char error_text[])`

is the error handler usually used in *Equating Recipes*.

## 2.2 Example

Table 2.1 provides a `main()` function illustrating reading data and printing results for the two structures `USTATS` and `BSTATS`.

TABLE 2.1. Main() Code to Illustrate Utilities

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x;
    struct BSTATS xv;
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    FILE *outf;

    outf = fopen("Chap 2 out","w");

    /*****

    /* Random Groups Design: Kolen and Brennan (2004)
       Chapter 2 example (see pp. 50-52) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    Print_USTATS(outf,"ACT Math X",&x);

    /* Common-items Nonequivalent Groups Design:
       Kolen and Brennan (2004) Chapter 4 example (see page 123)*/

    convertFtoW("mondatx.dat",2,fieldsACT,"mondatx-temp");
    ReadRawGet_BSTATS("mondatx-temp",1,2,0,36,1,0,12,1,'X','V",&xv);
    Print_BSTATS(outf,"xv",&xv,1);

    fclose(outf);
    return 0;
}
*****/

```

### 2.2.1 USTATS

The `USTATS` structure is named `x`. The output file is named `"Chap 2 out"` (without the quotation marks) with the file pointer being `outf`. The function `ReadFdGet_USTATS()` is called, followed by `Print_USTATS()` that writes output to the file named `"Chap 2 out"`.

The arguments in `ReadFdGet_USTATS()` (see page 19) specify that the data are read from a file named `"actmathfreq.dat"` (without the quotes) in which the first column contains the raw scores, the second column contains the frequencies, the minimum raw score is 0, the maximum raw score is 40, and the increment is 1. The letter `X` represents the variable, and results are stored in the `USTATS` structure named `x`. Note that the address of the structure is passed, not the structure itself.

The arguments in `Print_USTATS()` (see page 25) specify that members of the `USTATS` structure named `x` are written to the file pointed to by `outf`, and the first line in the file will be the title `"ACT Math X"` (without the quotation marks). Again, note that the address of the structure is passed, not the structure itself. A listing of the output is provided in Table 2.2 (see Kolen & Brennan, 2004, pp. 50–52).

TABLE 2.2. Output for struct USTATS x (ACT Math)

```

/*****
ACT Math X

Input filename:  actmathfreq.dat

Variable identifier: X

Number of persons =   4329

Min score in data =      1.00000
Max score in data =     40.00000
      Mean =      19.85239
      S.D. =      8.21164
      Skew =      0.37527
      Kurt =      2.30244

Min score for fd[] =      0.00000
Max score for fd[] =     40.00000
Increment between scores =      1.00000
Number of scores =      41

      Score  Freq  CFreq    Rel Freq    Cum RFreq    Perc Rank
0.00000      0      0    0.00000    0.00000    0.00000
1.00000      1      1    0.00023    0.00023    0.01155
2.00000      1      2    0.00023    0.00046    0.03465
3.00000      3      5    0.00069    0.00116    0.08085
4.00000      9     14    0.00208    0.00323    0.21945
5.00000     18     32    0.00416    0.00739    0.53130
6.00000     59     91    0.01363    0.02102    1.42065
7.00000     67    158    0.01548    0.03650    2.87595
8.00000     91    249    0.02102    0.05752    4.70085
9.00000    144    393    0.03326    0.09078    7.41511
10.00000   149    542    0.03442    0.12520   10.79926
11.00000   192    734    0.04435    0.16955   14.73781
12.00000   192    926    0.04435    0.21391   19.17302
13.00000   192   1118    0.04435    0.25826   23.60822
14.00000   201   1319    0.04643    0.30469   28.14738
15.00000   204   1523    0.04712    0.35181   32.82513
16.00000   217   1740    0.05013    0.40194   37.68769
17.00000   181   1921    0.04181    0.44375   42.28459
18.00000   184   2105    0.04250    0.48626   46.50035
19.00000   170   2275    0.03927    0.52553   50.58905
20.00000   201   2476    0.04643    0.57196   54.87410
21.00000   147   2623    0.03396    0.60591   58.89351
22.00000   163   2786    0.03765    0.64357   62.47401
23.00000   147   2933    0.03396    0.67752   66.05452
24.00000   140   3073    0.03234    0.70986   69.36937
25.00000   147   3220    0.03396    0.74382   72.68422
26.00000   126   3346    0.02911    0.77293   75.83738
27.00000   113   3459    0.02610    0.79903   78.59783
28.00000   100   3559    0.02310    0.82213   81.05798
29.00000   106   3665    0.02449    0.84662   83.43728
30.00000   107   3772    0.02472    0.87133   85.89744
31.00000    91   3863    0.02102    0.89235   88.18434
32.00000    83   3946    0.01917    0.91153   90.19404
33.00000    73   4019    0.01686    0.92839   91.99584
34.00000    72   4091    0.01663    0.94502   93.67059
35.00000    75   4166    0.01733    0.96235   95.36845
36.00000    50   4216    0.01155    0.97390   96.81220
37.00000    37   4253    0.00855    0.98244   97.81705
38.00000    38   4291    0.00878    0.99122   98.68330
39.00000    23   4314    0.00531    0.99653   99.38785
40.00000    15   4329    0.00347    1.00000   99.82675
*****/

```

### 2.2.2 BSTATS

The **BSTATS** structure is named **xv**. As before, the output file is named "out chap 2" with the file pointer being **outf**.

The function **convertFtoW()** (see page 24) converts 2 variables in the fixed-format file named "mondatx.dat" to tab-delimited variables in a new file named "mondatx-temp". The conversion specifications are provided by the two-dimensional integer array **fieldsACT[2][3]**, which says that the first variable to be read in "mondatx.dat" is an integer in columns 37–38, and the second variable is an integer in columns 39–40. These two variables will be in tab-delimited locations 1 and 2, respectively, in "mondatx-temp".

The function **ReadRawGet\_BSTATS()** (see page 19) reads the first two tab-delimited variables in "mondatx-temp", with identifiers **X** and **V**, respectively. The minimum raw score for **X** is 0, the maximum raw score is 36, and the increment is 1. The minimum raw score **V** is 0, the maximum raw score is 12, and the increment is 1. Results are stored in the **BSTATS** structure named **xv**. Again, note that the address of the structure is passed, not the structure itself.

The arguments in **Print\_BSTATS()** (see page 25) specify that members of the **BSTATS** structure named **xv** are written to the file pointed to by **outf**, and the first line in the file will be "**xv**" (without the quotation marks). The last parameter in the argument list is 1 which indicates that the bivariate frequency distribution should be printed. Again, note that the address of the structure is passed, not the structure itself.

The output is provided in Tables 2.3–2.6 (see Kolen & Brennan, 2004, p. 123), which includes results for the marginal distribution of *X*, the marginal distribution of *V*, the bivariate frequency distribution of *X* and *V*, and the bivariate relative frequency distribution of *X* and *V*.

TABLE 2.3. Output for  $X$  in struct BSTATS xv using mondatx

```

/*****
xv

Input filename: mondatx-temp

Number of Persons = 1655

Variable 1 identifier (rows of bfd[]): X

Min score in data = 2.00000
Max score in data = 36.00000
      Mean = 15.82054
      S.D. = 6.52783
      Skew = 0.57991
      Kurt = 2.72166

Min score for fd1[] = 0.00000
Max score for fd1[] = 36.00000

      Score  Freq  CFreq  Rel Freq  Cum RFreq  Perc Rank
0.00000    0    0    0.00000    0.00000    0.00000
1.00000    0    0    0.00000    0.00000    0.00000
2.00000    1    1    0.00060    0.00060    0.03021
3.00000    5    6    0.00302    0.00363    0.21148
4.00000    9   15    0.00544    0.00906    0.63444
5.00000   13   28    0.00785    0.01692    1.29909
6.00000   36   64    0.02175    0.03867    2.77946
7.00000   51  115    0.03082    0.06949    5.40785
8.00000   77  192    0.04653    0.11601    9.27492
9.00000   86  278    0.05196    0.16798   14.19940
10.00000  92  370    0.05559    0.22356   19.57704
11.00000 113  483    0.06828    0.29184   25.77039
12.00000 128  611    0.07734    0.36918   33.05136
13.00000  90  701    0.05438    0.42356   39.63746
14.00000 113  814    0.06828    0.49184   45.77039
15.00000  80  894    0.04834    0.54018   51.60121
16.00000  91  985    0.05498    0.59517   56.76737
17.00000  87 1072    0.05257    0.64773   62.14502
18.00000  73 1145    0.04411    0.69184   66.97885
19.00000  50 1195    0.03021    0.72205   70.69486
20.00000  73 1268    0.04411    0.76616   74.41088
21.00000  54 1322    0.03263    0.79879   78.24773
22.00000  54 1376    0.03263    0.83142   81.51057
23.00000  41 1417    0.02477    0.85619   84.38066
24.00000  35 1452    0.02115    0.87734   86.67674
25.00000  42 1494    0.02538    0.90272   89.00302
26.00000  31 1525    0.01873    0.92145   91.20846
27.00000  31 1556    0.01873    0.94018   93.08157
28.00000  23 1579    0.01390    0.95408   94.71299
29.00000  22 1601    0.01329    0.96737   96.07251
30.00000  17 1618    0.01027    0.97764   97.25076
31.00000   8 1626    0.00483    0.98248   98.00604
32.00000  14 1640    0.00846    0.99094   98.67069
33.00000   7 1647    0.00423    0.99517   99.30514
34.00000   6 1653    0.00363    0.99879   99.69789
35.00000   1 1654    0.00060    0.99940   99.90937
36.00000   1 1655    0.00060    1.00000   99.96979
*****/

```



TABLE 2.4. Output for  $V$  in struct BSTATS xv using mondatx

```

/*****
Variable 2 identifier (columns of bfd[] []): V

Min score in data =      0.00000
Max score in data =     12.00000
      Mean =          5.10634
      S.D. =          2.37602
      Skew =          0.41168
      Kurt =          2.76829

Min score for fd2[] =      0.00000
Max score for fd2[] =     12.00000

      Score  Freq
      Score  Freq  CFreq    Rel Freq    Cum RFreq    Perc Rank
0.00000     14     14    0.00846    0.00846    0.42296
1.00000     54     68    0.03263    0.04109    2.47734
2.00000    142    210    0.08580    0.12689    8.39879
3.00000    249    459    0.15045    0.27734   20.21148
4.00000    274    733    0.16556    0.44290   36.01208
5.00000    247    980    0.14924    0.59215   51.75227
6.00000    232   1212    0.14018    0.73233   66.22356
7.00000    173   1385    0.10453    0.83686   78.45921
8.00000    118   1503    0.07130    0.90816   87.25076
9.00000     75   1578    0.04532    0.95347   93.08157
10.00000    42   1620    0.02538    0.97885   96.61631
11.00000    27   1647    0.01631    0.99517   98.70091
12.00000     8   1655    0.00483    1.00000   99.75831

Covariance(1,2) =      13.40881
Correlation(1,2) =       0.86451
*****/

```

TABLE 2.5. Output for Bivariate Frequency Distribution for struct BSTATS xv Using mondatx  
 /\*\*\*\*\*Bivariate Frequency Distribution bfd[rows=var1=x][cols=var2=v] \*\*\*\*\*/

Loc	Score	0	1	2	3	4	5	6	7	8	9	10	11	12	fd1[]
0	0.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2.00	1	0	0	0	0	0	0	0	0	0	0	0	0	1
3	3.00	2	2	1	0	0	0	0	0	0	0	0	0	0	5
4	4.00	0	4	3	2	0	0	0	0	0	0	0	0	0	9
5	5.00	2	4	5	2	0	0	0	0	0	0	0	0	0	13
6	6.00	0	7	13	15	1	0	0	0	0	0	0	0	0	36
7	7.00	6	9	19	11	6	0	0	0	0	0	0	0	0	51
8	8.00	0	7	19	30	16	3	2	0	0	0	0	0	0	77
9	9.00	3	12	20	29	18	2	2	0	0	0	0	0	0	86
10	10.00	0	5	16	30	30	9	2	0	0	0	0	0	0	92
11	11.00	0	2	18	29	33	20	9	2	0	0	0	0	0	113
12	12.00	0	0	16	29	36	29	16	1	1	0	0	0	0	128
13	13.00	0	2	4	21	28	24	11	0	0	0	0	0	0	90
14	14.00	0	0	26	31	27	19	19	5	0	1	0	0	0	113
15	15.00	0	0	1	11	20	23	15	8	2	0	0	0	0	80
16	16.00	0	0	2	7	16	25	30	10	1	0	0	0	0	91
17	17.00	0	0	1	3	16	27	24	12	3	0	1	0	0	87
18	18.00	0	0	0	4	10	16	26	11	6	0	0	0	0	73
19	19.00	0	0	0	0	5	12	17	9	6	1	0	0	0	50
20	20.00	0	0	0	0	0	16	16	24	10	2	0	0	0	73
21	21.00	0	0	0	0	3	7	13	20	8	2	1	0	0	54
22	22.00	0	0	0	0	0	3	16	20	11	3	1	0	0	54
23	23.00	0	0	0	0	0	2	7	17	7	6	2	0	0	41
24	24.00	0	0	0	0	0	0	2	12	13	7	1	0	0	35
25	25.00	0	0	0	0	0	2	2	11	16	9	2	0	0	42
26	26.00	0	0	0	0	0	0	0	5	16	9	1	0	0	31
27	27.00	0	0	0	0	0	0	3	1	7	14	3	3	0	31
28	28.00	0	0	0	0	0	0	0	2	6	12	3	0	0	23
29	29.00	0	0	0	0	0	0	0	3	4	7	4	4	0	22
30	30.00	0	0	0	0	0	0	0	0	1	1	12	2	1	17
31	31.00	0	0	0	0	0	0	0	0	0	1	4	3	0	8
32	32.00	0	0	0	0	0	0	0	0	0	0	5	8	1	14
33	33.00	0	0	0	0	0	0	0	0	0	0	2	3	2	7
34	34.00	0	0	0	0	0	0	0	0	0	0	0	3	3	6
35	35.00	0	0	0	0	0	0	0	0	0	0	0	1	0	1
36	36.00	0	0	0	0	0	0	0	0	0	0	0	0	1	1
fd2[]															
		14	54	142	249	274	247	232	173	118	75	42	27	8	1655

\*\*\*\*\*/

TABLE 2.6. Output for Bivariate Relative Frequency Distribution for struct BSTATS xv Using mdatx

```

/*****
/***** Biv Props bp12[rows=var1=x][cols=var2=y] *****/
/*****

```

Loc	Score	0	1	2	3	4	5	6	7	8	9	10	11	12	marg
0	0.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
1	1.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2	2.0	0.0060	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0060
3	3.0	0.0121	0.0060	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00302
4	4.0	0.0000	0.00242	0.00181	0.00121	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00544
5	5.0	0.00121	0.00242	0.00302	0.00121	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00785
6	6.0	0.0000	0.00423	0.00785	0.00906	0.00060	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.02175
7	7.0	0.00363	0.00544	0.01148	0.00665	0.00363	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.03082
8	8.0	0.0000	0.00423	0.01148	0.01813	0.00967	0.00181	0.00121	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.04653
9	9.0	0.00181	0.00725	0.01208	0.01752	0.01088	0.00121	0.00121	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.05196
10	10.0	0.0000	0.00302	0.00967	0.01813	0.01813	0.00544	0.00121	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.05559
11	11.0	0.0000	0.00121	0.01088	0.01752	0.01994	0.01208	0.00544	0.00121	0.0000	0.0000	0.0000	0.0000	0.0000	0.06828
12	12.0	0.0000	0.0000	0.00967	0.01752	0.02175	0.01752	0.00967	0.00060	0.00060	0.0000	0.0000	0.0000	0.0000	0.07734
13	13.0	0.0000	0.00121	0.00242	0.01269	0.01692	0.01450	0.00665	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.05438
14	14.0	0.0000	0.0000	0.00242	0.01571	0.01873	0.01631	0.01148	0.00302	0.0000	0.00060	0.0000	0.0000	0.0000	0.06828
15	15.0	0.0000	0.0000	0.00060	0.00665	0.01208	0.01390	0.00906	0.00483	0.00121	0.0000	0.0000	0.0000	0.0000	0.04834
16	16.0	0.0000	0.0000	0.00121	0.00423	0.00967	0.01511	0.01813	0.00604	0.00060	0.0000	0.0000	0.0000	0.0000	0.05498
17	17.0	0.0000	0.0000	0.00060	0.00181	0.00967	0.01631	0.01450	0.00725	0.00181	0.0000	0.00060	0.0000	0.0000	0.05257
18	18.0	0.0000	0.0000	0.0000	0.00242	0.00604	0.00967	0.01571	0.00665	0.00363	0.0000	0.0000	0.0000	0.0000	0.04411
19	19.0	0.0000	0.0000	0.0000	0.0000	0.00302	0.00725	0.01027	0.00544	0.00363	0.00060	0.0000	0.0000	0.0000	0.03021
20	20.0	0.0000	0.0000	0.0000	0.0000	0.00302	0.00967	0.00967	0.01450	0.00604	0.00121	0.0000	0.0000	0.0000	0.04411
21	21.0	0.0000	0.0000	0.0000	0.0000	0.00181	0.00423	0.00785	0.01208	0.00483	0.00121	0.00060	0.0000	0.0000	0.03263
22	22.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.00181	0.00967	0.01208	0.00665	0.00181	0.00060	0.0000	0.0000	0.03263
23	23.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.00121	0.00423	0.01027	0.00423	0.00363	0.00121	0.0000	0.0000	0.02477
24	24.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00121	0.00725	0.00785	0.00423	0.00060	0.0000	0.0000	0.02115
25	25.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.00121	0.00121	0.00665	0.00967	0.00544	0.00121	0.0000	0.0000	0.02538
26	26.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00302	0.00967	0.00544	0.00060	0.0000	0.0000	0.01873
27	27.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00181	0.00060	0.00423	0.00846	0.00181	0.0000	0.0000	0.01873
28	28.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00121	0.00363	0.00725	0.00181	0.0000	0.0000	0.01390
29	29.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00181	0.00242	0.00423	0.00242	0.0000	0.0000	0.01329
30	30.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00060	0.00060	0.00725	0.00121	0.00060	0.01027
31	31.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00060	0.00181	0.0000	0.00483
32	32.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00302	0.00483	0.00846
33	33.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00121	0.00181	0.00121	0.00423
34	34.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00181	0.00181	0.00363
35	35.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00060	0.0000	0.00060
36	36.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.00060	0.00060

```

/*****
/***** rfd2[] | 0.00846 0.03263 0.08580 0.15045 0.16556 0.14924 0.14018 0.10453 0.07130 0.04532 0.02538 0.01631 0.00483 | 1.00000
/*****

```



# 3

## Linear Equating with Random-Groups and Single-Group Designs

Linear equating (including mean equating) for the random-groups design is discussed by Kolen and Brennan (2004, sections 2.1–2.4). The same equations provided there apply as well for the single-group design. For both designs, the equations use only the first two moments of the marginal distributions for Forms X and Y.

For mean equating, the equation that puts raw scores for the new Form X on the scale of raw scores for the old Form Y is:

$$m_Y(x) = x - \mu(X) + \mu(Y). \quad (3.1)$$

More generally, for linear equating:

$$l_Y(x) = \left[ \mu(Y) - \frac{\sigma(Y)}{\sigma(X)} \mu(X) \right] + \frac{\sigma(Y)}{\sigma(X)} x. \quad (3.2)$$

Obviously Equation 3.1 is identical to Equation 3.2 when the slope is  $\sigma(Y)/\sigma(X) = 1$ .

### 3.1 Functions

The functions to perform linear equating with the random-groups and single-group design are in `RGandSG_NoSmooth.c`, with the function prototypes in `RGandSG_NoSmooth.h`. The functions for the two types of designs are very similar. Provided next are detailed descriptions for the random-groups design functions. Then, a brief description of the single-group design functions is provided.

### 3.1.1 Wrapper\_RN()

`Wrapper_RN()` calls functions that perform mean, linear, or equipercentile equating (the last of which is discussed in Chapter 5) for the random groups design with no smoothing. The function prototype for `Wrapper_RN()` is:

```
void Wrapper_RN(char design, char method, char smoothing,
                struct USTATS *x, struct USTATS *y, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' designating the random groups design;
- ▷ `method` = 'M' or 'L' for mean or linear, respectively (equipercentile options are discussed later);
- ▷ `smoothing` = 'N' (none);
- ▷ `x` = a `USTATS` structure for the new form;
- ▷ `y` = a `USTATS` structure for the old form; and
- ▷ `rep` = replication number (must be set to 0 for actual equating; used to count replications when bootstrap standard errors are estimated).

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_RN()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

`Wrapper_RN()` populates the elements of the `PDATA` structure `inall`, dynamically allocates space for the `eraw[][]` and `mts[][]` matrices in the `ERAW_RESULTS` structure `r`, calls a function that does the actual equating (either `RGandSG_LinEq()` or `EquiEquate()`) and stores results in `r`, and calls the function `MomentsFromFD()` that gets the equated raw-score moments and stores them in `r`.

### 3.1.2 Print\_RN()

The function prototype for `Print_RN()` is:

```
void Print_RN(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the linear equating results for the random-groups design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the PDATA structure `inall` (recall that PDATA is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the ERAW\_RESULTS structure `r`.

### 3.1.3 Single-group Design Functions

The wrapper and print function prototypes for the single-groups design are:

```
void Wrapper_SN(char design, char method, char smoothing,
                struct BSTATS *xy, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

and

```
void Print_SN(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

There are only two differences between these functions and the random-groups functions. First, the single-group function names use `SN` rather than `RN`. Second, `Wrapper_SN` requires one `BSTATS` structure rather than two `USTATS` structures.

Equating with the single-group design uses only the marginals of the bivariate distribution. It follows that the single-group and random-groups designs employ exactly the same formulas for obtaining equated scores.<sup>1</sup>

## 3.2 Random Groups Example

Table 3.1 is a `main()` function that illustrates linear equating with the random-groups design based on an example discussed by Kolen and Brennan (2004, pp. 50–53, 57, 60). This example extends the `USTATS` example in Section 2.2.1. The example here has three parts:

- read the raw data for Form X and Form Y and store statistics in two `USTATS` structures, `x` and `y`, as discussed in Section 2.2.1;
- compute and print the equated raw scores using `Wrapper_RN()` and `Print_RN()`, respectively; and

---

<sup>1</sup>Note, however, that the standard errors of equated raw scores will be different for the two designs.

TABLE 3.1. Main() Code to Illustrate Linear Equating with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct PDATA pdRLN;
    struct ERAW_RESULTS rRLN;
    struct ESS_RESULTS sRLN;

    FILE *outf;

    outf = fopen("Chap 3 out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004) Chapter 2 example
       (see pp. 50-53, 57, 60) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y",&y);

    Wrapper_RN('R','L','N",&x,&y,0,&pdRLN,&rRLN);
    Print_RN(outf,"ACT Math---Linear",&pdRLN,&rRLN);

    Wrapper_ESS(&pdRLN,&rRLN,0,40,1,"yctmath.TXT",1,1,36,&sRLN);
    Print_ESS(outf,"ACT Math---Linear",&pdRLN,&sRLN);

    fclose(outf);
    return 0;
}
*****/

```

- compute and print the unrounded and rounded scale scores using `Wrapper_ESS()` and `Print_ESS()`, respectively, as discussed in Chapter 2 (see pages 23 and 25).

Note that the `PDATA` structure `pdRLN`, the `ERAW_RESULTS` structure `rRLN`, and the `ESS_RESULTS` structure `sRLN` are declared at the top of the `main()` function. (As noted previously, the definitions of these structures are in `ERutilities.h`.) Also, note that it is the addresses of all structures that are passed (designated by preceding each structure name with `&`), not the structures themselves.



### 3.2.1 *Equated Raw Scores*

The equated raw-score output is in Table 3.2. The initial part of the output provides the slope and intercept of the linear equation. Then, the raw to equated raw-score conversion table is provided along with moments that are based on using the new-form frequencies.

### 3.2.2 *Equated Scale Scores*

The equated scale-score output is split into three parts:

- the raw-to-scale score conversion table for Form Y is provided in Table 3.3 (note the terminating horizontal line—see page 26);
- the unrounded equated scale scores for Form X are provided in Table 3.4; and
- the rounded equated scale scores for Form X are provided in Table 3.5.

The equated scale-score moments are based on using the new-form frequencies.

TABLE 3.2. Linear Equating Raw-Score Results for the Random Groups Design

```

/*****
ACT Math--Linear

Linear Equating with Random Groups Design

Input file for x: actmathfreq.dat
Input file for y: actmathfreq.dat

inter b      -2.63186
slope a      1.08862
-----

Equated Raw Scores

Raw Score (x)      Method 0:
                    y-equiv

0.00000      -2.63186
1.00000      -1.54325
2.00000      -0.45463
3.00000       0.63398
4.00000       1.72260
5.00000       2.81122
6.00000       3.89983
7.00000       4.98845
8.00000       6.07706
9.00000       7.16568
10.00000      8.25430
11.00000      9.34291
12.00000     10.43153
13.00000     11.52015
14.00000     12.60876
15.00000     13.69738
16.00000     14.78599
17.00000     15.87461
18.00000     16.96323
19.00000     18.05184
20.00000     19.14046
21.00000     20.22907
22.00000     21.31769
23.00000     22.40631
24.00000     23.49492
25.00000     24.58354
26.00000     25.67216
27.00000     26.76077
28.00000     27.84939
29.00000     28.93800
30.00000     30.02662
31.00000     31.11524
32.00000     32.20385
33.00000     33.29247
34.00000     34.38108
35.00000     35.46970
36.00000     36.55832
37.00000     37.64693
38.00000     38.73555
39.00000     39.82417
40.00000     40.91278

Mean      18.97977
S.D.      8.93932
Skew      0.37527
Kurt      2.30244
*****/

```

TABLE 3.3. Scale Score Conversion Table for Old Form Y

```

/*****
ACT Math--Linear

Name of file containing yct[]: yctmath.TXT

Minimum raw score in base form conversion, Y =      0.00000
Maximum raw score in base form conversion, Y =      40.00000
    Increment for base form conversion, Y =      1.00000

Lowest possible rounded scale score =      1
Highest possible rounded scale score =      36

CONVERSION TABLE FOR Y (BASE FORM)

      Raw      Y |
      Score    Scale |
      Score    Score |
-0.50000    0.50000 |
 0.00000    0.50000 |
 1.00000    0.50000 |
 2.00000    0.50000 |
 3.00000    0.50000 |
 4.00000    0.50000 |
 5.00000    0.69000 |
 6.00000    1.65620 |
 7.00000    3.10820 |
 8.00000    4.69710 |
 9.00000    6.12070 |
10.00000    7.47320 |
11.00000    8.90070 |
12.00000   10.33920 |
13.00000   11.63880 |
14.00000   12.82540 |
15.00000   14.01570 |
16.00000   15.21270 |
17.00000   16.35280 |
18.00000   17.38240 |
19.00000   18.34030 |
20.00000   19.28440 |
21.00000   20.18390 |
22.00000   20.99470 |
23.00000   21.70000 |
24.00000   22.32200 |
25.00000   22.91780 |
26.00000   23.51830 |
27.00000   24.13140 |
28.00000   24.75250 |
29.00000   25.29150 |
30.00000   25.72870 |
31.00000   26.15340 |
32.00000   26.64800 |
33.00000   27.23850 |
34.00000   27.90810 |
35.00000   28.69250 |
36.00000   29.74860 |
37.00000   31.20100 |
38.00000   32.69140 |
39.00000   34.19520 |
40.00000   35.46150 |
40.50000   36.50000 |
*****/

```

TABLE 3.4. Equated Unrounded Scale Scores for  
Linear Equating with the Random Groups Design

```

/*****
ACT Math--Linear

```

```

Name of file containing yct[] []: yctmath.TXT

```

```

Equated Scale Scores (unrounded)

```

```

Raw Score (x)      Method 0:
                   y-equiv

```

```

0.00000 0.50000
1.00000 0.50000
2.00000 0.50000
3.00000 0.50000
4.00000 0.50000
5.00000 0.50000
6.00000 0.50000
7.00000 0.68781
8.00000 1.76810
9.00000 3.37145
10.00000 5.05912
11.00000 6.58449
12.00000 8.08921
13.00000 9.64893
14.00000 11.13035
15.00000 12.46631
16.00000 13.76097
17.00000 15.06261
18.00000 16.31087
19.00000 17.43206
20.00000 18.47291
21.00000 19.49045
22.00000 20.44148
23.00000 21.28127
24.00000 22.00784
25.00000 22.66967
26.00000 23.32143
27.00000 23.98473
28.00000 24.65895
29.00000 25.25808
30.00000 25.74001
31.00000 26.21040
32.00000 26.76837
33.00000 27.43434
34.00000 28.20702
35.00000 29.18855
36.00000 30.55950
37.00000 32.16519
38.00000 33.79752
39.00000 35.23884
40.00000 36.50000

```

```

Mean      16.58753
S.D.      8.36881
Skew      -0.11681
Kurt      2.19791

```

```

/*****

```

TABLE 3.5. Equated Rounded Scale Scores for Linear Equating with the Random Groups Design

```

/*****
ACT Math---Linear

Name of file containing yct[] []: yctmath.TXT

Equated Scale Scores (rounded)

Raw Score (x)      Method 0:
                    y-equiv

0.00000            1
1.00000            1
2.00000            1
3.00000            1
4.00000            1
5.00000            1
6.00000            1
7.00000            1
8.00000            2
9.00000            3
10.00000           5
11.00000           7
12.00000           8
13.00000          10
14.00000          11
15.00000          12
16.00000          14
17.00000          15
18.00000          16
19.00000          17
20.00000          18
21.00000          19
22.00000          20
23.00000          21
24.00000          22
25.00000          23
26.00000          23
27.00000          24
28.00000          25
29.00000          25
30.00000          26
31.00000          26
32.00000          27
33.00000          27
34.00000          28
35.00000          29
36.00000          31
37.00000          32
38.00000          34
39.00000          35
40.00000          36

Mean              16.50820
S.D.              8.30653
Skew              -0.07758
Kurt              2.19489
*****/

```



# 4

## Linear Equating with the Common-item Nonequivalent Groups Design

Kolen and Brennan (2004, chap. 4) provides an extensive discussion of, and derivation of results for, linear equating with the common-item nonequivalent groups design. They do not consider chained linear equating, however, which is treated by Brennan (2006) and is incorporated into *Equating Recipes*. There are two classes of linear equating procedures: observed-score procedures and true-score procedures.

### 4.1 Observed-Score Equating

The basic equation for observed-score linear equating for the common-item nonequivalent groups design in terms of synthetic group ( $s$ ) means and standard deviations is

$$l_{Ys}(x) = \left[ \mu_s(Y) - \frac{\sigma_s(Y)}{\sigma_s(X)} \mu_s(X) \right] + \frac{\sigma_s(Y)}{\sigma_s(X)}(x), \quad (4.1)$$

where

$$\mu_s(X) = \mu_1(X) - w_2\gamma_1[\mu_1(V) - \mu_2(V)] \quad (4.2)$$

$$\mu_s(Y) = \mu_2(Y) + w_1\gamma_2[\mu_1(V) - \mu_2(V)] \quad (4.3)$$

$$\sigma_s^2(X) = \sigma_1^2(X) - w_2\gamma_1^2[\sigma_1^2(V) - \sigma_2^2(V)] + w_1w_2\gamma_1^2[\mu_1(V) - \mu_2(V)]^2 \quad (4.4)$$

$$\sigma_s^2(Y) = \sigma_2^2(Y) + w_1\gamma_2^2[\sigma_1^2(V) - \sigma_2^2(V)] + w_1w_2\gamma_2^2[\mu_1(V) - \mu_2(V)]^2 \quad (4.5)$$

with  $w_1$  and  $w_2$  being the synthetic population weights that are defined such that  $w_1 + w_2 = 1$ . For mean equating, the slope  $\sigma_s(Y)/\sigma_s(X)$  is set

to 1. It can be shown (see Kolen & Brennan, 2004, and Brennan, 2006) that the linear observed-score equating procedures differ only with respect to their  $\gamma$  terms.

#### 4.1.1 Tucker Equating

For Tucker equating, it is assumed that:

1. there is a linear regression of  $X$  on  $V$  that is the same in both populations, and
2. the conditional variance of  $X$  given  $V$  is the same for both populations.

Analogous assumptions are made for  $Y$  and  $V$ . Under these assumptions,

$$\gamma_1 = \alpha_1(X|V) = \frac{\sigma_1(X, V)}{\sigma_1^2(V)} \quad \text{and} \quad \gamma_2 = \alpha_2(Y|V) = \frac{\sigma_2(Y, V)}{\sigma_2^2(V)}. \quad (4.6)$$

These  $\gamma$  terms apply for both an internal and an external anchor. If the anchor is internal, then  $X$  and  $Y$  include scores for the common items  $V$ . If the anchor is external, then  $X$  and  $Y$  do not include scores for the common items  $V$ .

#### 4.1.2 Levine Observed-Score Equating

For Levine observed-score equating, assumptions are made about *true* scores for  $X$ ,  $Y$ , and  $V$ . Under these assumptions, and with the additional assumptions of a classical congeneric model, the  $\gamma$  terms for Levine observed-score equating with an internal anchor are

$$\gamma_1 = \frac{1}{\alpha_1(V|X)} = \frac{\sigma_1^2(X)}{\sigma_1(X, V)} \quad \text{and} \quad \gamma_2 = \frac{1}{\alpha_1(V|Y)} = \frac{\sigma_2^2(Y)}{\sigma_2(Y, V)}, \quad (4.7)$$

and with an external anchor

$$\gamma_1 = \frac{\sigma_1^2(X) + \sigma_1(X, V)}{\sigma_1^2(V) + \sigma_1(X, V)} \quad \text{and} \quad \gamma_2 = \frac{\sigma_2^2(Y) + \sigma_2(Y, V)}{\sigma_2^2(V) + \sigma_2(Y, V)}. \quad (4.8)$$

#### 4.1.3 Chained Linear Equating

For chained linear equating, Brennan (2006) shows that

$$\gamma_1 = \frac{\sigma_1(X)}{\sigma_1(V)} \quad \text{and} \quad \gamma_2 = \frac{\sigma_2(Y)}{\sigma_2(V)}. \quad (4.9)$$

Brennan (2006) shows that chained linear equating is invariant with respect to synthetic population weights.



## 4.2 Levine True-Score Equating

The true-score analogue of Equation 4.1 is

$$l_{Ys}(t_x) = \left[ \mu_s(T_Y) - \frac{\sigma_s(T_Y)}{\sigma_s(T_X)} \mu_s(T_X) \right] + \frac{\sigma_s(T_Y)}{\sigma_s(T_X)}(t_x). \quad (4.10)$$

Even if estimates of the synthetic means and standard deviations were available, however, Equation 4.10 could not be used directly, since true scores are unknown. So, typically  $t_x$  is replaced by  $x$ , without any real psychometric justification! Then, it can be shown that under Levine's assumptions, combined with the assumptions of a classical congeneric model (see Kolen & Brennan, 2004, pp. 112–115), Equation 4.10 becomes

$$l_Y(x) = \frac{\gamma_1}{\gamma_2} [x - \mu_1(X)] + \mu_2(Y) + \gamma_2 [\mu_1(V) - \mu_2(V)], \quad (4.11)$$

where the  $\gamma$  terms are given by Equation Set 4.7 for an internal anchor and Equation Set 4.8 for an external anchor. As indicated by the lack of  $s$  subscripts in Equation 4.11, Levine true-score equating applied to observed scores is invariant with respect to synthetic population weights. Hanson (1991) has shown that under Levine's assumptions with the additional assumptions of a classical congeneric model, Equation 4.11 possesses the property of first-order equity.

## 4.3 Functions

The wrapper function for linear (and equipercentile) equating with the common-item nonequivalent groups design is in `CG_EquiEquate.c` with the function prototypes in `CG_EquiEquate.h`. The functions to perform linear equating are in `CG_NoSmooth.c`, with the function prototypes in `CG_NoSmooth.h`. In this chapter the entire calling sequence for functions is discussed to provide the user with a perspective on how *Equating Recipes* distributes the tasks involved in getting the equating results. Generally, for other chapters in *Equating Recipes*, the wrapper and print functions are described in detail, but, in most cases, the user is directed to the C code for details about functions called by the wrapper functions.

`Wrapper_CN()` calls `CI_LinEq()`, which in turn calls `CI_LinObsEq()`. Then `Wrapper_CN()` calls `MomentsFromFD()`, which is described on page 20. After `Wrapper_CN()` returns to `main()`, `Print_CN()` can be called. So, the function call structure is:

- `Wrapper_CN()`
  - `CI_LinEq()`
    - \* `CI_LinObsEq()`

– MomentsFromFD()

- Print\_CN()

Wrapper\_CN() is a “high-level” function that uses structures so that only a few arguments need to be passed to the functions CI\_LinEq(), CI\_LinObsEq(), and MomentsFromFD(). These latter functions do most of the computational work. CI\_LinEq() and CI\_LinObsEq() have a large number of arguments that do not use structures, which should make it easier for users to modify the functions, if desired.

#### 4.3.1 Wrapper\_CN()

Wrapper\_CN() calls functions that perform mean, linear, or equipercentile equating (the last of which is discussed in Chapter 6) for the common-item nonequivalent groups design with no smoothing. The function prototype for Wrapper\_CN() is:

```
void Wrapper_CN(char design, char method, char smoothing,
                double w1, int anchor, double rv1, double rv2,
                struct BSTATS *xv, struct BSTATS *yv, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ **design** = 'C' designating the common-item nonequivalent groups design;
- ▷ **method** = 'M' or 'L' for mean or linear, respectively (several equipercentile options are discussed later);
- ▷ **smoothing** = 'N' (none);
- ▷ **w1** = weight for new Form X (if **w1** = -1, then **w1** is set proportional to the sample size for the new form);
- ▷ **anchor** = 0 means external anchor; **anchor** = 1 means internal anchor;
- ▷ **rv1** = reliability for common items for the new group (set to 0 for mean or linear equating);
- ▷ **rv2** = reliability for common items for the old group (set to 0 for mean or linear equating);
- ▷ **xv** = a BSTATS structure for the new form;
- ▷ **yv** = a BSTATS structure for the old form; and

- ▷ `rep` = replication number (must be set to 0 for actual equating; used to count replications when bootstrap standard errors are estimated).

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_CN()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

`Wrapper_CN()` populates the elements of the `PDATA` structure `inall`, dynamically allocates space for the `eraw[][]` and `mts[][]` matrices in the `ERAW_RESULTS` structure `r`, calls a function that does the actual equating and stores results in `r`, and calls the function `MomentsFromFD()` that gets the equated raw-score moments and stores them in `r`. Results are computed for four linear (or mean) equating methods: Tucker equating, Levine observed score equating, Levine true score equating, and chained equating.

#### 4.3.2 `CI_LinEq()`

`Wrapper_CN()` calls `CI_LinEq()` which does most of the computation for linear equating. The function prototype is:

```
void CI_LinEq(double mnx1, double sdx1, double mnv1, double sdv1,
             double covxv1, double mny2, double sdy2, double mnv2,
             double sdv2, double covyv2, double w1, int anchor,
             char method, double min, double max, double inc,
             int nm, double *msx, double *msy, double *ssx,
             double *ssy, double *gamma1, double *gamma2,
             double *a, double *b, double **eraw)
```

The input variables are:

- ▷ `mnx1` = mean for  $X$  for population 1;
- ▷ `sdx1` = sd for  $X$  for population 1;
- ▷ `mnv1` = mean for  $V$  for population 1;
- ▷ `sdv1` = sd for  $V$  for population 1;
- ▷ `covxv1` = covariance for  $Y$  and  $V$  for population 1;
- ▷ `mny2` = mean for  $Y$  for population 2;
- ▷ `sdv2` = sd for  $X$  for population 2;
- ▷ `mnv2` = mean for  $V$  for population 2;

- ▷ `sdv2` = sd for  $V$  for population 2;
- ▷ `covyv2` = covariance for  $Y$  and  $V$  for population 2;
- ▷ `w1` = weight for new Form  $X$ ;
- ▷ `anchor` = 0 means external anchor; `anchor` = 1 means internal anchor;
- ▷ `method` = 'M' or 'L' for mean or linear, respectively;
- ▷ `min` = minimum score for  $X$ ;
- ▷ `max` = maximum score for  $X$ ;
- ▷ `inc` = increment for  $X$ ;
- ▷ `nm` = 4 (number of methods); and
- ▷ `fdx[]` = frequency distribution for  $X$  in population 1.

The output variables provide results for each of the four methods in the following vectors or matrices:

- ▷ `msx[]` = means for  $X$  for the synthetic population;
- ▷ `msy[]` = means for  $Y$  for the synthetic population;
- ▷ `ssx[]` = sd's for  $X$  for the synthetic population;
- ▷ `ssy[]` = sd's for  $Y$  for the synthetic population;
- ▷ `gamma1[]` =  $\gamma$  terms for population 1;
- ▷ `gamma2[]` =  $\gamma$  terms for population 2;
- ▷ `a[]` = slopes;
- ▷ `b[]` = intercepts; and
- ▷ `eraw[][]` = equated raw scores; first dimension indexes methods (0 for Tucker, 1 for Levine observed, 2 for Levine true, 3 for chained).

#### 4.3.3 `CI_LinObsEq()`

`CI_LinEq()` calls `CI_LinObsEq()`, which computes observed-score linear equating results for a single method. The function prototype is:

```
void CI_LinObsEq(double mnx1, double sdx1, double mnv1, double sdv1,
                 double mny2, double sdy2, double mnv2, double sdv2,
                 double w1, char method, double gamma1, double gamma2,
```

```
double *msx, double *msy, double *ssx, double *ssy,
double *a, double *b)
```

The variables in the argument list are described in the foregoing discussion of `CI_LinEq()`. Note, however, that the last six arguments of `CI_LinObsEq()` are pointers to single, specific elements of vectors (corresponding to the particular method—0 for Tucker, 1 for Levine observed, 2 for Levine true, 3 for chained), as opposed to pointers to the first elements.

#### 4.3.4 `Print_CN()`

The function prototype for `Print_CN()` is:

```
void Print_CN(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the linear equating results for the common-item nonequivalent groups design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

## 4.4 Example

Table 4.1 is a `main()` function that illustrates linear equating with the common-item nonequivalent groups design based on an example discussed by Kolen and Brennan (2004, pp. 121–124). This example, which extends the `BSTATS` example in Section 2.2.2, has two parts:

- raw data for  $X$  and  $V$  are read, and statistics for this bivariate distribution are computed and stored in the `BSTATS` structure, `xv`; similarly, raw data for  $Y$  and  $V$  are read and statistics are stored in `yv`
- equated raw scores are computed using `Wrapper_CN()` and printed using `Print_CN()`.

Note that the `BSTATS` structures `xv` and `yv`, the `PDATA` structure `pdCLN`, and the `ERAW_RESULTS` structure `rCLN` are declared at the top of the `main()` function. (These structures are declared in `ERutilities.h`.) Also, note

that it is the addresses of all structures that are passed (designated by preceding each structure name with  $\&$ ), not the structures themselves.

TABLE 4.1. Main() Code to Illustrate Linear Equating with Common-item Nonequivalent Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xv,yv;
    struct PDATA pdCLN;
    struct ERAW_RESULTS rCLN;

    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    FILE *outf;

    outf = fopen("Chap 4 out","w");

    /* Common-item Nonequivalent Groups Design:
       Kolen and Brennan (2004) Chapter 4 example:
       Linear equating pp. 121-124 */

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'X','V",&xv);

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'X','V",&xv);

    Wrapper_CN('C','L','N',-1,1,0,0,&xv,&yv,0,&pdCLN,&rCLN);
    Print_CN(outf,"Chapter 4: proportional wts",&pdCLN,&rCLN);

    fclose(outf);
    return 0;
}
*****/

```

The output from `Print_CN()` is provided in Table 4.2. The top of the output provides synthetic group means and standard deviations for the Tucker and Levine observed-score methods,  $\gamma$  terms, slopes, and intercepts. The bottom part provides the equated raw scores for the four methods, as well as their moments based on using the new-form frequencies.

TABLE 4.2. Linear Equating Raw-Score Results for the Common-items Nonequivalent Groups Design

```

/*****
Chapter 4: proportional wts

Linear Equating with CINEG Design (Internal Anchor; w1 = 0.50258)

Input file for xv and pop 1: mondatx-temp (Variables: X and V)
Input file for yv and pop 2: mondaty-temp (Variables: Y and V)

```

	Method 0: Tucker	Method 1: Lev Obs	Method 2: Lev True	Method 3: ChainedL
msx	16.71406	17.01607		
msy	17.73924	17.45440		
ssx	6.66645	6.77399		
ssy	6.86083	6.84841		
gamma1	2.37514	3.17795	3.17795	2.74737
gamma2	2.45603	3.20542	3.20542	2.80582
inter b	0.53783	0.25137	0.29124	0.39368
slope a	1.02916	1.01099	1.00864	1.02127

```

-----
Equated Raw Scores on Scale of Y

Raw Score (X)    Method 0:    Method 1:    Method 2:    Method 3:
                  Tucker      Lev Obs      Lev True      ChainedL

```

0.00000	0.53783	0.25137	0.29124	0.39368
1.00000	1.56699	1.26236	1.29988	1.41495
2.00000	2.59615	2.27335	2.30853	2.43622
3.00000	3.62531	3.28433	3.31717	3.45749
4.00000	4.65447	4.29532	4.32581	4.47877
5.00000	5.68362	5.30631	5.33446	5.50004
6.00000	6.71278	6.31730	6.34310	6.52131
7.00000	7.74194	7.32828	7.35175	7.54258
8.00000	8.77110	8.33927	8.36039	8.56385
9.00000	9.80026	9.35026	9.36903	9.58513
10.00000	10.82941	10.36124	10.37768	10.60640
11.00000	11.85857	11.37223	11.38632	11.62767
12.00000	12.88773	12.38322	12.39497	12.64894
13.00000	13.91689	13.39420	13.40361	13.67021
14.00000	14.94605	14.40519	14.41226	14.69148
15.00000	15.97520	15.41618	15.42090	15.71276
16.00000	17.00436	16.42716	16.42954	16.73403
17.00000	18.03352	17.43815	17.43819	17.75530
18.00000	19.06268	18.44914	18.44683	18.77657
19.00000	20.09184	19.46012	19.45548	19.79784
20.00000	21.12099	20.47111	20.46412	20.81911
21.00000	22.15015	21.48210	21.47276	21.84039
22.00000	23.17931	22.49308	22.48141	22.86166
23.00000	24.20847	23.50407	23.49005	23.88293
24.00000	25.23763	24.51506	24.49870	24.90420
25.00000	26.26678	25.52604	25.50734	25.92547
26.00000	27.29594	26.53703	26.51599	26.94674
27.00000	28.32510	27.54802	27.52463	27.96802
28.00000	29.35426	28.55900	28.53327	28.98929
29.00000	30.38342	29.56999	29.54192	30.01056
30.00000	31.41257	30.58098	30.55056	31.03183
31.00000	32.44173	31.59197	31.55921	32.05310
32.00000	33.47089	32.60295	32.56785	33.07437
33.00000	34.50005	33.61394	33.57649	34.09565
34.00000	35.52921	34.62493	34.58514	35.11692
35.00000	36.55836	35.63591	35.59378	36.13819
36.00000	37.58752	36.64690	36.60243	37.15946
Mean	16.81967	16.24574	16.24854	16.55075
S.D.	6.71816	6.59955	6.58425	6.66668
Skew	0.57991	0.57991	0.57991	0.57991
Kurt	2.72166	2.72166	2.72166	2.72166

```

/*****

```



## 5

# Equipercntile Equating with Random-Groups and Single-Group Designs

Kolen and Brennan (2004, Section 2.5—especially 2.5.2) treat equipercntile equating with the random groups design in considerable detail. The fundamental equation is:

$$e_Y(x) = Q^{-1}[P(x)], \quad (5.1)$$

where  $P(x)$  is the percntile rank for  $x$  and  $Q^{-1}(\ast)$  is the percntile point function for  $Y$ . In other words,  $e_Y(x)$  is the score on the scale of  $Y$  associated with the percntile rank of  $P(x)$ . Note that the percntile point function for  $X$  is the inverse of the percntile rank function for  $X$ ; similarly for  $Y$ .

In equating situations, the  $x$  and  $y$  scores are discrete—usually non-negative integers corresponding to the number of items correct. This discreteness presents no problems in getting values for  $P(x)$ , but discreteness does usually present difficulties in getting percntile points on the scale of  $Y$ . The basic problem is that there is not likely to be an integer score on  $Y$  that has a percntile rank exactly equal to  $P(x)$ . To resolve this matter, the densities for  $X$  and  $Y$  need to be continuized in some manner. For integer scores, the traditional approach is as follows: for each  $x$ , assume a uniform distribution over the interval  $x - .5$  to  $x + .5$ ; similarly for  $y$ . Other approaches discussed in Chapters 12–14 modify this assumption.

Equating with the single-group design uses only the marginals of the bivariate distribution. It follows that, if there is no presmoothing of the bivar-

ate distribution, exactly the same approach applies for obtaining equipercentile equated scores.<sup>1</sup>

## 5.1 Functions

The functions to perform equipercentile equating with the random-groups and single-group design are in `RGandSG_NoSmooth.c`, with the function prototypes in `RGandSG_NoSmooth.h`. The `Wrapper_RN()` and `Print_RN()` functions are the same as those used for mean and linear equating discussed in Section 3.1. To obtain equipercentile results, however, the `method` variable in `Wrapper_RN()` should be set to `'E'`. When `Wrapper_RN()` is employed with equipercentile equating, it calls `EquiEquate()`, which is described on page 21.

## 5.2 Random Groups Example

Table 5.1 is a `main()` function that illustrates equipercentile equating with the random-groups design based on an example discussed by Kolen and Brennan (2004, pp. 50–53, 57, 60). Linear equating results for this example were considered in Section 3.2. This equipercentile example has three parts:

- read the raw data for Form X and Form Y and store statistics in two `USTATS` structures, `x` and `y` (same code as in Section 3.2);
- compute and print the equipercentile-equated raw scores using `Wrapper_RN()` (with `method` is set to `'E'`) and `Print_RN()`, respectively (see Section 3.1); and
- compute and print the unrounded and rounded scale scores using `Wrapper_ESS()` and `Print_ESS()`, respectively (see pages 23 and 25).

Note that the `PDATA` structure `pdREN`, the `ERAW_RESULTS` structure `rREN`, and the `ESS_RESULTS` structure `sREN` are declared at the top of the `main()` function.

The equated raw-score output is in Table 5.2 where the raw to equated raw-score conversion table is provided along with moments that are based on using the new-form frequencies. The unrounded and rounded equated scale scores are provided in Tables 5.3 and 5.4, respectively. The raw-to-scale score conversion for the old Form Y has already been provided in Table 3.3.

---

<sup>1</sup>The standard errors of equated raw scores will be different for the two designs.

TABLE 5.1. Main() Code to Illustrate Equipercntile Equating with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct PDATA pdREN;
    struct ERAW_RESULTS rREN;
    struct ESS_RESULTS sREN;

    FILE *outf;

    outf = fopen("Chap 5 out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004): Chapter 2 example:
       Equipercntile equating (see pp. 52, 53, 57, 60) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y",&y);

    Wrapper_RN('R','E','N",&x,&y,0,&pdREN,&rREN);
    Print_RN(outf,"ACT Math---Equipercntile",&pdREN,&rREN);

    Wrapper_ESS(&pdREN,&rREN,0,40,1,"yctmath.TXT",1,1,36,&sREN);
    Print_ESS(outf,"ACT Math---Equipercntile",&pdREN,&sREN);

    fclose(outf);
    return 0;
}
*****/

```

## 5.3 Zero Frequencies

In Table 5.2 the  $Y$  equivalent of a raw score of  $x = 0$  is 0. This result is related to the facts that: (a) for Form X the frequency of  $x = 0$  is 0; and (b) for Form Y the frequency of  $y = 0$  is 0, but the frequency of  $y = 1$  is non-zero. In this case, the  $Y$  equivalent of  $x = 0$  is 0 because the percentile point (on the  $Y$  scale) associated with a percentile rank of 0 (on the  $X$  scale) is 0.

By contrast, suppose that for Form Y the frequency of *both*  $y = 0$  and  $y = 1$  were 0, and the frequency for  $y = 2$  were non-zero. This means that the percentile ranks are 0 for both  $y = 0$  and  $y = 1$ . In this case, it could be

argued that the  $Y$  equivalent of a raw score of  $x = 0$  is anything between  $-.5$  and  $1.5$ . The `EquiEquate()` function employs an algorithm that effectively declares the  $Y$  equivalent to the midpoint of the interval, namely,  $.5$ . The same type of logic applies to zero frequencies at the top of the  $X$  and  $Y$  scales.

The convention discussed above is somewhat arbitrary, but a convention *is* necessary. In practical contexts, equating is often done with only a sample of the examinees who have taken, or will take, the new form. In such cases, zero frequencies are quite common in the tails of the distribution. However, typically there needs to be an equated score assigned to every possible raw score, since, in the full set of tested examinees, every possible raw score is likely to occur.

TABLE 5.2. Equipercentile Equating Raw-Score Results for the Random Groups Design

```

/*****
ACT Math--Equipercentile

Equipercentile Equating with Random Groups Design

Input file for x: actmathfreq.dat
Input file for y: actmathfreq.dat

-----

Equated Raw Scores

Raw Score (x)      Method 0:
                   y-equiv

0.00000            0.00000
1.00000            0.97956
2.00000            1.64622
3.00000            2.28563
4.00000            2.89320
5.00000            3.62047
6.00000            4.49965
7.00000            5.51484
8.00000            6.31242
9.00000            7.22424
10.00000           8.16067
11.00000           9.18270
12.00000           10.18590
13.00000           11.25130
14.00000           12.38963
15.00000           13.39289
16.00000           14.52401
17.00000           15.71690
18.00000           16.82344
19.00000           18.00922
20.00000           19.16472
21.00000           20.36760
22.00000           21.45563
23.00000           22.68712
24.00000           23.91566
25.00000           25.02916
26.00000           26.16123
27.00000           27.26329
28.00000           28.18006
29.00000           29.14243
30.00000           30.13048
31.00000           31.12970
32.00000           32.13571
33.00000           33.07807
34.00000           34.01719
35.00000           35.10160
36.00000           36.24255
37.00000           37.12476
38.00000           38.13209
39.00000           39.08073
40.00000           39.90055

Mean              18.97994
S.D.              8.93522
Skew              0.35453
Kurt              2.14650
*****/

```

TABLE 5.3. Equated Unrounded Scale Scores for Equipercentile Equating with the Random Groups Design

```

/*****
ACT Math---Equipercentile

Name of file containing yct[]: yctmath.TXT

Equated Scale Scores (unrounded)

Raw Score (x)      Method 0:
                   y-equiv
0.00000            0.50000
1.00000            0.50000
2.00000            0.50000
3.00000            0.50000
4.00000            0.50000
5.00000            0.50000
6.00000            0.59493
7.00000            1.18744
8.00000            2.10983
9.00000            3.46449
10.00000           4.92582
11.00000           6.36780
12.00000           7.73857
13.00000           9.26220
14.00000          10.84557
15.00000          12.10500
16.00000          13.44912
17.00000          14.87383
18.00000          16.15151
19.00000          17.39124
20.00000          18.49581
21.00000          19.61506
22.00000          20.55332
23.00000          21.47933
24.00000          22.26954
25.00000          22.93531
26.00000          23.61715
27.00000          24.29493
28.00000          24.84955
29.00000          25.35377
30.00000          25.78412
31.00000          26.21755
32.00000          26.72813
33.00000          27.29077
34.00000          27.92158
35.00000          28.79980
36.00000          30.10088
37.00000          31.38695
38.00000          32.89003
39.00000          34.29743
40.00000          35.33557

Mean              16.51256
S.D.              8.37253
Skew              -0.13002
Kurt              2.05146
*****/

```

TABLE 5.4. Equated Rounded Scale Scores for Equipercentile Equating with the Random Groups Design

```

/*****
ACT Math---Equipercentile

Name of file containing yct[]: yctmath.TXT

Equated Scale Scores (rounded)

Raw Score (x)      Method 0:
                   y-equiv

0.00000            1
1.00000            1
2.00000            1
3.00000            1
4.00000            1
5.00000            1
6.00000            1
7.00000            1
8.00000            2
9.00000            3
10.00000           5
11.00000           6
12.00000           8
13.00000           9
14.00000          11
15.00000          12
16.00000          13
17.00000          15
18.00000          16
19.00000          17
20.00000          18
21.00000          20
22.00000          21
23.00000          21
24.00000          22
25.00000          23
26.00000          24
27.00000          24
28.00000          25
29.00000          25
30.00000          26
31.00000          26
32.00000          27
33.00000          27
34.00000          28
35.00000          29
36.00000          30
37.00000          31
38.00000          33
39.00000          34
40.00000          35

Mean      16.43243
S.D.      8.39725
Skew      -0.12118
Kurt       2.02941
*****/

```





## 6

# Equipercentile Equating with the Common-item Nonequivalent Groups Design

Equipercentile equating with the common-item nonequivalent groups design is discussed by Kolen and Brennan (2004, chap. 5). Specifically, they provide an extensive discussion of frequency estimation (FE) and a limited discussion of chained equipercentile equating. Here these two methods are considered along with the modified frequency estimation (MFE) method developed by Wang and Brennan (2006, 2009). There is a linear-equating special case of FE first proposed by Braun and Holland (1982). Similarly, there is a linear equating special case for MFE.

### 6.1 Frequency Estimation

FE (sometimes called post-stratification) is essentially the equipercentile analogue of Tucker equating discussed in Section 4.1.1. FE involves conducting an equipercentile equating that puts  $X$  on the scale of  $Y$  using the synthetic population distributions

$$f_s(x) = w_1 f_1(x) + w_2 f_2(x) \quad (6.1)$$

and

$$g_s(y) = w_1 g_1(y) + w_2 g_2(y), \quad (6.2)$$

where  $f_1(x)$  and  $f_2(x)$  are the distributions for Form X in populations 1 and 2, respectively;  $g_1(y)$  and  $g_2(y)$  are the distributions for Form Y in populations 1 and 2, respectively; and  $s$  is the synthetic population with

$w_1 + w_2 = 1$ . Clearly,  $f_2(x)$  and  $g_1(y)$  are not available from the common-item nonequivalent groups design. To estimate them, it is assumed that,

$$f_1(x|v) = f_2(x|v) \text{ for all } v \quad \text{and} \quad g_2(y|v) = g_1(y|v) \text{ for all } v. \quad (6.3)$$

Then, since

$$f_2(x, v) = f_2(x|v)h_2(v) \quad \text{and} \quad g_1(y, v) = g_1(y|v)h_1(v), \quad (6.4)$$

it is easy to show that

$$f_s(x) = w_1 f_1(x) + w_2 \sum_v f_1(x|v)h_2(v) \quad (6.5)$$

and

$$g_s(y) = w_1 \sum_v g_2(y|v)h_1(v) + w_2 g_2(y), \quad (6.6)$$

where  $h_1(v)$  and  $h_2(v)$  are the marginal distributions of  $V$  in populations 1 and 2, respectively.

Braun-Holland equating under FE assumptions is simply Equation 4.1 using the first two moments of the synthetic densities for  $X$  and  $Y$  from FE equating.

## 6.2 Modified Frequency Estimation

Wang and Brennan (2006, 2009) show that there is good reason to believe that FE results are likely to be biased in many realistic circumstances. To mitigate this problem, they suggest replacing Equation Set 6.3 with corresponding assumptions based on conditioning on true scores for the common items,  $t_v$ :

$$f_1(x|t_v) = f_2(x|t_v) \text{ for all } v \quad \text{and} \quad g_2(y|t_v) = g_1(y|t_v) \text{ for all } v. \quad (6.7)$$

These assumptions are not directly useful, however, because we do not immediately have the distributions of observed scores conditional on true score.

Let us focus on  $X$  (corresponding results apply to  $Y$ ). We can use a certain relationship between true scores and observed scores to replace  $t_v$  with observed scores for  $V$ , so that we have

$$f_1(x|v_1) = f_2(x|v_2). \quad (6.8)$$

The observed data provide  $f_1(x|v_1)$  directly. To obtain  $f_2(x|v_2)$ , for every  $v_2$  we need to find the corresponding  $v_1$ . This is accomplished by using

Brennan and Lee's (2006) approach to estimating true scores from observed scores.<sup>1</sup> Their approach applied to MFE gives:

$$t_{v_1} = \mu_1(V) + \sqrt{\rho_1(V, V')} [v_1 - \mu_1(V)] \quad (6.9)$$

and

$$t_{v_2} = \mu_2(V) + \sqrt{\rho_2(V, V')} [v_2 - \mu_2(V)], \quad (6.10)$$

where  $\rho_1(V, V')$ , and  $\rho_2(V, V')$  are the reliabilities of  $V$  in the two populations. By setting  $t_{v_1} = t_{v_2}$ , for every  $v_2$  we can compute the corresponding  $v_1$ , namely,

$$v_1 = \frac{\sqrt{\rho_2(VV')}}{\sqrt{\rho_1(VV')}} v_2 + \frac{1 - \sqrt{\rho_2(VV')}}{\sqrt{\rho_1(VV')}} \mu_2(V) - \frac{1 - \sqrt{\rho_1(VV')}}{\sqrt{\rho_1(VV')}} \mu_1(V). \quad (6.11)$$

It is then possible to estimate  $f_s(x)$  using the basic ideas in Section 6.1, and, of course, the same approach can be used to estimate  $g_s(y)$ .

Braun-Holland equating under MFE assumptions is simply Equation 4.1 using the first two moments of the synthetic densities for  $X$  and  $Y$  from MFE equating.

## 6.3 Chained Equipercentile Equating

Chained equipercentile equating involves two initial steps:

1. using examinees from population 1, find  $e_{V1}(x)$ , which is the equipercentile transformation for converting scores on  $X$  to the scale of  $V$  in population 1; and
2. using examinees from population 2, find  $e_{Y2}(v)$ , which is the equipercentile transformation for converting scores on  $V$  to the scale of  $Y$  in population 2.

Then, chained equipercentile equating is

$$e_{Y(chain)} = e_{Y2}[e_{V1}(x)], \quad (6.12)$$

which converts  $X$  to  $V$  in population 1, and then converts these  $V$  equivalents to the scale of  $Y$  in population 2.

---

<sup>1</sup>The basic idea is to find a linear transformation of observed scores to estimated true scores such that the estimates have a variance equal to true score variance.

## 6.4 Zero Frequencies

There is often a very real possibility that some of the  $f_1(x, v)$  will be zero. This is particularly problematic when it leads to one or more of the  $h_1(v)$  being zero, since

$$f_1(x|v) = \frac{f_1(x, v)}{h_1(v)},$$

which is needed (see Equation 6.3) but undefined when  $h_1(v) = 0$ . [A similar argument holds, of course, for  $g_2(y, v)$ ,  $g_2(y|v)$ , and  $h_2(v)$ .]

Various approaches might be taken to deal with this problem. For example, Kolen and Brennan (2004, pp. 142–143) note that if  $h_1(v) = 0$ , then  $h_1(v+1)$  might be used for  $h_1(v)$ . In *Equating Recipes* a different approach is taken that involves mixing the actual bivariate frequency distribution with a “small” uniform distribution in the sense discussed next.

Let  $n_x$  be the number of score categories for  $X$  and  $n_v$  be the number of score categories for  $V$ . Then the “smoothed” relative frequency for  $x$  and  $v$  is

$$\tilde{f}(x, v) = [1 - (1.0e-10)] f(x, v) + \frac{1.0e-10}{n_x n_v}. \quad (6.13)$$

Obviously, when  $f(x, v) = 0$ ,  $\tilde{f}(x, v) = (1.0e-10)/(n_x n_v)$ . Furthermore,  $\sum_x \sum_v \tilde{f}(x, v) = 1$ , which makes  $\tilde{f}(x, v)$  a “true” density. In effect, the small relative frequency added to each cell with  $f(x, v) = 0$  is borrowed from the other cells in a uniform manner.

## 6.5 Functions

The wrapper and print functions for equipercntile equating with the common-item nonequivalent groups design are in `CG_NoSmooth.c`, with the function prototypes in `CG_NoSmooth.h`. Most of the functions that actually perform the equipercntile equating are in `CG_EquiEquate.c`, with the function prototypes in `CG_EquiEquate.h`. The utility functions `EquiEquate()` and `perc_point()` are also involved in equipercntile equating (see page 21).

The wrapper function is the same function as in Chapter 4, namely, `Wrapper_CN()` (see page 50). There are only two differences involved in using it for equipercntile equating. First, `method` must be one of the following:

- ‘E’ = FE and Braun-Holland under FE (BH-FE)
- ‘F’ = MFE and Braun-Holland under MFE (BH-MFE)
- ‘G’ = FE, BH-FE, MFE, and BH-MFE

- 'C' = Chained
- 'H' = FE, BH-FE, and Chained
- 'A' = FE, BH-FE, MFE, BH-MFE, and Chained

Second, if MFE is requested ('F', 'G', or 'A'), then reliability estimates must be provided for `rv1` and `rv2`.

The print function for equipercentile equating with the common-item nonequivalent groups design is identically the same as `Print_CN()` described on page 53. There is also a function to print synthetic densities. Its name is `Print_SynDens()`, and its argument list is the same as `Print_CN()`.

## 6.6 Example

Table 6.1 is a `main()` function that illustrates equipercentile equating with the common-item nonequivalent groups design based on an example discussed by Kolen and Brennan (2004, chap. 5, p. 151). This example uses the same data sets as the linear-equating example in Section 4.4. It is instructive to compare the `main()` function in Table 6.1 (equipercentile equating) with the `main()` function in Table 4.1 (linear equating). There are many similarities, as well as several important differences. In particular, note that the `Wrapper_CN()` function

- uses 'A' as the `method`,
- uses `w1 = 1` as the weight for the new group (fourth parameter in `Wrapper_CN()`), and
- includes specific reliabilities for the common items for the old and new forms.

The output from `Print_CN()` is provided in Table 6.2. The top of the output provides general information followed by the Braun-Holland slopes and intercepts for FE and MFE. The bottom part provides the equated raw scores for the five methods (FE, BH-FE, MFE, BH-MFE, and Chained), as well as their moments based on using the new-form frequencies.

The synthetic-densities output from `Print_SynDens()` is provided in Tables 6.3 and Table 6.4, for FE and MFE, respectively.

TABLE 6.1. Main() Code to Illustrate Equipercntile Equating with Common-item Nonequivalent Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xv,yv;
    struct PDATA pdCEN;
    struct ERAW_RESULTS rCEN;

    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    FILE *outf;

    outf = fopen("Chap 6 out","w");

    /* Common-item Nonequivalent Groups Design:
       Kolen and Brennan (2004) Chapter 5 example:
       Equipercntile equating (see p. 151) */

    convertFtoW("mondatx.dat",2,fieldsACT,"mondatx-temp");
    ReadRawGet_BSTATS("mondatx-temp",1,2,0,36,1,0,12,1,'X','V',&xv);

    convertFtoW("mondaty.dat",2,fieldsACT,"mondaty-temp");
    ReadRawGet_BSTATS("mondaty-temp",1,2,0,36,1,0,12,1,'Y','V',&yv);

    Wrapper_CN('C','A','N',1,1,.5584431,.5735077,&xv,&yv,0,
               &pdCEN,&rCEN);
    Print_CN(outf,"Chapter 5 in K&B: w1 = 1",&pdCEN,&rCEN);
    Print_SynDens(outf,"Chapter 5 in K&B: w1 = 1",&pdCEN,&rCEN);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 6.2. Equipercntile Equating Raw-Score Results for the Common-item Nonequivalent Groups Design

/\*\*\*\*\*  
Chapter 5 in K&B: w1 = 1

Equipercntile Equating with CINEG Design (Internal Anchor; w1 = 1.00000)

Input file for xv and pop 1: mondatx-temp (Variables: X and V)

Input file for yv and pop 2: mondaty-temp (Variables: Y and V)

Braun-Holland (BH) Under Frequency Estimation (FE):

Intercept: b = 0.83338; Slope: a = 1.01131

Braun-Holland (BH) Under Modified Frequency Estimation (MFE):

Intercept: b = 0.70548; Slope: a = 0.99933

Equated Raw Scores on Scale of Y					
Raw Score (X)	Method 0: FE	Method 1: BH-FE	Method 2: MFE	Method 3: BH-MFE	Method 4: ChainedE
0.00000	1.00000	0.83338	1.00000	0.70548	1.00000
1.00000	1.00000	1.84469	1.00000	1.70481	1.00000
2.00000	2.89286	2.85601	2.83926	2.70414	2.89286
3.00000	4.03552	3.86732	3.90862	3.70347	4.08333
4.00000	4.84387	4.87863	4.77972	4.70280	4.92500
5.00000	5.53432	5.88995	5.51389	5.70213	5.58000
6.00000	6.16675	6.90126	6.16102	6.70146	6.23333
7.00000	7.35490	7.91257	7.18868	7.70079	7.38850
8.00000	8.61427	8.92389	8.40961	8.70012	8.54618
9.00000	9.79061	9.93520	9.58156	9.69945	9.66334
10.00000	10.82032	10.94652	10.58628	10.69878	10.59055
11.00000	11.91254	11.95783	11.66415	11.69812	11.59291
12.00000	13.22028	12.96914	12.87559	12.69745	12.77892
13.00000	14.34805	13.98046	14.03795	13.69678	13.93697
14.00000	15.32075	14.99177	15.04007	14.69611	14.88501
15.00000	16.37138	16.00308	16.02670	15.69544	15.95149
16.00000	17.21688	17.01440	16.92291	16.69477	16.88265
17.00000	18.20768	18.02571	17.86760	17.69410	17.81875
18.00000	19.17479	19.03702	18.81137	18.69343	18.80357
19.00000	20.02742	20.04834	19.59603	19.69276	19.54000
20.00000	21.04662	21.05965	20.52303	20.69209	20.46971
21.00000	22.18786	22.07096	21.69126	21.69142	21.85420
22.00000	23.12848	23.08228	22.71081	22.69075	22.96412
23.00000	24.06153	24.09359	23.61006	23.69008	23.92366
24.00000	24.90362	25.10490	24.45623	24.68941	24.75472
25.00000	25.85258	26.11622	25.39647	25.68874	25.64424
26.00000	26.87355	27.12753	26.46715	26.68808	26.66786
27.00000	27.83696	28.13885	27.39229	27.68741	27.58824
28.00000	29.04966	29.15016	28.54030	28.68674	28.82973
29.00000	29.99944	30.16147	29.66089	29.68607	29.90714
30.00000	31.01396	31.17279	30.62753	30.68540	31.15625
31.00000	31.95474	32.18410	31.53654	31.68473	32.27593
32.00000	32.74011	33.19541	32.52461	32.68406	32.84705
33.00000	33.34331	34.20673	33.29563	33.68339	33.33676
34.00000	34.41848	35.21804	34.22987	34.68272	34.31250
35.00000	35.42161	36.22935	35.30252	35.68205	35.41250
36.00000	36.09375	37.24067	35.98835	36.68138	36.09375
Mean	16.83581	16.83291	16.51533	16.51543	16.55556
S.D.	6.59496	6.60168	6.51422	6.52346	6.58886
Skew	0.46456	0.57991	0.49605	0.57991	0.54402
Kurt	2.62381	2.72166	2.68287	2.72166	2.69409

/\*\*\*\*\*

TABLE 6.3. Synthetic Densities of  $X$  and  $Y$  Under Frequency Estimation

```

/*****
Chapter 5 in K&B: w1 = 1

Synthetic Densities and Equated Scores Under
Frequency Estimation (w1 = 1.00000)

Input file for xv and pop 1: mondatx-temp
Input file for yv and pop 2: mondaty-temp

```

	Raw (X)	fxs	Raw (Y)	gys	FE
	0.00000	0.000000000	0.00000	0.000000000	1.00000
	1.00000	0.000000000	1.00000	0.000000000	1.00000
	2.00000	0.000604230	2.00000	0.000000000	2.89286
	3.00000	0.003021148	3.00000	0.000769020	4.03552
	4.00000	0.005438066	4.00000	0.002513046	4.84387
	5.00000	0.007854985	5.00000	0.008905611	5.53432
	6.00000	0.021752266	6.00000	0.023407420	6.16675
	7.00000	0.030815710	7.00000	0.021620490	7.35490
	8.00000	0.046525680	8.00000	0.031123382	8.61427
	9.00000	0.051963746	9.00000	0.038595515	9.79061
	10.00000	0.055589124	10.00000	0.051820223	10.82032
	11.00000	0.068277946	11.00000	0.053121101	11.91254
	12.00000	0.077341390	12.00000	0.062607881	13.22028
	13.00000	0.054380665	13.00000	0.050022390	14.34805
	14.00000	0.068277946	14.00000	0.061161897	15.32075
	15.00000	0.048338369	15.00000	0.063400627	16.37138
	16.00000	0.054984894	16.00000	0.053872865	17.21688
	17.00000	0.052567976	17.00000	0.062398684	18.20768
	18.00000	0.044108761	18.00000	0.051025848	19.17479
	19.00000	0.030211480	19.00000	0.049530439	20.02742
	20.00000	0.044108761	20.00000	0.039915226	21.04662
	21.00000	0.032628399	21.00000	0.033473439	22.18786
	22.00000	0.032628399	22.00000	0.033716724	23.12848
	23.00000	0.024773414	23.00000	0.035170285	24.06153
	24.00000	0.021148036	24.00000	0.027842766	24.90362
	25.00000	0.025377644	25.00000	0.026640035	25.85258
	26.00000	0.018731118	26.00000	0.020917970	26.87355
	27.00000	0.018731118	27.00000	0.022786268	27.83696
	28.00000	0.013897281	28.00000	0.013225642	29.04966
	29.00000	0.013293051	29.00000	0.013726797	29.99944
	30.00000	0.010271903	30.00000	0.014843613	31.01396
	31.00000	0.004833837	31.00000	0.008468288	31.95474
	32.00000	0.008459215	32.00000	0.007557997	32.74011
	33.00000	0.004229607	33.00000	0.010517970	33.34331
	34.00000	0.003625378	34.00000	0.002481716	34.41848
	35.00000	0.000604230	35.00000	0.002075160	35.42161
	36.00000	0.000604230	36.00000	0.000743667	36.09375
Mean		15.820543807		16.832910126	
S.D.		6.527825708		6.601678506	
Skew		0.579908289		0.462186004	
Kurt		2.721659901		2.622854041	
Sum of RF's		1.000000000		1.000000000	

```

/*****

```



TABLE 6.4. Synthetic Densities of  $X$  and  $Y$  Under Modified Frequency Estimation

```

/*****
Chapter 5 in K&B: w1 = 1

Synthetic Densities and Equated Scores Under
Modified Frequency Estimation (w1 = 1.00000)

Input file for xv and pop 1: mondatx-temp
Input file for yv and pop 2: mondaty-temp

```

	Raw (X)	fxs	Raw (Y)	gys	MFE
	0.00000	0.000000000	0.00000	0.000000000	1.00000
	1.00000	0.000000000	1.00000	0.000000000	1.00000
	2.00000	0.000604230	2.00000	0.000000000	2.83926
	3.00000	0.003021148	3.00000	0.000890515	3.90862
	4.00000	0.005438066	4.00000	0.002996164	4.77972
	5.00000	0.007854985	5.00000	0.008786441	5.51389
	6.00000	0.021752266	6.00000	0.022875939	6.16102
	7.00000	0.030815710	7.00000	0.026905846	7.18868
	8.00000	0.046525680	8.00000	0.033304575	8.40961
	9.00000	0.051963746	9.00000	0.041891818	9.58156
	10.00000	0.055589124	10.00000	0.053242218	10.58628
	11.00000	0.068277946	11.00000	0.056523512	11.66415
	12.00000	0.077341390	12.00000	0.062667684	12.87559
	13.00000	0.054380665	13.00000	0.054390835	14.03795
	14.00000	0.068277946	14.00000	0.059297321	15.04007
	15.00000	0.048338369	15.00000	0.062826681	16.02670
	16.00000	0.054984894	16.00000	0.055842910	16.92291
	17.00000	0.052567976	17.00000	0.059660558	17.86760
	18.00000	0.044108761	18.00000	0.052631640	18.81137
	19.00000	0.030211480	19.00000	0.048347127	19.59603
	20.00000	0.044108761	20.00000	0.040266149	20.52303
	21.00000	0.032628399	21.00000	0.033037611	21.69126
	22.00000	0.032628399	22.00000	0.031851398	22.71081
	23.00000	0.024773414	23.00000	0.032583131	23.61006
	24.00000	0.021148036	24.00000	0.027134990	24.45623
	25.00000	0.025377644	25.00000	0.024624607	25.39647
	26.00000	0.018731118	26.00000	0.020167336	26.46715
	27.00000	0.018731118	27.00000	0.020249701	27.39229
	28.00000	0.013897281	28.00000	0.013648581	28.54030
	29.00000	0.013293051	29.00000	0.012025392	29.66089
	30.00000	0.010271903	30.00000	0.012768595	30.62753
	31.00000	0.004833837	31.00000	0.008376747	31.53654
	32.00000	0.008459215	32.00000	0.006688406	32.52461
	33.00000	0.004229607	33.00000	0.008228584	33.29563
	34.00000	0.003625378	34.00000	0.003077039	34.22987
	35.00000	0.000604230	35.00000	0.001599474	35.30252
	36.00000	0.000604230	36.00000	0.000590476	35.98835
Mean		15.820543807		16.515432599	
S.D.		6.527825708		6.523456394	
Skew		0.579908289		0.493594629	
Kurt		2.721659901		2.679559214	
Sum of RF's		1.000000000		1.000000000	

```

/*****

```



# 7

## Analytic Standard Errors

Kolen and Brennan (2004, chap. 7) discuss estimating standard errors of estimated score equivalents using the bootstrap procedure as well as analytic procedures,<sup>1</sup> primarily the delta method. The *Equating Recipes* implementation of the bootstrap is treated in Chapter 8. Here we briefly discuss the delta method.

### 7.1 Delta Method

A theoretical discussion of the delta method is provided by Kendall and Stuart (1977). It is essentially a second order approximation to a Taylor series expansion. Let the  $Y$ -equivalent of  $x_i$  in the population be denoted as  $eq_Y(x_i; \Theta_1, \Theta_2, \dots, \Theta_t)$ , which is clearly conditional on the parameters  $\Theta_1, \Theta_2, \dots, \Theta_t$ . In linear equating,  $\Theta_1, \Theta_2, \dots, \Theta_t$  are moments; in equipercentile equating,  $\Theta_1, \Theta_2, \dots, \Theta_t$  are cumulative probabilities. Under the delta method, an approximate expression for the sampling variance of the estimated  $Y$ -equivalents of  $x_i$  is

$$var[\hat{eq}_Y(x_i)] \cong \sum_j (eq'_{Yj})^2 var(\hat{\Theta}_j) + \sum_{j \neq k} eq'_{Yj} eq'_{Yk} cov(\hat{\Theta}_j, \hat{\Theta}_k), \quad (7.1)$$

---

<sup>1</sup>The word “analytic” has different connotations in different literature. Here, it means a procedure that can be represented in a closed-form formula, as opposed to a procedure like the bootstrap that is a resampling procedure.

where  $eq'_{Yj}$  is the partial derivative of  $eq_{Yj}$  with respect to  $\hat{\Theta}_j$  evaluated at  $x_i, \Theta_1, \Theta_2, \dots, \Theta_t$ . The standard error is the square root of  $var[\hat{eq}_Y(x_i)]$  in Equation 7.1.

## 7.2 Functions

Kolen and Brennan (2004, p. 250) give references that provide analytic standard errors for various equating methods discussed in Kolen and Brennan (2004). Section 7.3 in Kolen and Brennan (2004) provides equations for several analytic standard errors, including equipercentile equating under the random groups design, a function for which is discussed below.

```
void SE_EPequate(int nsy, double npy, double *crfdy,
                 int nsx, double incx, double npx,
                 double *prdx, double *se_eeq)
```

The input variables are:

- ▷ **nsy** = number of raw score categories for old Form Y;
- ▷ **npy** = number of persons who took old Form Y;
- ▷ **crfdy[]** = cumulative relative frequency distribution for old Form Y;
- ▷ **nsx** = number of raw score categories for new Form X;
- ▷ **incx** = raw-score increment for new Form X;
- ▷ **npx** = number of persons who took new Form X; and
- ▷ **prdx[]** = percentile rank distribution for new Form X.

The output variable is **se\_eeq[]**, which provides the estimated standard errors for the estimated equipercentile  $Y$ -equivalents of  $x_i$ .

The **SE\_EPequate()** function is in **Analytic\_SEs.c**, which has an associated header file **Analytic\_SEs.h**. It is anticipated that other standard error functions will be added to **Analytic\_SEs.c** over time.

**Print\_vector()** can be used to print the elements in **se\_eeq[]**. See page 26 for a discussion of the **Print\_vector()** arguments.

## 7.3 Example

Table 7.1 is a **main()** function that provides estimated standard errors for equipercentile equating with the random-groups design based on an

example discussed by Kolen and Brennan (2004, pp. 50–53, 57, 60). The equipercentile equivalents for this example are provided in Section 5.2, and much of the code in Table 7.1 is the same as in Table 5.1.

Since the raw scores for this example are integers from 0 to 40, there are 41 estimated standard errors that are stored in the vector `se_eeq[41]` declared at the top of the `main()` function. The function calls for computing and printing the estimated standard errors are:

```
SE_EPequate(y.ns, y.n, y.crfd, x.ns, x.inc, x.n, x.prd, se_eeq);
Print_vector(outf, "ACT Math---Equipercentile delta SE's",
             se_eeq, x.ns, "    Raw Score", "          SE");
```

In the examples in previous chapters, the wrapper functions passed the addresses of structures (designated with `&` preceding the structure names). In this example, for the USTATS structures `x` and `y`, selected variables (designated with `.` preceding the variable names) are passed to `SE_EPequate()`.

The statements in Table 7.1 that are enclosed in `/*...*/` are *not* required to obtain the estimated standard errors. Rather, they are the statements that would be needed to get the *Y*-equivalents of the new-form raw scores, which are reported in Table 5.2.

The estimated standard errors are provided in Table 7.2. Note that the estimated standard error for  $x = 0$  is listed as 0, which is obviously impossible unless the population were available. For these particular data, however, no examinee got a score of 0 on Form Y (or Form X, for that matter). Whenever there is a zero frequency for a score of  $y$ , `SE_EPequate()` reports an estimated standard error of 0. An investigator might want to get an actual estimate through interpolation or extrapolation.

TABLE 7.1. Main() Code to Illustrate Estimating Analytic Raw-Score Standard Errors for Equipercentile Equating with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;

    /* The next two statements are not needed to get SE's only
    struct PDATA pdREN;
    struct ERAW_RESULTS rREN;
    */

    double se_eeq[41];      /* analytic se's for equi equating */
    FILE *outf;

    outf = fopen("Chap 7 out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004): Chapter 2 example:
       Standard Errors of Equipercentile equating (see pp. 81) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y",&y);

    /* The next statement is not needed to get the SE's, only
    Wrapper_RN('R','E','N",&x,&y,0,&pdREN,&rREN);
    */

    SE_EPequate(y.ns, y.n, y.crfd, x.ns, x.inc, x.n, x.prd, se_eeq);
    Print_vector(outf, "ACT Math---Equipercentile delta SE's",
                  se_eeq, x.ns, "    Raw Score", "          SE");

    fclose(outf);
    return 0;
}
*****/

```

TABLE 7.2. Estimated Analytic  
Raw-Score Standard Errors for  
Equipercentile Equating with Ran-  
dom Groups Design

/\*\*\*\*\*/  
ACT Math--Equipercentile delta SE's

Raw Score	SE
0	0.00000
1	0.83055
2	0.52100
3	0.82097
4	0.29502
5	0.14781
6	0.25411
7	0.15818
8	0.19691
9	0.17612
10	0.17312
11	0.19516
12	0.17995
13	0.23109
14	0.24312
15	0.21385
16	0.27635
17	0.26173
18	0.33835
19	0.28261
20	0.29473
21	0.32987
22	0.31827
23	0.38646
24	0.35546
25	0.30133
26	0.36831
27	0.35323
28	0.30691
29	0.34220
30	0.28963
31	0.32680
32	0.33093
33	0.30477
34	0.30798
35	0.30435
36	0.32400
37	0.27137
38	0.34301
39	0.20179
40	0.27872

/\*\*\*\*\*/





## 8

# Bootstrap Standard Errors

For nearly all equating methods, analytic procedures exist that provide approximate estimates of standard errors for equated scores, but these procedures are limited in that they do not provide estimates for unrounded or rounded scale scores. Almost always, that is a rather serious limitation, because in the vast majority of testing programs the only reported scores are scale scores—usually rounded scale scores. If scale scores are linear transformations of raw scores, it is easy to transform raw-score estimates to unrounded scale-score estimates, but most scale scores are non-linear transformations. In any case, rounded scale scores are virtually certain to be non-linear transformations of raw scores.

The bootstrap provides a way to circumvent these problems and obtain estimated standard errors for equated raw scores and equated scale scores—both unrounded and rounded. The bootstrap is discussed in detail by Kolen and Brennan (2004, sect. 7.2).

For the random groups design, the algorithm for estimating standard errors for equated raw scores is as follows:

1. get the frequency distribution of the raw scores for a random sample *with* replacement of the  $N_1$  persons who took the new Form X; and get the frequency distribution of the raw scores for a random sample *with* replacement of the  $N_2$  persons who took the new Form Y;
2. using these frequency distributions (or simply their first two moments for linear procedures), get the Form Y equivalents of the Form X scores—call these the bootstrap equivalents;

3. repeat steps 1 and 2  $R$  times; and
4. for each possible raw score on Form X, get the standard deviation of the  $R$  bootstrap equivalents. These standard deviations are the estimated bootstrap standard errors for the equated scores.

To get bootstrap estimates of standard errors for unrounded or rounded scale scores, Step 2 is expanded as follows:

transform the Form Y equivalents to scale scores, and round them if desired.

For the single group design, Step 1 is replaced so that it involves sampling with replacement from a single group, which results in a single bivariate distribution, with the marginals used in Step 2. For the common-item nonequivalent groups design, Step 1 is replaced so that it involves sampling with replacement from both the old and new groups, which results in two bivariate distributions that are used to get the synthetic (marginal) frequency distributions used in Step 2.

There are two types of bootstrap procedures: the nonparametric bootstrap and the parametric bootstrap. The above procedure is the nonparametric bootstrap. The parametric bootstrap begins by fitting a parametric model to the data. The fitted model gives a (bivariate) frequency distribution that is treated as the population distribution, and the bootstrap sampling is from it. In *Equating Recipes*, the nonparametric bootstrap is used when there is no presmoothing. The parametric bootstrap is used when there is presmoothing (see Chapters 9 and 10).

## 8.1 Functions

Only one function, `Wrapper_Bootstrap()`, is required to estimate bootstrap standard errors for raw scores, scale scores, and rounded scale scores. There are two print functions: `Print_Boot_se_erau()` for raw scores and `Print_Boot_se_ess()` for scale scores (both rounded and unrounded). These functions, as well as many functions called by `Wrapper_Bootstrap()`, are included on `Bootstrap.c`, which has an associated header file named `Bootstrap.h`.

### 8.1.1 `Wrapper_Bootstrap()`

```
void Wrapper_Bootstrap(struct PDATA *inall, int nrep,
                      long *idum, struct BOOT_ERAW_RESULTS *t,
                      struct BOOT_ESS_RESULTS *u)
```

The input variables are:

- ▷ `inall` = the `PDATA` structure associated with the actual equating;
- ▷ `nrep` = number of bootstrap replications; and
- ▷ `idum` = random-number seed.

The output variables are:

- ▷ `t` = a `BOOT_ERAW_RESULTS` structure that stores the bootstrap results for raw scores; and
- ▷ `u` = a `BOOT_ESS_RESULTS` structure that stores the bootstrap results for scale scores (`NULL` in the calling sequence indicates no bootstrapping for scale scores).

`Wrapper_Bootstrap()` calls the following functions (in addition to a few *Numerical Recipes* utilities):

- `Wrapper_RN()` in `RGandSG_NoSmooth.c` (see page 38);
- `Wrapper_SN()` in `RGandSG_NoSmooth.c` (see page 39);
- `Wrapper_CN()` in `CG_NoSmooth.c` (see page 50);
- `Wrapper_RB()` in `BetaBinomial.c` (see page 97);
- `Wrapper_RL()` in `LogLinear.c` (see page 121);
- `Wrapper_SL()` in `LogLinear.c` (see page 121);
- `Boot_USTATS()` in `Bootstrap.c`;
- `Boot_BSTATS()` in `Bootstrap.c`;
- `Parametric_boot_univ_BB()` in `Bootstrap.c`;
- `Parametric_boot_univ_ULL()` in `Bootstrap.c`;
- `Parametric_boot_biv()` in `Bootstrap.c`;
- `Boot_initialize_eraw()` in `Bootstrap.c`;
- `Boot_accumulate_eraw()` in `Bootstrap.c`;
- `Boot_accumulate_ess()` in `Bootstrap.c`;
- `Boot_se_eraw()` in `Bootstrap.c`;
- `Boot_se_ess()`; in `Bootstrap.c`; and
- `Wrapper_ESS()` in `ERutilities.c`.

It is particularly important to note that `Wrapper_Bootstrap()` uses identically the same wrapper functions that are used to get results for actual equatings.

To obtain bootstrap estimated standard errors for raw scores, the wrapper function for the method must be called before calling `Wrapper_Bootstrap()`. To obtain bootstrap estimated standard errors for scale scores, the wrapper function for the method and `Wrapper_ESS()` must be called before calling `Wrapper_Bootstrap()`.

Although `Wrapper_Bootstrap()` has a very short argument list, it can perform bootstrapping for “any method.” This feat is possible because the `PDATA` structure `inall` contains all of the data needed to do the bootstrapping for raw scores and scale scores (provided `u`  $\neq$  `NULL`). In fact, a principal reason for having wrapper functions for the various methods is to store data in a `PDATA` structure, the address of which can be used in a call to `Wrapper_Bootstrap()`. That is also one of the reasons that the wrapper functions for the various methods contain somewhat more complex code than many other *Equating Recipes* functions.

The phrase “any method” is in quotes in the above paragraph because there is some code that needs to be included in `Wrapper_Bootstrap()` for each method. The currently distributed version of `Wrapper_Bootstrap()` applies to the methods incorporated in `Wrapper_RN()`, `Wrapper_SN()`, `Wrapper_CN()`, `Wrapper_RL()`, `Wrapper_RB()`, and `Wrapper_SL()` (the last three of which are discussed in Chapters 9 and 10).

A user who wants to enable `Wrapper_Bootstrap()` to perform bootstrapping for a new method needs to:

1. add any necessary elements (variables, pointers to variables, or structures) to the structure `PDATA` in `ERutilities.h`;
2. add any necessary elements (variables, pointers to variables, or structures) within the `if(in->rep == 0){ }` statement in the wrapper function for the method;
3. add any necessary declarations (variables, structures, and/or headers) at the beginning of `Wrapper_Bootstrap()`; and
4. add appropriate code between the `start` and `end` comments in `Wrapper_Bootstrap()`.

As an example, these steps are specified in explicit detail for the method discussed in Chapter 9 (random groups equipercenile equating with beta-binomial presmoothing) (see page 98).

### 8.1.2 Print\_Boot\_se\_eraw()

The print function for bootstrap estimates of standard errors for equated raw scores is:

```
void Print_Boot_se_eraw(FILE *fp, char tt[], struct PDATA *inall,
                        struct ERAW_RESULTS *r,
                        struct BOOT_ERAW_RESULTS *t, int mdiff)
```

The input variables are:

- ▷ **fp** = output file pointer;
- ▷ **tt[]** = user-specified title;
- ▷ **inall** = the PDATA structure associated with the actual equating;
- ▷ **r** = the ERAW\_RESULTS structure that contains the equated raw scores for the actual equating;
- ▷ **t** = the BOOT\_ERAW\_RESULTS structure that contains the bootstrap results for raw scores; and
- ▷ **mdiff** = 1 means print differences between actual equated raw scores and means of **nrep** bootstrap equated raw scores (large values for such differences strongly suggest the need for more bootstrap replications); no differences printed if **mdiff** = 0.

### 8.1.3 Print\_Boot\_se\_ess()

If **Wrapper\_ESS** (see page 23) was called prior to **Wrapper\_Bootstrap**, then estimated standard errors for unrounded scale scores are in an **ESS\_RESULTS** structure. In addition, bootstrap estimated standard errors for rounded scale scores are in the **ESS\_RESULTS** structure if the **Wrapper\_ESS** argument **round** > 0. (See page 24 for a discussion of **round**.)

The print function for bootstrap estimates of standard errors for unrounded and rounded equated scale scores is:

```
void Print_Boot_se_ess(FILE *fp, char tt[], struct PDATA *inall,
                      struct ESS_RESULTS *s,
                      struct BOOT_ESS_RESULTS *u, int mdiff)
```

The input variables are:

- ▷ **fp** = output file pointer;
- ▷ **tt[]** = user-specified title;
- ▷ **inall** = the PDATA structure associated with the actual equating;

- ▷ **s** = the **ESS\_RESULTS** structure that contains the equated scale scores for the actual equating; this is the structure that is populated when **Wrapper\_ESS()** is called (see page 23);
- ▷ **u** = the **BOOT\_ESS\_RESULTS** structure that contains the bootstrap results for scale scores; and
- ▷ **mdiff** = 1 means print differences between actual equated scale scores and means of **nrep** bootstrap equated scale scores (large values for such differences strongly suggest the need for more bootstrap replications); no differences printed if **mdiff** = 0.

Bootstrap estimated standard errors for unrounded scale scores are always provided. In addition, bootstrap estimated standard errors for rounded scale scores are provided when **inall->round** > 0.

## 8.2 Example

Table 8.1 is a **main()** function that illustrates bootstrapping for both raw and scale scores for equipercentile equating with the random-groups design based on an example discussed by Kolen and Brennan (2004, pp. 50-53, 57, 60). The actual equating results for this example were reported in Section 5.2.

The intent of this example is to obtain bootstrap estimated standard errors for both raw and scale scores. This requires calling **Wrapper\_RN()** and **Wrapper\_ESS()** before calling **Wrapper\_Bootstrap()**. Note that in **Wrapper\_ESS()** **round** is specified as 1 (the '1' immediately following "yctmath.TXT") indicating that results for rounded scale scores should be provided, with rounding to the first digit before the decimal point.

The statement **long idum = -15L** at the top of the function declares the variable **idum** and assigns it an initial value of -15L. This is the beginning value of the seed that is included in the input parameters for **Wrapper\_Bootstrap()** and is used in the random number generator employed with the bootstrap. The number of bootstrap samples, **nrep**, is set to 1000 in **Wrapper\_Bootstrap()**.

The output in the tables for this example was generated using the random number function **er\_random()**, which is equivalent to the *Numerical Recipes* function **ran2()**; other random number functions and/or a different beginning seed would produce slightly different results.

### 8.2.1 Raw Scores

For raw scores, the bootstrap estimated standard errors are reported in Table 8.2. **Ave. bse** is the square root of the average of the squared standard errors, where the average is based on the new-form frequencies. The

TABLE 8.1. Main() Code to Illustrate Estimating Bootstrap Standard Errors for Equipercentile Equating with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct PDATA pdREN;
    struct ERAW_RESULTS rREN;
    struct ESS_RESULTS sREN;
    struct BOOT_ERAW_RESULTS tREN;
    struct BOOT_ESS_RESULTS uREN;

    long idum = -15L;          /* negative beginning seed */
    FILE *outf;

    outf = fopen("Chap 8 out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004): Chapter 2 example:
       Bootstrap standard errors (see also p. 238) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y",&y);

    Wrapper_RN('R','E','N",&x,&y,0,&pdREN,&rREN);
    Wrapper_ESS(&pdREN,&rREN,0,40,1,"yctmath.TXT",1,1,36,&sREN);

    Wrapper_Bootstrap(&pdREN,1000,&idum,&tREN,&uREN);
    Print_Boot_se_eraw(outf,"ACT Math---Equipercentile",
                       &pdREN,&rREN,&tREN,0);
    Print_Boot_se_ess(outf,"ACT Math---Equipercentile",
                      &pdREN,&sREN,&uREN,0);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 8.2. Estimated Bootstrap Raw Score Standard Errors for Equipercntile Equating with Random Groups Design

/\*\*\*\*\*  
 ACT Math---Equipercntile

Bootstrap standard errors for raw scores

Number of bootstrap replications = 1000

Score (x)	fd(x)	Method 0:	y-equiv
		eraw	bse
0.00000	0	0.00000	0.26730
1.00000	1	0.97956	0.71324
2.00000	1	1.64622	0.73400
3.00000	3	2.28563	0.51501
4.00000	9	2.89320	0.30875
5.00000	18	3.62047	0.18458
6.00000	59	4.49965	0.18778
7.00000	67	5.51484	0.18270
8.00000	91	6.31242	0.17589
9.00000	144	7.22424	0.16764
10.00000	149	8.16067	0.16886
11.00000	192	9.18270	0.18951
12.00000	192	10.18590	0.17818
13.00000	192	11.25130	0.22357
14.00000	201	12.38963	0.22339
15.00000	204	13.39289	0.21902
16.00000	217	14.52401	0.26080
17.00000	181	15.71690	0.26155
18.00000	184	16.82344	0.31605
19.00000	170	18.00922	0.27824
20.00000	201	19.16472	0.29061
21.00000	147	20.36760	0.31856
22.00000	163	21.45563	0.32497
23.00000	147	22.68712	0.36435
24.00000	140	23.91566	0.34635
25.00000	147	25.02916	0.30181
26.00000	126	26.16123	0.36062
27.00000	113	27.26329	0.34418
28.00000	100	28.18006	0.31524
29.00000	106	29.14243	0.33326
30.00000	107	30.13048	0.29265
31.00000	91	31.12970	0.32228
32.00000	83	32.13571	0.32580
33.00000	73	33.07807	0.30990
34.00000	72	34.01719	0.31395
35.00000	75	35.10160	0.31330
36.00000	50	36.24255	0.32071
37.00000	37	37.12476	0.27834
38.00000	38	38.13209	0.32018
39.00000	23	39.08073	0.19404
40.00000	15	39.90055	0.22617
Ave. bse			0.27716

\*\*\* means bse = 0

/\*\*\*\*\*



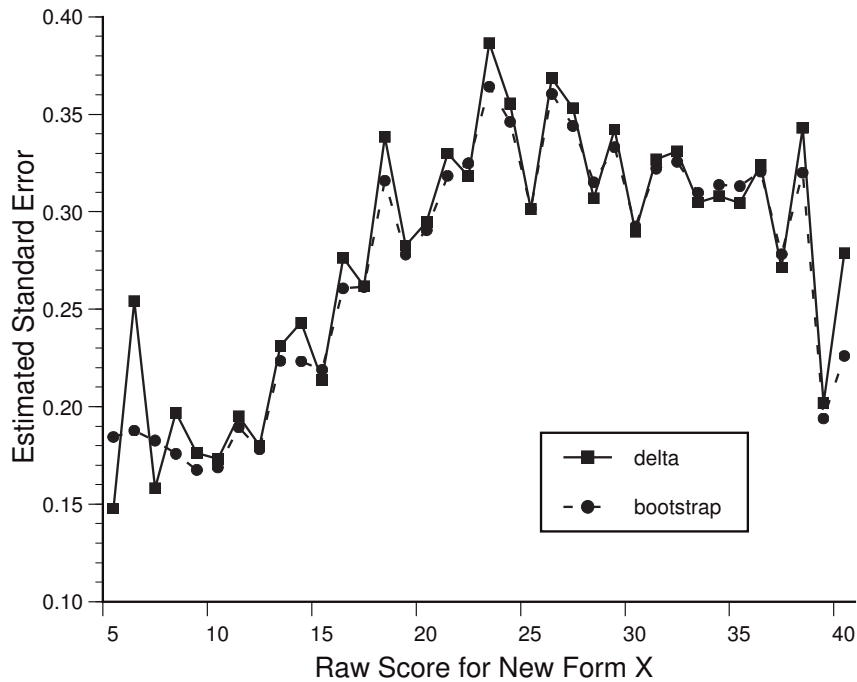


FIGURE 8.1. Estimated standard errors for delta method and bootstrap.

bootstrap estimates in Table 8.2 can be compared to the estimates in Table 7.2 for the delta method. The two sets of estimates are similar but not identical as indicated in Figure 8.1.

Recall from Chapter 5 that, for the data sets used in this example, the  $Y$  equivalent of  $x = 0$  is 0 (see Table 5.2). Also, the estimated analytic standard error for  $x = 0$  is 0 (see Table 7.2). It may seem surprising, therefore, that in Table 8.2 the bootstrap estimated standard error for  $x = 0$  is .26716. The explanation for this apparent contradiction is based on the zero-frequencies discussion in Section 5.3. In the data for  $Y$ , the frequency at  $y = 0$  is 0, which will not change for any bootstrap sample. However, by random chance, the frequency at  $y = 1$  (and subsequent values for  $y$ ) has a non-zero probability of being 0. So, at  $x = 0$ , there can be different bootstrap  $Y$  equivalents and, hence, a positive value for the estimated standard error.

### 8.2.2 Scale Scores

For unrounded and rounded scale scores, the bootstrap estimated standard errors are reported in Tables 8.3 and 8.4, respectively. In these tables an

asterick indicates that the estimated standard error is 0 because all the  $Y$  equivalents for the `nrep` = 1000 replications were the same.<sup>1</sup>

The estimated standard errors for the unrounded scale scores are somewhat “bumpy,” but they are considerably more so for the rounded scale scores, as illustrated by the following subset of results from Tables 8.3 and 8.4:

$x$	Unrounded		Rounded	
	ss	bse	ss	bse
24	22.26954	.21204	22	.32975
25	22.93531	.18058	23	.11354
26	23.61715	.21962	24	.46354

For  $x = 25$  the estimated standard error for unrounded scale scores is relatively small (.18058), but the estimated standard error is considerably smaller for rounded scale scores (.11354), which is to be expected since the  $Y$  equivalent of 22.93531 is very close to the rounded value of 23. In this case, a relatively large proportion of the rounded scale score equivalents for the bootstrap replications are likely to be 23. By contrast, for  $x = 24$  the  $Y$  equivalent (22.26954) is further from its rounded value (22), which is associated with a larger difference between the rounded and unrounded estimated standard errors (.32975 vs. .21204), and for  $x = 26$  the  $Y$  equivalent (23.61715) is even further from its rounded value (24) leading to an even larger difference between rounded and unrounded estimated standard errors (.46354 vs .21962). These results illustrate that sometimes rounding actually decreases the estimated standard error, but more often than not rounding introduces additional error, which can be substantial.

---

<sup>1</sup>The raw-to-scale score conversion table for Form Y is given in Table 3.3.

TABLE 8.3. Estimated Bootstrap Unounded Scale Score Standard Errors for Equipercentile Equating with Random Groups Design

```

/*****
ACT Math---Equipercentile

Bootstrap standard errors for unrounded scale scores

Number of bootstrap replications = 1000

Name of file containing conversion table for Y: yctmath.TXT

```

Score (x)	fd(x)	Method O: y-equiv	
		essu	bse
0.00000	0	0.50000	***
1.00000	1	0.50000	***
2.00000	1	0.50000	***
3.00000	3	0.50000	***
4.00000	9	0.50000	***
5.00000	18	0.50000	***
6.00000	59	0.59493	0.03539
7.00000	67	1.18744	0.17554
8.00000	91	2.10983	0.25282
9.00000	144	3.46449	0.26486
10.00000	149	4.92582	0.24460
11.00000	192	6.36780	0.25852
12.00000	192	7.73857	0.25258
13.00000	192	9.26220	0.32133
14.00000	201	10.84557	0.29148
15.00000	204	12.10500	0.26051
16.00000	217	13.44912	0.31050
17.00000	181	14.87383	0.31110
18.00000	184	16.15151	0.34980
19.00000	170	17.39124	0.27676
20.00000	201	18.49581	0.27550
21.00000	147	19.61506	0.28765
22.00000	163	20.55332	0.26316
23.00000	147	21.47933	0.25154
24.00000	140	22.26954	0.21204
25.00000	147	22.93531	0.18058
26.00000	126	23.61715	0.21962
27.00000	113	24.29493	0.21273
28.00000	100	24.84955	0.17716
29.00000	106	25.35377	0.15824
30.00000	107	25.78412	0.12568
31.00000	91	26.21755	0.15144
32.00000	83	26.72813	0.18131
33.00000	73	27.29077	0.19744
34.00000	72	27.92158	0.22843
35.00000	75	28.79980	0.29822
36.00000	50	30.10088	0.43185
37.00000	37	31.38695	0.41100
38.00000	38	32.89003	0.47982
39.00000	23	34.29743	0.26487
40.00000	15	35.33557	0.32342
Ave. bse			0.26331

```

-----
*** means bse = 0
/*****/

```

TABLE 8.4. Estimated Bootstrap Rounded Scale Score Standard Errors for Equipercentile Equating with Random Groups Design

```

/*****
ACT Math---Equipercentile

Bootstrap standard errors for rounded scale scores

Number of bootstrap replications = 1000

Name of file containing conversion table for Y: yctmath.TXT

```

Score (x)	fd(x)	Method O: y-equiv	
		essr	bse
0.00000	0	1	***
1.00000	1	1	***
2.00000	1	1	***
3.00000	3	1	***
4.00000	9	1	***
5.00000	18	1	***
6.00000	59	1	***
7.00000	67	1	0.12161
8.00000	91	2	0.25292
9.00000	144	3	0.49888
10.00000	149	5	0.23080
11.00000	192	6	0.46354
12.00000	192	8	0.37221
13.00000	192	9	0.44447
14.00000	201	11	0.34336
15.00000	204	12	0.27830
16.00000	217	13	0.49922
17.00000	181	15	0.35287
18.00000	184	16	0.40229
19.00000	170	17	0.48734
20.00000	201	18	0.50417
21.00000	147	20	0.47561
22.00000	163	21	0.49459
23.00000	147	21	0.49977
24.00000	140	22	0.32975
25.00000	147	23	0.11354
26.00000	126	24	0.46354
27.00000	113	24	0.37137
28.00000	100	25	0.17873
29.00000	106	25	0.35127
30.00000	107	26	0.13659
31.00000	91	26	0.16789
32.00000	83	27	0.30015
33.00000	73	27	0.34296
34.00000	72	28	0.19186
35.00000	75	29	0.38259
36.00000	50	30	0.50854
37.00000	37	31	0.54005
38.00000	38	33	0.55417
39.00000	23	34	0.39985
40.00000	15	35	0.45672
Ave. bse			0.38930

```

-----
*** means bse = 0
/****

```

# 9

## Beta-Binomial Presmoothing

This chapter and the *Equating Recipes* code for beta-binomial smoothing are both based largely on Hanson (1991b), who provides a detailed treatment of the beta-binomial model—the theory as well as method of moments estimators of the parameters. Kolen and Brennan (2004, pp. 75–77) provide a less mathematical discussion.

The phrase “beta-binomial model” is a generic description of a class of several models, the most all-encompassing of which is the four-parameter beta compound binomial model, which is the subject of Section 9.1 and the focus of Hanson (1991b). Section 9.2 discusses special cases. Section 9.3 considers characteristics of the actual and fitted observed score densities. For these sections, the discussion is with respect to Form X only; of course, an analogous discussion applies to Form Y. The applicability of the beta-binomial model to equating is treated in Section 9.4 and subsequent sections, where the smoothing is often called “presmoothing,” since smoothing is done prior to equating.

### 9.1 Four-parameter Beta Compound Binomial Model

For the four-parameter beta compound binomial model (sometimes called the beta4 method), it is assumed that the test score to be modeled is the sum of  $K$  dichotomously scored test items. This sum is often referred to as the “raw” or “observed” test score. The probability that the raw score

random variable  $X$  in the population of interest equals  $i$  ( $i = 0, \dots, K$ ) under the four-parameter beta compound binomial model is

$$\Pr(X = i) = \int_l^u \Pr(X = i | \tau, k) g(\tau | \alpha, \beta, l, u) d\tau, \quad (9.1)$$

where  $\tau$  is the proportion-correct true score and  $k$  is a variable defined later. The true-score density,  $g(\tau | \alpha, \beta, l, u)$ , is defined on the interval  $[l, u]$ , and is assumed to belong to the four parameter beta family of distributions. The density is

$$g(\tau | \alpha, \beta, l, u) = \frac{(-l + \tau)^{\alpha-1} (u - \tau)^{\beta-1}}{(u - l)^{\alpha+\beta-1} B(\alpha, \beta)}, \quad (9.2)$$

where  $B(\alpha, \beta)$  is the Beta function, which is related to the gamma function  $[\Gamma(x)]$  by

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)}. \quad (9.3)$$

The density has two shape parameters ( $\alpha > 0$  and  $\beta > 0$ ) and two limit parameters ( $l$  and  $u$  where  $0 \leq l < u \leq 1$ ).

The first term to the right of the equal sign in Equation 9.1 is the conditional error distribution,  $\Pr(X = i | \tau, k)$ . It is assumed to be Lord's (1965) two-term approximation to the compound binomial distribution:

$$\Pr(X = i | \tau, k) = p(i | K, \tau) - k\tau(1 - \tau)[p(i | K - 2, \tau) - 2p(i - 1 | K - 2, \tau) + p(i - 2 | K - 2, \tau)], \quad (9.4)$$

where

$$p(i | n, \tau) \equiv \Pr(Z = i | n, \tau)$$

with the distribution of  $Z$  being binomial with parameters  $n$  and  $\tau$ , and

$$k = \frac{K[(K - 1)(\sigma_x^2 - \sigma_e^2) - K\sigma_x^2 + \mu_x(k - \mu_x)]}{2[\mu_x(K - \mu_x) - (\sigma_x^2 - \sigma_e^2)]}. \quad (9.5)$$

To estimate  $k$  an estimate of  $\sigma_e^2$  is needed, which is usually obtained as  $\hat{\sigma}_e^2 = \hat{\sigma}_x^2(1 - \text{KR20})$ . Clearly, given an estimate of reliability, say KR20,  $k$  can be estimated, and Equation 9.4 can be used to approximate the compound binomial error distribution for any proportion-correct true score.

## 9.2 Special Cases of the Four-parameter Beta Compound Binomial Model

Sometimes the method of moments estimation procedure can fit only three parameters, resulting in a three-parameter beta compound binomial model. This is one special case, but not the most frequently discussed case.

A second special case occurs when the binomial error model replaces the compound binomial error model, resulting in the three- or four-parameter beta binomial error model. In this case,  $\Pr(X = i | \tau, k)$  in Equation 9.4 is replaced by

$$\Pr(X = i | \tau) = \binom{K}{i} \tau^i (1 - \tau)^{(K-i)}. \quad (9.6)$$

Hanson and Brennan (1990), among others, have found that often the two error models give very similar results with real data, suggesting that the more complex two-term approximation to the compound binomial model may not be worth the added complexity in most cases.

The final special case is the two-parameter beta-binomial model in which  $l$  and  $u$  are set to 0 and 1, respectively, and the error model is given by Equation 9.6. This is a relatively frequently discussed model that was introduced by Keats and Lord (1962) and popularized by Huynh (1976) who provided an elegant recursive algorithm for computing  $\Pr(X = i)$ , which is the negative hypergeometric distribution. Many authors, however, including Hanson and Brennan (1990) and Kolen and Brennan (2004), suggest that often the two-parameter beta-binomial model is not flexible enough to fit real data very well.

### 9.3 Actual and Fitted Observed Score Densities

When scores for Form X are presmoothed using any one of the versions of the beta-binomial model, then  $\Pr(X = i)$  (e.g., Equation 9.1) is a smoothed version of the actual observed score density. Stated differently,  $\Pr(X = i)$  is the expected observed score density for the population.

In addition, beta-binomial smoothing using method-of-moments estimates has a moments preservation property. Specifically, the first  $m$  moments of the actual and expected observed-score densities are identical, where  $m$  is the number of parameters fit for the true-score distribution.

### 9.4 The Beta-binomial Model Applied to Equating

In the context of equating, presmoothing means that that observed score distributions are smoothed prior to equating. Almost always beta-binomial presmoothing is restricted to the random groups design, because there is no currently available beta-binomial smoothing procedure for bivariate distributions.<sup>1</sup>

---

<sup>1</sup>Beta-binomial presmoothing might be used with the single group design, but the authors are unaware of any such use reported in the literature.

Because of the moments preservation property, linear equating results using beta-binomial presmoothing are identical to results using the actual data. Equipercntile results will generally differ, however, since the largest number of preserved moments under the beta-binomial model is four.

## 9.5 Functions

The functions for beta-binomial presmoothing are in `BetaBinomial.c`, with the function prototypes in `BetaBinomial.h`. The wrapper function for beta-binomial presmoothing is `Wrapper_Smooth_BB()`, which must be called for both Forms X and Y before calling `Wrapper_RB()`, which is the wrapper function for equipercntile equating with the random groups design under beta-binomial smoothing.

### 9.5.1 `Wrapper_Smooth_BB()` and `Print_BB()`

The wrapper function for beta-binomial smoothing is:

```
void Wrapper_Smooth_BB(struct USTATS *x, int nparm, double rel,
                      struct BB_SMOOTH *s)
```

The input variables are:

- ▷ `x` = a `USTATS` structure;
- ▷ `nparm` = number of parameters for beta (2 or 4); if `nparm` = 4, but *Equating Recipes* cannot fit four parameters, then it will fit three, if possible; and;
- ▷ `rel` = reliability (usually KR20); if `rel` = 0, then the binomial error model is used; otherwise, the two-term approximation to the compound binomial error model is used.

The output variable is:

- ▷ `s` = a `BB_SMOOTH` structure that contains the beta-binomial smoothing results (see `ERutilities.h` for the definition of `BB_SMOOTH`).

The associated print function is:

```
void Print_BB(FILE *fp, char tt[], struct USTATS *x,
              struct BB_SMOOTH *s)
```

The input variables are:

- ▷ `fp` = output file pointer;



- ▷ `tt[]` = user-specified title;
- ▷ `x` = USTATS structure; and
- ▷ `s` = BB\_SMOOTH structure;

### 9.5.2 Wrapper\_RB()

`Wrapper_Smooth_BB()` must be called for both Forms X and Y before calling `Wrapper_RB()`, which is the wrapper function for equipercntile equating with the random groups design under beta-binomial smoothing:

```
void Wrapper_RB(char design, char method, char smoothing,
                struct USTATS *x, struct USTATS *y,
                struct BB_SMOOTH *bbx, struct BB_SMOOTH *bby, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' designating the random groups design;
- ▷ `method` = 'E' designating equipercntile equating;
- ▷ `smoothing` = 'B' designating beta-binomial smoothing;
- ▷ `x` = USTATS structure for the new form;
- ▷ `y` = USTATS structure for the old form;
- ▷ `bbx` = BB\_SMOOTH structure for the new form;
- ▷ `bby` = BB\_SMOOTH structure for the old form; and
- ▷ `rep` = replication number (must be set to 0 for actual equating; used to count replications when bootstrap standard errors are estimated).

The output variables are:

- ▷ `inall` = a PDATA structure (selected elements are populated by `Wrapper_RB()`); and
- ▷ `r` = an ERAW\_RESULTS structure (elements are populated by `Wrapper_RB()`).

`Wrapper_RB()` calls `EquiEquate()` to perform the equating with the smoothed densities, and it calls `MomentsFromFD()` to get the moments relative to the new-form frequencies. `ERutilities.c` contains the last two functions.

The associated print function for `Wrapper_RB()` is:

```
void Print_RB(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `fp` = output file pointer;
- ▷ `tt[]` = user-specified title;
- ▷ `inall` = the `PDATA` structure associated with the beta-binomial presmoothed equating;
- ▷ `r` = the `ERAW_RESULTS` structure that contains the beta-binomial presmoothed equated raw scores;

## 9.6 Code for `Wrapper_Bootstrap()`

Chapter 8 (page 84) provided a list of four general steps for enabling `Wrapper_Bootstrap()` to perform bootstrapping for a new method. For equipercntile equating with beta-binomial presmoothing, these four steps involved the following specific additions of code:

1. adding `struct BB_SMOOTH *bbx` and `struct BB_SMOOTH *bby` to the definition of `struct PDATA` in `ERutilities.h`;
2. adding `inall->bbx = bbx` and `inall->bby = bby` within the `if(inall->rep == 0){ }` code in `Wrapper_RB()`;
3. adding `struct BB_SMOOTH bt_bbx, bt_bby` at the beginning of `Wrapper_Bootstrap()`; and
4. adding the following code between the `start` and `end` comments in `Wrapper_Bootstrap()`:

```
else if(inall->design == 'R' && inall->smoothing == 'B'){

    Parametric_boot_univ_BB(inall->bbx, idum, rep, &bt_bbx);
    Parametric_boot_univ_BB(inall->bby, idum, rep, &bt_bby);
    Wrapper_RB('R','E','B',inall->x, inall->y, &bt_bbx, &bt_bby,
               rep, inall, &br);
}
```

## 9.7 Example

Recall, again, the equipercntile equating example for the random-groups design based discussed by Kolen and Brennan (2004, pp. 50-53, 57, 60). The

TABLE 9.1. Main() Code to Illustrate Beta-binomial Presmoothing for Equipercenile Equating with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct PDATA pddb;
    struct ERAW_RESULTS rbb;
    struct ESS_RESULTS sbb;
    struct BOOT_ERAW_RESULTS tbb;
    struct BOOT_ESS_RESULTS ubb;
    struct BB_SMOOTH bbx,bby;

    long idum = -15L;                      /* beginning seed */
    FILE *outf;

    outf = fopen("Chap 9 out","w");

    /* Random Groups Design: Kolen and Brennan (2004)
       Chapter 2 example: Equipercenile equating with
       4 parm beta binomial smoothing
       (see pages 79, 81-85) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X',&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y',&y);

    Wrapper_Smooth_BB(&x, 4, 0, &bbx);
    Print_BB(outf,"beta4 (binominal) for ACT Math X", &x, &bbx);

    Wrapper_Smooth_BB(&y, 4, 0, &bby);
    Print_BB(outf,"beta4 (binominal) for ACT Math Y", &y, &bby);

    Wrapper_RB('R','E','B', &x, &y, &bbx, &bby, 0, &pddb, &rbb);
    Print_RB(outf,"ACT Math---Equi---beta bin",&pddb, &rbb);

    Wrapper_ESS(&pddb,&rbb,0,40,1,"yctmath.TXT",1,1,36,&sbb);
    Print_ESS(outf,"ACT Math---Equi---beta bin",&pddb,&sbb);

    Wrapper_Bootstrap(&pddb,1000,&idum,&tbb,&ubb);
    Print_Boot_se_eraf(outf,"ACT Math---Equi---beta bin",&pddb,
        &rbb,&tbb,0);
    Print_Boot_se_ess(outf,"ACT Math---Equi---beta bin",&pddb,
        &sbb,&ubb,0);

    fclose(outf);
    return 0;
}
*****/

```

actual equating results for unsmoothed raw scores and scales scores based on unsmoothed raw scores were reported in Chapter 5 (see Section 5.2). For the same example, Table 9.1 is a `main()` function that provides code for: (a) beta-binomial smoothing for raw scores for Forms X and Y; (b) equipercentile equating using the beta-binomial presmoothed raw scores; (c) equated scale scores (both unrounded and rounded) based on (b); and (d) bootstrap estimated standard errors for (b) and (c). Kolen and Brennan (2004, pp. 79, 81–85) provide a discussion of some of these results.

### 9.7.1 Presmoothing

After reading in the data for both forms using `ReadFdGet_USTATS()`, `Wrapper_Smooth_BB()` is called to perform beta-binomial presmoothing with the results stored in the `BB_SMOOTH` structures `bbx` and `bby`, which are declared at the top of `main()`. Since `nparm = 4` and `rel = 0` for both Forms X and Y, the model to be fit for both forms is the four-parameter beta with binomial error. Tables 9.2 and 9.3 provide the output based on using `Print_BB()`. Note that for Form X, only three moments could be fit with the upper limit  $l$  set to 1.

### 9.7.2 Equipercentile Equating

`Wrapper_RB()` performs equipercentile equating of the beta-binomial presmoothed raw scores using the results in `bbx` and `bby`, with the equated raw scores and their moments stored in the `ERAW_RESULTS` structure `rbb` declared at the top of `main()`. Then `Wrapper_ESS()` uses the equated raw scores in `rbb` to get equated scale scores (both unrounded and rounded), with the results stored in the `ESS_RESULTS` structure `sbb` declared at the top of `main()`.

Using `Print_RB`, the beta-binomial smoothed raw-score output is in Table 9.4, which can be compared with the unsmoothed raw-score equating in Table 5.2. Similarly, the scale score results (based on the beta-binomial smoothed raw-score results) in Tables 9.5 and 9.6 can be compared with Tables 5.3 and 5.4, respectively.<sup>2</sup>

### 9.7.3 Bootstrap Standard Errors

Using the structures `tbb` and `sbb` as input, `Wrapper_Bootstrap()` calls functions that perform parametric bootstrapping under the beta-binomial model. The resulting smoothed raw-score, unrounded scale-score, and rounded scale-score output is in Tables 9.7, 9.8, and 9.9. These results can be compared with the corresponding unsmoothed results in Tables 8.2, 8.3, and 8.4, respectively.

---

<sup>2</sup>The raw-to-scale score conversion table for Form Y is given in Table 3.3 on 43.

TABLE 9.2. Output for Beta-binomial Smoothing for Form X  
 /\*\*\*\*\*  
 beta4 (binominal) for ACT Math X; Variable identifier = X

Input filename: actmathfreq.dat

Beta-Binomial Smoothing: 4-parameter beta with binomial error model

Number of examinees: 4329  
 Number of items: 40  
 Reliability (usually KR20): 0.00000  
 Lord's k: 0.00000  
 Number of moments fit: 3

\*\*\*Parameter Estimates for Beta Distribution\*\*\*

alpha: 0.99046  
 beta: 1.80049  
 lower limit: 0.21923  
 upper limit: 1.00000

\*\*\*Moments\*\*\*

	Raw	Fitted-Raw	True
Mean	19.85239	19.85239	19.85239
S.D.	8.21164	8.21164	7.67490
Skew	0.37527	0.37527	0.49302
Kurt	2.30244	2.28064	2.29889

\*\*\*Chi-Square Statistics for Fitted Distribution\*\*\*

Likelihood ratio: 33.97183  
 Pearson: 39.06521

\*\*\*Frequencies and Proportions\*\*\*

Score	freq	fitted-freq	prop	fitted-prop	fitted-crfd	fitted-prd
0	0	0.00963	0.00000	0.00000	0.00000	0.00011
1	1	0.12014	0.00023	0.00003	0.00003	0.00161
2	1	0.73832	0.00023	0.00017	0.00020	0.01153
3	3	2.98423	0.00069	0.00069	0.00089	0.05452
4	9	8.94014	0.00208	0.00207	0.00296	0.19225
5	18	21.22330	0.00416	0.00490	0.00786	0.54064
6	59	41.71431	0.01363	0.00964	0.01749	1.26757
7	67	70.09792	0.01548	0.01619	0.03369	2.55900
8	91	103.32457	0.02102	0.02387	0.05755	4.56203
9	144	136.55651	0.03326	0.03154	0.08910	7.33266
10	149	165.05569	0.03442	0.03813	0.12723	10.81628
11	192	185.83742	0.04435	0.04293	0.17016	14.86910
12	192	198.24463	0.04435	0.04579	0.21595	19.30526
13	192	203.44108	0.04435	0.04699	0.26294	23.94473
14	201	203.41699	0.04643	0.04699	0.30993	28.64395
15	204	200.12573	0.04712	0.04623	0.35616	33.30487
16	217	195.03753	0.05013	0.04505	0.40122	37.86901
17	181	189.06891	0.04181	0.04367	0.44489	42.30544
18	184	182.70803	0.04250	0.04221	0.48710	46.59947
19	170	176.17947	0.03927	0.04070	0.52780	50.74463
20	201	169.57157	0.04643	0.03917	0.56697	54.73805
21	147	162.91201	0.03396	0.03763	0.60460	58.57824
22	163	156.20498	0.03765	0.03608	0.64068	62.26405
23	147	149.44666	0.03396	0.03452	0.67520	65.79433
24	140	142.63082	0.03234	0.03295	0.70815	69.16783
25	147	135.75041	0.03396	0.03136	0.73951	72.38313
26	126	128.79787	0.02911	0.02975	0.76926	75.43867
27	113	121.76499	0.02610	0.02813	0.79739	78.33267
28	100	114.64268	0.02310	0.02648	0.82387	81.06318
29	106	107.42068	0.02449	0.02481	0.84869	83.62802
30	107	100.08709	0.02472	0.02312	0.87181	86.02474
31	91	92.62788	0.02102	0.02140	0.89320	88.25060
32	83	85.02610	0.01917	0.01964	0.91285	90.30250
33	73	77.26079	0.01686	0.01785	0.93069	92.17692
34	72	69.30527	0.01663	0.01601	0.94670	93.86976
35	75	61.12445	0.01733	0.01412	0.96082	95.37622
36	50	52.67010	0.01155	0.01217	0.97299	96.69055
37	37	43.87206	0.00855	0.01013	0.98312	97.80561
38	38	34.61979	0.00878	0.00800	0.99112	98.71219
39	23	24.71610	0.00531	0.00571	0.99683	99.39752
40	15	13.72313	0.00347	0.00317	1.00000	99.84150

/\*\*\*\*\*

TABLE 9.3. Output for Beta-binomial Smoothing for Form Y

/\*\*\*\*\*  
 beta4 (binominal) for ACT Math Y; Variable identifier = Y  
 \*\*\*\*\*/

Input filename: actmathfreq.dat

Beta-Binomial Smoothing: 4-parameter beta with binomial error model

Number of examinees: 4152  
 Number of items: 40  
 Reliability (usually KR20): 0.00000  
 Lord's k: 0.00000  
 Number of momemts fit: 4

\*\*\*Parameter Estimates for Beta Distribution\*\*\*

alpha: 0.89542  
 beta: 1.48248  
 lower limit: 0.17206  
 upper limit: 0.97522

\*\*\*Moments\*\*\*

	Raw	Fitted-Raw	True
Mean	18.97977	18.97977	18.97977
S.D.	8.93932	8.93932	8.46940
Skew	0.35269	0.35269	0.42782
Kurt	2.14636	2.14636	2.10781

\*\*\*Chi-Square Statistics for Fitted Distribution\*\*\*

Likelihood ratio: 31.63588  
 Pearson: 31.74158

\*\*\*Frequencies and Proportions\*\*\*

Score	freq	fitted-freq	prop	fitted-prop	fitted-crfd	fitted-prd
0	0	0.11026	0.00000	0.00003	0.00003	0.00133
1	1	1.03432	0.00024	0.00025	0.00028	0.01511
2	3	4.80181	0.00072	0.00116	0.00143	0.08539
3	13	14.75002	0.00313	0.00355	0.00498	0.32084
4	42	33.85002	0.01012	0.00815	0.01314	0.90610
5	59	62.20688	0.01421	0.01498	0.02812	2.06286
6	95	95.96170	0.02288	0.02311	0.05123	3.96759
7	131	128.83403	0.03155	0.03103	0.08226	6.67466
8	158	155.15610	0.03805	0.03737	0.11963	10.09458
9	161	172.13757	0.03878	0.04146	0.16109	14.03598
10	194	180.13891	0.04672	0.04339	0.20448	18.27823
11	164	181.43268	0.03950	0.04370	0.24817	22.63242
12	166	178.68210	0.03998	0.04304	0.29121	26.96906
13	197	173.98645	0.04745	0.04190	0.33311	31.21603
14	177	168.62501	0.04263	0.04061	0.37373	35.34189
15	158	163.21653	0.03805	0.03931	0.41304	39.33806
16	169	157.98522	0.04070	0.03805	0.45109	43.20609
17	132	152.96974	0.03179	0.03684	0.48793	46.95073
18	158	148.14162	0.03805	0.03568	0.52361	50.57683
19	151	143.45762	0.03637	0.03455	0.55816	54.08839
20	134	138.87733	0.03227	0.03345	0.59161	57.48837
21	137	134.36673	0.03300	0.03236	0.62397	60.77889
22	122	129.89726	0.02938	0.03129	0.65526	63.96125
23	110	125.44418	0.02649	0.03021	0.68547	67.03618
24	116	120.98510	0.02794	0.02914	0.71461	70.00377
25	132	116.49879	0.03179	0.02806	0.74267	72.86365
26	104	111.96417	0.02505	0.02697	0.76963	75.61489
27	104	107.35940	0.02505	0.02586	0.79549	78.25607
28	114	102.66086	0.02746	0.02473	0.82021	80.78521
29	97	97.84195	0.02336	0.02357	0.84378	83.19974
30	107	92.87152	0.02577	0.02237	0.86615	85.49639
31	88	87.71161	0.02119	0.02113	0.88727	87.67104
32	80	82.31394	0.01927	0.01983	0.90710	89.71856
33	79	76.61407	0.01903	0.01845	0.92555	91.63243
34	70	70.52031	0.01686	0.01698	0.94254	93.40428
35	61	63.89059	0.01469	0.01539	0.95792	95.02291
36	48	56.47953	0.01156	0.01360	0.97153	96.47245
37	47	47.83049	0.01132	0.01152	0.98305	97.72859
38	29	37.17274	0.00698	0.00895	0.99200	98.75224
39	32	23.84080	0.00771	0.00574	0.99774	99.48698
40	12	9.38003	0.00289	0.00226	1.00000	99.88704

/\*\*\*\*\*

TABLE 9.4. Output for Equated Raw Scores using Beta-binomial Smoothing

```

/*****
ACT Math---Equi---beta bin
Equipercntile Equating with Random Groups Design:

```

```

Beta-Binomial Smoothing for X (new form):
  4-parameter beta with binomial error model
Beta-Binomial Smoothing for Y (old form):
  4-parameter beta with binomial error model
Input file for X: actmathfreq.dat
Input file for Y: actmathfreq.dat
-----

```

## Equated Raw Scores

Raw Score (X)	equiv
0.00000	-0.45811
1.00000	0.10629
2.00000	0.85605
3.00000	1.73306
4.00000	2.63801
5.00000	3.55172
6.00000	4.44336
7.00000	5.33115
8.00000	6.25720
9.00000	7.21206
10.00000	8.19313
11.00000	9.20095
12.00000	10.23672
13.00000	11.30032
14.00000	12.38919
15.00000	13.49848
16.00000	14.62629
17.00000	15.76330
18.00000	16.90466
19.00000	18.04703
20.00000	19.18803
21.00000	20.32584
22.00000	21.45892
23.00000	22.58897
24.00000	23.71312
25.00000	24.82875
26.00000	25.93465
27.00000	27.02963
28.00000	28.11242
29.00000	29.18174
30.00000	30.23621
31.00000	31.27434
32.00000	32.29455
33.00000	33.29508
34.00000	34.27406
35.00000	35.22960
36.00000	36.16033
37.00000	37.06686
38.00000	37.95528
39.00000	38.84420
40.00000	39.79840
Mean	18.98054
S.D.	8.93068
Skew	0.35560
Kurt	2.16648

```

/*****

```

TABLE 9.5. Output for Unrounded Equated Scale Scores using Beta-binomial Smoothing

```

/*****
ACT Math---Equi---beta bin

Name of file containing yct[] []: yctmath.TXT

Equated Scale Scores (unrounded)

Raw Score (x)      Method 0:
                   y-equiv

    0.00000      0.50000
    1.00000      0.50000
    2.00000      0.50000
    3.00000      0.50000
    4.00000      0.50000
    5.00000      0.50000
    6.00000      0.58424
    7.00000      1.00996
    8.00000      2.02965
    9.00000      3.44514
   10.00000      4.97204
   11.00000      6.39249
   12.00000      7.81111
   13.00000      9.33271
   14.00000     10.84499
   15.00000     12.23029
   16.00000     13.57088
   17.00000     14.92937
   18.00000     16.24410
   19.00000     17.42745
   20.00000     18.51782
   21.00000     19.57749
   22.00000     20.55599
   23.00000     21.41010
   24.00000     22.14356
   25.00000     22.81577
   26.00000     23.47906
   27.00000     24.14980
   28.00000     24.81310
   29.00000     25.37096
   30.00000     25.82902
   31.00000     26.28909
   32.00000     26.82193
   33.00000     27.43608
   34.00000     28.12307
   35.00000     28.93499
   36.00000     29.98146
   37.00000     31.30064
   38.00000     32.62474
   39.00000     33.96091
   40.00000     35.20622

      Mean      16.52304
      S.D.       8.35542
      Skew       -0.14115
      Kurt        2.06285
*****/

```



TABLE 9.6. Output for Rounded Equated Scale Scores using Beta-binomial Smoothing

```

/*****/
ACT Math---Equi---beta bin

Name of file containing yct[] []: yctmath.TXT

Equated Scale Scores (rounded)

Raw Score (x)      Method 0:
                   y-equiv

    0.00000         1
    1.00000         1
    2.00000         1
    3.00000         1
    4.00000         1
    5.00000         1
    6.00000         1
    7.00000         1
    8.00000         2
    9.00000         3
   10.00000         5
   11.00000         6
   12.00000         8
   13.00000         9
   14.00000        11
   15.00000        12
   16.00000        14
   17.00000        15
   18.00000        16
   19.00000        17
   20.00000        19
   21.00000        20
   22.00000        21
   23.00000        21
   24.00000        22
   25.00000        23
   26.00000        23
   27.00000        24
   28.00000        25
   29.00000        25
   30.00000        26
   31.00000        26
   32.00000        27
   33.00000        27
   34.00000        28
   35.00000        29
   36.00000        30
   37.00000        31
   38.00000        33
   39.00000        34
   40.00000        35

      Mean      16.49988
      S.D.      8.36636
      Skew      -0.15091
      Kurt       2.05489
/*****/

```

TABLE 9.7. Output for Bootstrap Estimates of Standard Errors for Equated Raw Scores using Beta-binomial Smoothing

/\*\*\*\*\*  
 ACT Math---Equi---beta bin

Bootstrap standard errors for raw scores

Number of bootstrap replications = 1000

Score (x)	fd(x)	Method 0:	y-equiv
		eraw	bse
0.00000	0	-0.45811	0.32836
1.00000	1	0.10629	0.48037
2.00000	1	0.85605	0.71402
3.00000	3	1.73306	0.55058
4.00000	9	2.63801	0.32922
5.00000	18	3.55172	0.24326
6.00000	59	4.44336	0.21216
7.00000	67	5.33115	0.18953
8.00000	91	6.25720	0.17424
9.00000	144	7.21206	0.17005
10.00000	149	8.19313	0.17100
11.00000	192	9.20095	0.17926
12.00000	192	10.23672	0.19148
13.00000	192	11.30032	0.20386
14.00000	201	12.38919	0.22320
15.00000	204	13.49848	0.24563
16.00000	217	14.62629	0.26446
17.00000	181	15.76330	0.27382
18.00000	184	16.90466	0.28208
19.00000	170	18.04703	0.29454
20.00000	201	19.18803	0.30394
21.00000	147	20.32584	0.31239
22.00000	163	21.45892	0.31861
23.00000	147	22.58897	0.32544
24.00000	140	23.71312	0.32916
25.00000	147	24.82875	0.33523
26.00000	126	25.93465	0.33858
27.00000	113	27.02963	0.34122
28.00000	100	28.11242	0.33942
29.00000	106	29.18174	0.33749
30.00000	107	30.23621	0.33256
31.00000	91	31.27434	0.32477
32.00000	83	32.29455	0.31968
33.00000	73	33.29508	0.31698
34.00000	72	34.27406	0.31050
35.00000	75	35.22960	0.29815
36.00000	50	36.16033	0.28087
37.00000	37	37.06686	0.26588
38.00000	38	37.95528	0.25207
39.00000	23	38.84420	0.24806
40.00000	15	39.79840	0.24920
Ave. bse			0.27684

\*\*\* means bse = 0

/\*\*\*\*\*

TABLE 9.8. Output for Bootstrap Estimates of Standard Errors for Unrounded Equated Scale Scores using Beta-binomial Smoothing

```

/*****
ACT Math---Equi---beta bin

Bootstrap standard errors for unrounded scale scores

Number of bootstrap replications = 1000

Name of file containing conversion table for Y: yctmath.TXT

      Method 0:      y-equiv
-----
Score (x)  fd(x)      essu      bse

  0.00000    0      0.50000      ***
  1.00000    1      0.50000      ***
  2.00000    1      0.50000      ***
  3.00000    3      0.50000      ***
  4.00000    9      0.50000      ***
  5.00000   18      0.50000      ***
  6.00000   59      0.58424      0.03882
  7.00000   67      1.00996      0.17776
  8.00000   91      2.02965      0.24857
  9.00000  144      3.44514      0.26830
 10.00000  149      4.97204      0.24657
 11.00000  192      6.39249      0.24401
 12.00000  192      7.81111      0.27193
 13.00000  192      9.33271      0.29314
 14.00000  201     10.84499      0.29083
 15.00000  204     12.23029      0.29198
 16.00000  217     13.57088      0.31493
 17.00000  181     14.92937      0.32474
 18.00000  184     16.24410      0.31029
 19.00000  170     17.42745      0.29157
 20.00000  201     18.51782      0.28808
 21.00000  147     19.57749      0.28247
 22.00000  163     20.55599      0.25867
 23.00000  147     21.41010      0.22771
 24.00000  140     22.14356      0.20345
 25.00000  147     22.81577      0.20024
 26.00000  126     23.47906      0.20511
 27.00000  113     24.14980      0.21053
 28.00000  100     24.81310      0.19355
 29.00000  106     25.37096      0.15853
 30.00000  107     25.82902      0.14274
 31.00000   91     26.28909      0.15618
 32.00000   83     26.82193      0.18336
 33.00000   73     27.43608      0.20829
 34.00000   72     28.12307      0.23717
 35.00000   75     28.93499      0.29607
 36.00000   50     29.98146      0.37268
 37.00000   37     31.30064      0.39177
 38.00000   38     32.62474      0.37700
 39.00000   23     33.96091      0.36029
 40.00000   15     35.20622      0.34268

Ave. bse                                0.26133
*****/

```

TABLE 9.9. Output for Bootstrap Estimates of Standard Errors for Rounded Equated Scale Scores using Beta-binomial Smoothing

```

/*****
ACT Math---Equi---beta bin

Bootstrap standard errors for rounded scale scores

Number of bootstrap replications = 1000

Name of file containing conversion table for Y: yctmath.TXT

Method 0:      y-equiv
-----
Score (x)  fd(x)      essr      bse

0.00000    0          1          ***
1.00000    1          1          ***
2.00000    1          1          ***
3.00000    3          1          ***
4.00000    9          1          ***
5.00000   18          1          ***
6.00000   59          1          ***
7.00000   67          1          ***
8.00000   91          2    0.18522
9.00000  144          3    0.49533
10.00000  149          5    0.21425
11.00000  192          6    0.47495
12.00000  192          8    0.33633
13.00000  192          9    0.46354
14.00000  201         11    0.33301
15.00000  204         12    0.39385
16.00000  217         14    0.49903
17.00000  181         15    0.37193
18.00000  184         16    0.41924
19.00000  170         17    0.49850
20.00000  201         19    0.50272
21.00000  147         20    0.48754
22.00000  163         21    0.49444
23.00000  147         21    0.47561
24.00000  140         22    0.18638
25.00000  147         23    0.23004
26.00000  126         23    0.49518
27.00000  113         24    0.20296
28.00000  100         25    0.24308
29.00000  106         25    0.37930
30.00000  107         26    0.13659
31.00000   91         26    0.26987
32.00000   83         27    0.20337
33.00000   73         27    0.48331
34.00000   72         28    0.22589
35.00000   75         29    0.29725
36.00000   50         30    0.41463
37.00000   37         31    0.51593
38.00000   38         33    0.50320
39.00000   23         34    0.42208
40.00000   15         35    0.41861

Ave. bse                                0.39092
-----
*** means bse = 0
*****/

```

# 10

## Log-Linear Presmoothing

This chapter and the *Equating Recipes* code for log-linear smoothing are both based largely on Holland and Thayer (1987, 2000), although the treatment here is not as extensive as theirs. Kolen and Brennan (2004, pp. 74–75) provide a brief, less mathematical discussion. In equating contexts log-linear smoothing is often called “presmoothing,” since the smoothing is done prior to equating.

The first section in this chapter discusses log-linear smoothing in general. The next two sections consider log-linear smoothing in the specific contexts of univariate and bivariate distributions, respectively. Then functions are discussed for:

- smoothing univariate distributions;
- smoothing bivariate distributions;
- equating with the random-groups design when the two univariate distributions are presmoothed;
- equating with the single-group design when the bivariate distribution is presmoothed; and
- equating with the common-item nonequivalent groups design when the two bivariate distributions are presmoothed.

Finally, two examples are provided.

### 10.1 General Discussion of Log-linear Smoothing

There are at least two statistical approaches to log-linear smoothing. One uses the multinomial model, and the other uses the Poisson model. The resulting estimates

are asymptotically equal. Here the multinomial model is used.<sup>1</sup> The discussion provided next closely mirrors that in Holland and Thayer (1987, 2000), although there are some occasional notational differences.<sup>2</sup>

### 10.1.1 Model

Let  $\mathbf{n}$  be a column vector of frequencies for  $T$  score categories with entries  $n_1, n_2, \dots, n_T$ . The total frequency is

$$N = \sum_{i=1}^T n_i.$$

We assume that  $\mathbf{n}$  has a multinomial distribution:

$$\text{Prob}(\mathbf{n}) = \frac{N!}{n_1! \cdots n_T!} \prod_i p_i^{n_i}. \quad (10.1)$$

It follows that the log-likelihood function is proportional to:

$$L = \sum_i n_i \log p_i, \quad (10.2)$$

where  $p_i > 0$  and  $\sum_i p_i = 1$ .

The vector  $\mathbf{p}$  satisfies a log-linear model if

$$\log(p_i) = \alpha + \mu_i + \mathbf{b}_i \boldsymbol{\beta}, \quad (10.3)$$

where  $\mathbf{b}_i$  is a row vector of  $C$  known constants,  $\boldsymbol{\beta}$  is a column vector of  $C$  free parameters,  $\mu_i$  is a known constant, and  $\alpha$  is a normalizing constant selected to make  $\sum_i p_i = 1$ , i.e.,

$$\alpha = -\log \left[ \sum_i \exp(\mu_i + \mathbf{b}_i \boldsymbol{\beta}) \right]. \quad (10.4)$$

The  $n_i$  are the actual or observed frequencies. It is often convenient to express log-linear models in terms of the expected values of the  $n_i$ , which we designate  $m_i$ . Specifically,

$$\begin{aligned} \log(m_i) &= \log(Np_i) \\ &= \log N + \log p_i \\ &= \alpha' + \mu_i + \mathbf{b}_i \boldsymbol{\beta}, \end{aligned} \quad (10.5)$$

---

<sup>1</sup>To the senior author, in equating contexts, the multinomial model seems more natural than the Poisson model.

<sup>2</sup>Here, boldface uppercase Roman letters and the boldface Greek letter beta designate matrices, boldface lowercase Roman letters designate vectors, italicized lowercase Roman letters designate elements of vectors or matrices,  $i$  is used throughout to index score categories, and  $c = 1, 2, \dots, C$  indexes degrees of power.

where  $\alpha'$  is a normalizing constant selected to make  $\sum_i m_i = N$ , i.e.,

$$\alpha' = \log N - \log \left[ \sum_i \exp(\mu_i + \mathbf{b}_i \boldsymbol{\beta}) \right]. \quad (10.6)$$

The model Equations 10.3 and 10.5 can be specified more compactly as

$$\log(\mathbf{p}) = \alpha + \boldsymbol{\mu} + \mathbf{B}\boldsymbol{\beta} \quad (10.7)$$

and

$$\log(\mathbf{m}) = \alpha' + \boldsymbol{\mu} + \mathbf{B}\boldsymbol{\beta}, \quad (10.8)$$

where  $\mathbf{B}$  is a  $T \times C$  matrix whose rows are the  $\mathbf{b}_i$ .  $\mathbf{B}$  is often called the design matrix.

### 10.1.2 Maximum Likelihood Estimation

Maximum likelihood estimation proceeds by maximizing the log-likelihood function in Equation 10.2. This is accomplished by differentiating  $L$  and solving the resulting likelihood equations for  $\boldsymbol{\beta}$ , i.e., solving

$$\frac{\partial L}{\partial \boldsymbol{\beta}} = 0 \quad (10.9)$$

for  $\boldsymbol{\beta}$ . The solution,  $\hat{\boldsymbol{\beta}}$ , is the maximum likelihood estimate (mle) of  $\boldsymbol{\beta}$ . Holland and Thayer (1987) show that

$$\frac{\partial L}{\partial \boldsymbol{\beta}} = \mathbf{B}^t(\mathbf{n} - \mathbf{m}). \quad (10.10)$$

Replacing this result in Equation 10.9 gives

$$\mathbf{B}^t \mathbf{n} = \mathbf{B}^t \mathbf{m}, \quad (10.11)$$

where  $t$  indicates the transpose operation. In other words, the likelihood equations are solved when the  $C$  moments of the observed score distribution equal the  $C$  moments of the log-linear smoothed distribution. In this sense, Equation 10.11 is the so-called “moments matching” or “moments preservation” property of the mle’s.

The asymptotic covariance matrix of  $\hat{\boldsymbol{\beta}}$  is

$$\text{Cov}(\hat{\boldsymbol{\beta}}) = \left( -\frac{\partial^2 L}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}} \right)^{-1}. \quad (10.12)$$

Holland and Thayer (1987) show that

$$\frac{\partial^2 L}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}} = -(\mathbf{B}^t \boldsymbol{\Sigma}_{\mathbf{m}} \mathbf{B}), \quad (10.13)$$

where

$$\boldsymbol{\Sigma}_{\mathbf{m}} = \mathbf{D}_{\mathbf{m}} - \frac{\mathbf{m} \mathbf{m}^t}{N},$$

with  $\mathbf{D}_{\mathbf{m}}$  being the diagonal matrix with elements  $m_i$  that satisfy Equation 10.5.

### 10.1.3 Newton-Raphson Method

The Newton-Raphson method is an algorithm designed to find the roots of non-linear equations. It is briefly described by Kolen and Brennan (2004, p. 177). This method is also often used to solve certain types of maximization problems with non-linear equations. For such applications (and this is one of them) the Newton-Raphson method makes use of the fact that the roots of the first derivatives give the maximum values (assuming certain regularity conditions). Holland and Thayer (1987) provide a general mathematical discussion, as well as the following algorithm for solving the specific likelihood equations given by Equation 10.9.

The algorithm is:

1. pick initial values for  $\beta$ , denoted  $\beta^{(0)}$ ;
2. at iteration  $r$ , update  $\beta^{(r)}$  as follows:

$$\beta^{(r+1)} = \beta^{(r)} + \delta^{(r)}$$

where

$$(\mathbf{B}^t \Sigma_{\mathbf{m}} \mathbf{B}) \delta^{(r)} = \mathbf{B}^t (\mathbf{n} - \mathbf{m}), \quad (10.14)$$

and solve for  $\delta^{(r)}$ , preferably using a linear algebra routine that does not require taking the inverse of  $\mathbf{B}^t \Sigma_{\mathbf{m}} \mathbf{B}$ ; and

3. repeat until convergence.

Note that  $\mathbf{m}$  in Step 2 is the  $\mathbf{m}$  that results from applying Equation 10.5 with  $\beta^{(r)}$  and taking exponentials.

Let  $c$  and  $d$  designate columns of  $\mathbf{B}$ . Then a computational formula the  $(c, d)$ -th element of  $\mathbf{B}^t \Sigma_{\mathbf{m}} \mathbf{B}$  is

$$\sum_{i=1}^T b_{ic} b_{id} m_i - \frac{1}{N} \left( \sum_{i=1}^T b_{ic} m_i \right) \left( \sum_{i=1}^T b_{id} m_i \right), \quad (10.15)$$

and a computational formula for the  $c$ -th element of  $\mathbf{B}^t (\mathbf{n} - \mathbf{m})$  is

$$\sum_{i=1}^T b_{ic} (n_i - m_i). \quad (10.16)$$

#### Convergence Criteria

The likelihood equations are satisfied if

$$\mathbf{B}^t (\mathbf{n} - \mathbf{m}^{(r)}) = 0. \quad (10.17)$$

An obvious way to satisfy convergence is to require that

$$|\sum b_{ic} n_i - \sum b_{ic} m_i^{(r)}| < \text{crit}_{abs} \quad (10.18)$$

for all  $c = 1, 2, \dots, C$ , where  $\text{crit}_{abs}$  is a user-specified value. Equation 10.18 might be called an “absolute” criterion. Essentially, it states that any differences



between the  $C$  moments of the observed and log-linear smoothed distributions are small enough to ignore. An alternative is to use the “relative” criterion

$$\frac{|\sum b_{ic} n_j - \sum b_{ic} m_j^{(r)}|}{|\sum b_{ic} n_i|} < \text{crit}_{rel}, \quad (10.19)$$

for all  $c = 1, 2, \dots, C$ , where  $\text{crit}_{rel}$  is a user-specified value (likely different from  $\text{crit}_{abs}$ ). Obviously, Equation 10.19 is not a good criterion if  $\sum b_{ic} n_i = 0$ . Other possible convergence criteria are discussed by Holland and Thayer (1987, 2000).

### Initial Values

As initial values, Holland and Thayer (2000) proposed solving for  $\beta^{(0)}$  in the following equation:

$$(\mathbf{B}^t \boldsymbol{\Sigma}_a \mathbf{B}) \beta^{(0)} = \mathbf{B}^t \boldsymbol{\Sigma}_a (\log \mathbf{a} - \boldsymbol{\mu}), \quad (10.20)$$

where

$$\boldsymbol{\Sigma}_a = \mathbf{D}_a - \frac{\mathbf{a} \mathbf{a}^t}{N},$$

and

$$a_i = \rho n_i + (1 - \rho) \left( \frac{N}{T} \right). \quad (10.21)$$

The  $c, d$ -th element of  $\mathbf{B}^t \boldsymbol{\Sigma}_a \mathbf{B}$  can be obtained quite easily using Equation 10.15 with  $a_i$  replacing  $m_i$ . Similarly, the  $c$ -th element of  $\mathbf{B}^t \boldsymbol{\Sigma}_a (\log \mathbf{a} - \boldsymbol{\mu})$  is

$$\sum_{i=1}^T b_{ic} a_i [\log(a_i) - \mu_i] - \frac{1}{N} \left( \sum_{i=1}^T b_{ic} a_i \right) \left( \sum_{i=1}^T a_i [\log(a_i) - \mu_i] \right). \quad (10.22)$$

It is preferable to solve for  $\beta^{(0)}$  in Equation 10.20 using a linear algebra routine that does not require taking the inverse of  $\mathbf{B}^t \boldsymbol{\Sigma}_a \mathbf{B}$ .

Holland and Thayer (1987) set all elements of  $\boldsymbol{\mu}$  to 0 in Equation 10.20, and they set  $\rho$  to .8 in Equation 10.21. These are the conventions used in *Equating Recipes*.<sup>3</sup>

## 10.2 Univariate Smoothing in Equating

In equating with the random groups design, the observed univariate distributions for Forms X and Y might be smoothed using log-linear models. The smoothing degrees  $C_X$  and  $C_Y$  need not be the same. Kolen and Brennan (2004, pp. 77–80) discuss various procedures that might be used to arrive at final choices for  $C_X$  and  $C_Y$ .

The discussion of the log-linear model in Section 10.1.1 is quite general. In particular,

- the model says nothing about the scores associated with each of the  $T$  categories,

---

<sup>3</sup>These conventions can be altered by changing the code in `get.Beta0()`.

- the model does not place any constraints on  $\mu_i$  in Equations 10.3 and 10.5,
- the model says nothing about the  $C$  known constants in each of the rows ( $\mathbf{b}_i$ ) of the design matrix  $\mathbf{B}$ , and
- the moments as defined by  $\mathbf{B}^t \mathbf{n}$  and  $\mathbf{B}^t \mathbf{m}$  are quite general—they are not restricted to power moments.

When a log-linear function is employed to smooth a univariate distribution used for equating, very frequently

1. all  $T$  of the  $\mu_i$  are set to 0,
2. the elements of the design matrix  $\mathbf{B}$  are  $b_{ic} = (i - 1)^c$ , or some function of them,<sup>4</sup> and
3. the moments are power moments (which follows from 2).

These are the conventions used in *Equating Recipes*.

### 10.2.1 Scaling the Design Matrix

For increased numerical stability, Holland and Thayer (1987, p. 15) suggest that the columns of the design matrix  $\mathbf{B}$  be scaled such that

$$\sum_i b_{ic} = 0 \quad \text{and} \quad \sum_i b_{ic}^2 = 1.$$

Under this scaling, the elements of  $\mathbf{B}$  become

$$b'_{ic} = \frac{b_{ic} - \bar{b}_c}{\sqrt{\sum_i (b_{ic} - \bar{b}_c)^2}}, \quad (10.23)$$

which is the default used in *Equating Recipes* for the  $\mathbf{B}$  matrix.

*Equating Recipes* stores both  $\mathbf{B}$  (using the above scaling) and its raw-score counterpart named  $\mathbf{B\_raw}$ . For example, if  $C = 2$  and there are five categories with scores of 0, 1, 2, 3, and 4, then,

$$\mathbf{B\_raw} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} -.63246 & -.45486 \\ -.31623 & -.37905 \\ .00000 & -.15162 \\ .31623 & .22743 \\ .63246 & .75810 \end{bmatrix}. \quad (10.24)$$

### 10.2.2 Raw and Scaled Moments

The observed and estimated moments in *Equating Recipes* that are associated with the design matrix  $\mathbf{B}$  are stored in  $\mathbf{n\_mts}[]$  and  $\mathbf{m\_mts}[]$ , respectively. The  $c$ -th moment based on observed scores is

$$\mathbf{n\_mts}[c] = \sum_i b'_{ic} \left( \frac{n_i}{N} \right), \quad (10.25)$$

---

<sup>4</sup>Recall that  $i = 1, 2, \dots, T$ . For most test score distributions, however, the lowest score is 0.

and an analogous equation applies for  $\mathbf{m\_mts}[c]$  with  $m_i$  replacing  $n_i$ .

In *Equating Recipes*,  $\mathbf{B\_raw}$  is always the design matrix based on powers of observed raw scores, i.e., the scores that are actually associated with the  $T$  categories. Henceforth, the general element of  $\mathbf{B\_raw}$  will be designated  $b_{ic}^*$ . The associated observed and fitted moments are stored in vectors  $\mathbf{n\_mts\_raw}[]$  and  $\mathbf{m\_mts\_raw}[]$ , respectively, but they are computed as central moments (except for the first moment) in the metric of the actual observed scores. Specifically, for the observed moments,  $\mathbf{n\_mts\_raw}[1]$  is the observed mean,  $\bar{b}_1^*$ ;  $\mathbf{n\_mts\_raw}[2]$  is the observed standard deviation  $S(b_{i1}^*)$ ; and for  $c \geq 3$ ,

$$\mathbf{n\_mts\_raw}[c] = \frac{\sum_i (b_{i1}^* - \bar{b}_1^*)^c (n_i/N)}{S(b_{i1}^*)^c}. \quad (10.26)$$

It follows that  $\mathbf{n\_mts\_raw}[3]$  is skewness,  $\mathbf{n\_mts\_raw}[4]$  is kurtosis, etc. The  $\mathbf{m\_mts\_raw}[]$  are defined in an analogous manner, replacing  $n_i$  with  $m_i$  in Equation 10.26.

The model used to estimate  $\mathbf{m}$  employs the  $\mathbf{B}$  matrix. Consequently, to evaluate convergence in the manner of Equations 10.18 or 10.19, it is natural to use the moments in Equation 10.25 and the corresponding  $\mathbf{m}$  moments. However, the metric in the  $\mathbf{B}$  matrix is not the metric of the actual observed scores, whereas the moments in Equation 10.26 are in the observed score metric, as are the corresponding  $\mathbf{m}$  moments.

### 10.2.3 Estimation Options

In the function `Wrapper.Smooth_ULL()` (as well as `Wrapper.Smooth_BLL()`), there are variables that control estimation:

- **scale**: 0 means no scaling of the elements of  $\mathbf{B}$ ; 1 means scaling in the sense of Equation 10.23.
- **Btype**: 0 means match moments based on  $\mathbf{B}$ ; 1 means match moments based on  $\mathbf{B\_raw}$ . When **Btype**=0, there is a potential confusion. Specifically, if **Btype**=0 and **scale**=0, then the design matrix is based on raw scores, and the moment matching is also based on raw scores (see Equation 10.26). By contrast, if **Btype**=0 and **scale**=1, then the design matrix is based on scaled raw scores, and the moment matching is also based on scaled raw scores (see Equation 10.25). When **Btype**=1, the moment matching is based on raw scores (see Equation 10.26) whether **scale**=0 or **scale**=1.
- **ctype**: 0 means use the absolute criterion in Equation 10.18; 1 means use the relative criterion in Equation 10.19.
- **crit**: the convergence criterion value.

The authors usually use **scale**=1, **Btype**=1, **ctype**=0, and **crit**=.000001. Note, however, that the continuized log-linear method of equating (see Chapter 14) requires that **scale**=0.

In the distributed code, the maximum number of iterations (**max\_nit**) is set to 40 in `Smooth_ULL()` (and in `Smooth_BLL()`).

### 10.3 Bivariate Smoothing in Equating

A single bivariate distribution is involved in the single-group design, and two bivariate distributions are involved in the common-item nonequivalent groups design. Consequently, log-linear bivariate smoothing might be used with either design.

The discussion of the log-linear model in Section 10.1 applies to both univariate and bivariate distributions. Consider, for example, a bivariate distribution with three row categories and four column categories, and suppose the desired moment matching involves two moments for row marginals, two moments for column marginals, and one cross-product moment. In this case, the  $\mathbf{B}$  matrix has  $T = 12$  rows (one for each frequency) and  $C = 5$  columns (one for each moment).

Let the column vector of frequencies be ordered such that the elements for the first row of the bivariate frequency distribution come first, the elements for the second row come second, and the elements of the third row come third. If the scores associated with the three row marginals are (0, 1, 2) and the scores associated with the three column marginals are (0, 1, 2, 3), then

$$\mathbf{B}_{\text{raw}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 4 & 0 \\ 0 & 0 & 3 & 9 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 4 & 2 \\ 1 & 1 & 3 & 9 & 3 \\ 2 & 4 & 0 & 0 & 0 \\ 2 & 4 & 1 & 1 & 2 \\ 2 & 4 & 2 & 4 & 4 \\ 2 & 4 & 3 & 9 & 6 \end{bmatrix}.$$

The first two columns are for the first two row marginal moments, columns three and four are for the first two column marginal moments, and column five is for a cross-product moment. The corresponding  $\mathbf{B}$  matrix using the scaling in Equation 10.23 is:

$$\mathbf{B} = \begin{bmatrix} -.35355 & -.28307 & -.38730 & -.28868 & -.22875 \\ -.35355 & -.28307 & -.12910 & -.20620 & -.22875 \\ -.35355 & -.28307 & .12910 & .04124 & -.22875 \\ -.35355 & -.28307 & .38730 & .45363 & -.22875 \\ .00000 & -.11323 & -.38730 & -.28868 & -.22875 \\ .00000 & -.11323 & -.12910 & -.20620 & -.07625 \\ .00000 & -.11323 & .12910 & .04124 & .07625 \\ .00000 & -.11323 & .38730 & .45363 & .22875 \\ .35355 & .39630 & -.38730 & -.28868 & -.22875 \\ .35355 & .39630 & -.12910 & -.20620 & .07625 \\ .35355 & .39630 & .12910 & .04124 & .38125 \\ .35355 & .39630 & .38730 & .45363 & .68624 \end{bmatrix}.$$

### 10.3.1 Cross-products and their Moments

In general, let the bivariate distribution be denoted  $U \times V$ , and let **B.raw** have  $C_u$  columns for fitted moments of  $U$ ,  $C_v$  columns for fitted moments of  $V$ , and  $C_{uv}$  columns for cross-product moments.

For the previous example  $C_u = 2$ ,  $C_v = 2$ ,  $C_{uv} = 1$ , and the first four columns of **B.raw** might be labelled  $u_1, u_1^2, v_1$ , and  $v_1^2$ . Then, column five might be  $u_1 v_1$ , which is the lowest order cross-product moment. It is the cross-product moment associated with the Pearson product-moment correlation. In principal, we could also incorporate one or more of the following higher-order cross-product moments:  $u_1 v_1^2$  (product of columns 1 and 4),  $u_1^2 v_1$  (product of columns 2 and 3), or  $u_1^2 v_1^2$  (product of columns 2 and 4). Doing so, however, may not be advisable, as discussed in Section 10.3.2.

For the **B** matrix, Equation 10.25 applies to both the marginal moments and the cross-product moments. For the **B.raw** matrix, however, Equation 10.26 is for marginal moments, only. The corresponding equation for cross-product moments is discussed next.

Consider a general cross-product moment  $u_1^j v_1^k$ . Note that  $u_1$  is given by column 1 of **B.raw**, and  $v_1$  is given by column  $\nu = C_u + 1$ . The central cross-product moment for the observed scores associated with  $u_1^j v_1^k$  is

$$\mathbf{m\_mts\_raw}[c] = \frac{\sum_i (b_{i1}^* - \bar{b}_1^*)^j (b_{i\nu}^* - \bar{b}_\nu^*)^k (n_i/N)}{S^j(b_{i1}^*) S^k(b_{i\nu}^*)}, \quad (10.27)$$

where  $c = C_u + C_v + 1, \dots, C_u + C_v + C_{uv}$ . An analogous equation applies for **m\_mts\_raw[c]** with  $m_i$  replacing  $n_i$ . Equation 10.27 is the Pearson product-moment correlation when the cross-product moment is  $u_1^1 v_1^1$ .

### 10.3.2 Issues Specific to Equating

For the common-item nonequivalent groups design with an internal anchor, scores on  $V$  are included in  $X$ , i.e.,  $X = U + V$ . Consequently, the  $X \times V$  matrix, has a number of structural zeros. For example, if the possible scores for  $U$  are  $0, 1, \dots, \text{nsu}$ , and the possible scores for  $V$  are  $0, 1, \dots, \text{nsv}$ , then structural zeros occur whenever  $x < v$  or  $x > \text{nsu} + v - 1$ . If bivariate smoothing were performed on the  $X \times V$  matrix, it is highly likely that some if not all of these structural zeros would be given a positive frequency, which is clearly not correct. Therefore, the smoothing should be done on the  $U \times V$  matrix, and then the elements of this smoothed matrix can be positioned in the smoothed  $X \times V$  matrix that retains the structural zeros.

In equating it is expected that  $U$  and  $V$  will be positively correlated; indeed, it is usually expected that they will be quite highly correlated. Furthermore, it seems sensible to expect that for the population of examinees,

- $E(U|v) > E(U|v')$  for  $v > v'$ ,
- $E(V|u) > E(V|u')$  for  $u > u'$ ,
- $\text{Prob}(U > u|v) > \text{Prob}(U > u|v')$  if  $v < v'$ , and
- $\text{Prob}(V > v|u) > \text{Prob}(V > v|u')$  if  $u < u'$ .

If these conditions are fulfilled, then the categories are said to be stochastically ordered.<sup>5</sup> Rosenbaum and Thayer (1987) have noted that including  $u_1^1 v_1^1$  assures stochastic ordering when the correlation between  $U$  and  $V$  is positive, but the inclusion of higher order cross-product moments (i.e., moments other than  $u_1^1 v_1^1$ ) can lead to violations of stochastic ordering. This suggests that if higher order cross-product moments are used, the resulting smoothed bivariate frequency distributions and their marginals should be examined carefully prior to equating.

In the calling sequence of the function `Wrapper_Smooth.BLL()`, there are four variables that relate to estimation. Their names and meaning are the same as discussed in Section 10.2.3.

## 10.4 Functions

The functions used for log-linear smoothing are in `LogLinear.c` and `CG_EquiEquate.c` with the corresponding function prototypes in `LogLinear.h` and `CG_EquiEquate.h`, respectively. The wrapper function(s) for univariate or bivariate log-linear smoothing must be called before calling the wrapper functions for equating.

### 10.4.1 Univariate Smoothing:

#### `Wrapper_Smooth_ULL()` and `Print_ULL()`

The wrapper function that performs univariate log-linear smoothing using the multinomial model is:

```
void Wrapper_Smooth_ULL(struct USTATS *x, int c,
                        int scale, int Btype, int ctype, double crit,
                        FILE *fp, struct ULL_SMOOTH *s)
```

The input variables are:

- ▷ **x** = a `USTATS` structure that contains the univariate distribution to be smoothed;
- ▷ **c** = number of degrees for smoothing;
- ▷ **scale** = type of scaling (0 means powers of raw scores; 1 means powers of scaled raw scores);
- ▷ **Btype** = type of moment matching (see section 10.2.3);
- ▷ **ctype** = type of criterion (0 means absolute criterion; 1 means relative criterion);
- ▷ **crit** = convergence criterion for moment matching; and
- ▷ **fp** = pointer to output file (if non-NULL, then results for iterations are printed to the file pointed to by **fp**).

---

<sup>5</sup>A strict definition of stochastic ordering does not require all four conditions.

The output variable is:

- ▷ **s** = a `ULL_SMOOTH` structure that contains the log-linear smoothing results (see `ERutilities.h` for the definition of `ULL_SMOOTH`).

The associated print function for `Wrapper_Smooth_ULL()` is:

```
void Print_ULL(FILE *fp, char tt[], struct USTATS *x,
               struct ULL_SMOOTH *s, int print_dm,
               int print_mts)
```

The input variables are:

- ▷ **fp** = output file pointer;
- ▷ **tt[]** = user-specified title;
- ▷ **x** = the `USTATS` structure in the calling sequence for `Wrapper_Smooth_ULL()`;
- ▷ **s** = the `ULL_SMOOTH` structure in the calling sequence for `Wrapper_Smooth_ULL()`, which contains the log-linear presmoothed results;
- ▷ **print\_dm** = print design matrices (0 = no printing of matrices; 1 = print B and coefficients of B; 2 = print B, B<sub>raw</sub>, and coefficients of B); and
- ▷ **print\_mts** = print moments (0 = no printing of moments; 1 = print moments for B; 2 = print moments for B and B<sub>raw</sub>).

#### 10.4.2 Bivariate Smoothing:

##### `Wrapper_Smooth_BLL()` and `Print_BLL()`

The wrapper function that performs bivariate log-linear smoothing using the multinomial model is:

```
void Wrapper_Smooth_BLL(struct BSTATS *xv, int anchor,
                       int cu, int cv, int cuv, int cpm[][2],
                       int scale, int Btype, int ctype, double crit,
                       FILE *fp, struct BLL_SMOOTH *s)
```

The input variables are:

- ▷ **xv** = a `BSTATS` structure that contains the bivariate distribution to be smoothed;
- ▷ **anchor** = type of anchor (0 means an external anchor, i.e., scores on **v** are not included in scores on **x**; 1 means an internal anchor, i.e., scores on **v** are included in scores on **x**);
- ▷ **cu** = number of degrees for smoothing for *U*;
- ▷ **cv** = number of degrees for smoothing for *V*;
- ▷ **cuv** = number of degrees for smoothing for *U* × *V* cross-products;
- ▷ **cpm[][2]** = cross-product moments (see Section 10.6);
- ▷ **scale** = type of scaling (0 means powers of raw scores; 1 means powers of scaled raw scores);

- ▷ **Btype** = type of moment matching (see section 10.2.3);
- ▷ **ctype** = type of criterion (0 means absolute criterion; 1 means relative criterion);
- ▷ **crit** = convergence criterion for moment matching; and
- ▷ **fp** = pointer to output file (if non-NULL, then results for iterations are printed to the file pointed to by **fp**).

The output variable is:

- ▷ **s** = a **BLL\_SMOOTH** structure that contains the log-linear smoothing results (see **ERutilities.h** for the definition of **BLL\_SMOOTH**).

The associated print function for **Wrapper\_Smooth\_BLL()** is:

```
void Print_BLL(FILE *fp, char tt[], struct BSTATS *xv,
               struct BLL_SMOOTH *s, int print_dm
               int print_mts, int print_freq, int print_bfd)
```

The input variables are:

- ▷ **fp** = output file pointer;
- ▷ **tt[]** = user-specified title;
- ▷ **xv** = the **BSTATS** structure in the calling sequence for **Wrapper\_Smooth\_BLL()**;
- ▷ **s** = the **BLL\_SMOOTH** structure in the calling sequence for **Wrapper\_Smooth\_BLL()**, which contains the log-linear presmoothed results;
- ▷ **print\_dm** = print design matrices (0 = no printing of matrices; 1 = print B and coefficients for B; 2 = print B, B<sub>raw</sub>, and coefficients for B);
- ▷ **print\_mts** = print moments (0 = no printing of moments; 1 = print moments for B; 2 = print moments for B and B<sub>raw</sub>);
- ▷ **print\_freq** = format for printing actual and fitted frequencies for *U* by *V* matrix (0 = don't print; 1 = column vector format; 2 = matrix format);
- ▷ **print\_bfd** = print fitted bivariate frequency distribution, **bfd[] []** for *X* by *V* matrix (0 = don't print; 1 = print **bfd[] []** and marginals; 2 = print marginals only).

For an external anchor **print\_freq = 2** and **print\_bfd = 1** cause the same matrix to be printed twice.

### 10.4.3 Random Groups Design:

#### **Wrapper\_RL()** and **Print\_RL()**

The wrapper function for performing equipercntile equating with the random-groups design and log-linear presmoothing is **Wrapper\_RL**. Before **Wrapper\_RL** is called, **Wrapper\_Smooth\_ULL** must be called twice—once to smooth *X* and once to smooth *Y*.



```
void Wrapper_RL(char design, char method, char smoothing,
                struct USTATS *x, struct USTATS *y,
                struct ULL_SMOOTH *ullx, struct ULL_SMOOTH *ully,
                int rep, struct PDATA *inall,
                struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' (random-groups design);
- ▷ `method` = 'E' (equipercentile);
- ▷ `smoothing` = 'L' (log-linear smoothing);
- ▷ `x` = a USTATS structure associated with the new form;
- ▷ `y` = a USTATS structure associated with the old form;
- ▷ `ullx` = a ULL\_SMOOTH structure associated with the new form;
- ▷ `ully` = a ULL\_SMOOTH structure associated with the old form; and
- ▷ `rep` = replication number for bootstrap (should be set to 0 for actual equating).

Elements of the following structures are populated by `Wrapper_RL()`:

- ▷ `inall` = a PDATA structure; and
- ▷ `r` = an ERAW\_RESULTS structure that stores equated raw scores and their moments.

The associated print function for `Wrapper_RL()` is:

```
void Print_RL(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `fp` = output file pointer;
- ▷ `tt[]` = user-specified title;
- ▷ `inall` = the PDATA structure populated by `Wrapper_RL()`; and
- ▷ `r` = the ERAW\_RESULTS structure populated by `Wrapper_RL()`.

#### 10.4.4 Single Group Design: `Wrapper_SL()` and `Print_SL()`

The wrapper function for performing equipercentile equating with the single-group design and log-linear presmoothing is `Wrapper_SL()`. Before `Wrapper_SL()` is called, `Wrapper_Smooth_BLL()` must be called to smooth the bivariate  $X \times Y$  distribution. For the single-group design, *Equating Recipes* assumes that Forms X and Y do not share any items in common, or, if they do share common items, that fact is ignored. Functionally, this is analogous to the external anchor case for the common-item nonequivalent groups design. The bivariate log-linear smoothing procedure needs to know this. So, when `Wrapper_Smooth_BLL()`

is called, `anchor` must be set to 0.

```
void Wrapper_SL(char design, char method, char smoothing,
                struct BSTATS *xy, struct BLL_SMOOTH *bllxy,
                int rep, struct PDATA *inall,
                struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'S' (single-group design);
- ▷ `method` = 'E' (equipercentile);
- ▷ `smoothing` = 'L' (log-linear smoothing);
- ▷ `xy` = a `BSTATS` structure (scores on `x` will be put on scale of `y`);
- ▷ `bllxy` = a `BLL_SMOOTH` structure to store smoothing results;
- ▷ `rep` = replication number for bootstrap (should be set to 0 for actual equating).

Elements of the following structures are populated by `Wrapper_SL()`:

- ▷ `inall` = a `PDATA` structure; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores equated raw scores and their moments.

The associated print function for `Wrapper_SL()` is:

```
void Print_SL(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `fp` = output file pointer;
- ▷ `tt[]` = user-specified title;
- ▷ `inall` = the `PDATA` structure populated by `Wrapper_SL()`; and
- ▷ `r` = the `ERAW_RESULTS` structure populated by `Wrapper_SL()`.

#### 10.4.5 *Common-item Nonequivalent Groups Design:* `Wrapper_CL()` and `Print_CL()`

The wrapper function for equipercentile equating for the common-item nonequivalent groups design with log-linear presmoothing is `Wrapper_CL()`. Equipercentile equating includes each of the procedures discussed in Chapter 6, namely, frequency estimation, Braun-Holland (linear) results under frequency estimation, modified frequency estimation, Braun-Holland (linear) results under modified frequency estimation, and chained equipercentile equating.

```
void Wrapper_CL(char design, char method, char smoothing,
                double w1, int anchor, double rv1, double rv2,
                struct BSTATS *xv, struct BSTATS *yv,
```

```

    struct BLL_SMOOTH *bllxv, struct BLL_SMOOTH *blyv,
    int rep, struct PDATA *inall,
    struct ERAW_RESULTS *r)

```

The input variables are:

- ▷ **design** = 'C' (common-item nonequivalent groups design);
- ▷ **method**:
  - 'E' = Frequency Estimation (FE) with Braun-Holland (BH) under FE;
  - 'F' = Modified Frequency Estimation (MFE) with Braun-Holland (BH) under MFE;
  - 'G' = FE, BH-FE, MFE, and BH-MFE;
  - 'C' = Chained;
  - 'H' = FE, BH-FE, and Chained;
  - 'A' = FE, BH-FE, MFE, BH-MFE, and Chained;
- ▷ **smoothing** = 'L' (log-linear smoothing);
- ▷ **w1** = weight for population 1 that took new form; if **w1** is outside the [0,1] range, proportional weights are used;
- ▷ **anchor** = type of anchor (0 means external; 1 means internal);
- ▷ **rv1** = reliability of common-item scores for population 1 on new form (set to 0 for all methods except 'F', 'G, and 'A'; if **rv1** is 0, then MFE cannot be conducted);
- ▷ **rv2** = reliability of common-item scores for population 2 on old form (set to 0 for all methods except 'F', 'G, and 'A'; if **rv2** is 0, then MFE cannot be conducted);
- ▷ **xv** = a **BSTATS** structure for the new form (scores on **v** are for common items);
- ▷ **yv** = a **BSTATS** structure for the old form (scores on **v** are for common items);
- ▷ **bllxv** = a **BLL\_SMOOTH** structure that contains the log-linear smoothing results for the new form;
- ▷ **blyv** = a **BLL\_SMOOTH** structure that contains the log-linear smoothing results for the old form;
- ▷ **rep** = replication number for bootstrap (should be set to 0 for actual equating).

Elements of the following structures are populated by **Wrapper\_CL()**:

- ▷ **inall** = a **PDATA** structure; and
- ▷ **r** = an **ERAW\_RESULTS** structure that stores equating results based on the log-linear presmoothed raw scores.

The associated print function for `Wrapper_CL()` is:

```
void Print_CL(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `fp` = output file pointer;
- ▷ `tt[]` = user-specified title;
- ▷ `inall` = the `PDATA` structure populated by `Wrapper_CL()`; and
- ▷ `r` = the `ERAW_RESULTS` structure populated by `Wrapper_CL()`.

## 10.5 Example: Random Groups

Recall, again, the equipercentile equating example for the random-groups design discussed by Kolen and Brennan (2004, pp. 50-53, 57, 60). The actual equating results for unsmoothed raw scores and scales scores based on unsmoothed raw scores were reported in Chapter 5, and Chapter 9 reported beta-binomial presmoothing results. For the same example, Table 10.1 is a `main()` function that provides code for: (a) log-linear smoothing (degree of six) for raw scores for Forms X and Y; (b) equipercentile equated raw scores based on the log-linear presmoothing; (c) equated scale scores (both unrounded and rounded) based on (b); and (d) parametric bootstrap estimated standard errors for (b) and (c).

### 10.5.1 Presmoothing

After reading in the data for both forms using `ReadFdGet_USTATS()`, `Wrapper_Smooth_ULL()` is called to perform log-linear presmoothing (degree of six) with the results stored in the `ULL_SMOOTH` structures `ullx` and `ully`, which are declared at the top of `main()`. Using `Print_ULL()`, the output for Form X is in Tables 10.2 (partial output) and 10.3. The output for Form Y is in Tables 10.4 (partial output) and 10.5.

### 10.5.2 Equipercentile Equating

`Wrapper_RL()` performs equipercentile equating of the log-linear pre-smoothed raw scores using the results in `ullx` and `ully`, with the equated raw scores and their moments stored in the `ERAW_RESULTS` structure `rREL` declared at the top of `main()`. Then `Wrapper_ESS()` uses the equated raw scores in `rREL` to get equated scale scores (both unrounded and rounded), with the results stored in the `ESS_RESULTS` structure `sREL` declared at the top of `main()`.

Using `Print_RL`, the log-linear smoothed raw-score output is in Table 10.6, which can be compared with the unsmoothed raw-score equating results in Table 5.2. Similarly, the scale score results (based on the log-linear smoothed raw-

score results) in Tables 10.7 and 10.8 can be compared with Tables 5.3 and 5.4, respectively.<sup>6</sup>

### 10.5.3 *Bootstrap Standard Errors*

Using the structure **pdREL** as input, **Wrapper\_Bootstrap()** calls functions that perform parametric bootstrapping under the log-linear model, storing raw score results in **tREL** and scale-score results in **uREL**. The smoothed raw-score, unrounded scale-score, and rounded scale-score output is provided in Tables 10.9, 10.10, and 10.11, respectively. These results can be compared with the unsmoothed results in Tables 8.2, 8.3, and 8.4, respectively.

---

<sup>6</sup>The raw-to-scale score conversion table for Form Y is given in Table 3.3 on page 43.

TABLE 10.1. Main() Code to Illustrate Log-linear Presmoothing for Equipercenile Equating with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct PDATA pdREL;
    struct ERAW_RESULTS rREL;
    struct ESS_RESULTS sREL;
    struct BOOT_ERAW_RESULTS tREL;
    struct BOOT_ESS_RESULTS uREL;
    struct ULL_SMOOTH ullx,ully;

    long idum = -15L;                                /* beginning seed */
    FILE *outf;

    outf = fopen("Chap 10 RG out","w");

    /* Random Groups Design: Kolen and Brennan (2004) Chapter 2 example:
       Equipercenile equating with log-linear smoothing (six degrees
       for x and y) (see pages 79, 81-85) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X',&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y',&y);

    Wrapper_Smooth_ULL(&x, 6, 1, 1, 0, .000001, NULL, &ullx);
    Print_ULL(outf, "ACT Math X", &x, &ullx, 2, 2);

    Wrapper_Smooth_ULL(&y, 6, 1, 1, 0, .000001, NULL, &ully);
    Print_ULL(outf, "ACT Math Y", &y, &ully, 2, 2);

    Wrapper_RL('R','E','L', &x, &y, &ullx, &ully, 0, &pdREL, &rREL);
    Print_RL(outf,"ACT Math---Equi---Log Linear Smoothing",&pdREL, &rREL);

    Wrapper_ESS(&pdREL,&rREL,0,40,1,"yctmath.TXT",1,1,36,&sREL);
    Print_ESS(outf,"ACT Math---Equi---Log Linear Smoothing",&pdREL,&sREL);

    Wrapper_Bootstrap(&pdREL,1000,&idum,&tREL,&uREL);
    Print_Boot_se_eraf(outf,"ACT Math---Equi---Log Linear Smoothing",
        &pdREL,&rREL,&tREL,0);
    Print_Boot_se_ess(outf,"ACT Math---Equi---Log Linear Smoothing",
        &pdREL,&sREL,&uREL,0);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 10.2. RG Output for Log-linear Presmoothing for Form X: General Information and Design Matrices

```

/*****
ACT Math X
Input filename:  actmathfreq.dat

Univariate Log-Linear Smoothing using the Multinomial Model
Number of score categories = 41
Number of persons (i.e., total of frequencies) =    4329
Polynomial degree = 6
Design matrix B is scaled powers of raw scores
such that for any column of B, sum(elements) = 0 and
sum(elements^2) = 1

Convergence criterion:
  |(n_mts_raw[i] - m_mts_raw[i])| <=    0.0000010000  for i = 1 to 6
Number of iterations to convergence = 7
Likelihood-ratio chi-square =    30.60884 with 34 degrees of freedom

COEFFICIENTS:

      Beta[ 1] =    176.22707      Beta[ 4] =   -2013.75825
      Beta[ 2] =   -763.25659      Beta[ 5] =    1328.54662
      Beta[ 3] =   1645.78569      Beta[ 6] =   -368.03746

B DESIGN MATRIX

Category   Score   Deg 1   Deg 2   Deg 3   Deg 4   Deg 5   Deg 6
      0   0.000   -0.26398  -0.17227  -0.13630  -0.11609  -0.10278  -0.09319
      1   1.000   -0.25078  -0.17195  -0.13629  -0.11609  -0.10278  -0.09319
      2   2.000   -0.23758  -0.17099  -0.13623  -0.11609  -0.10278  -0.09319
      3   3.000   -0.22438  -0.16940  -0.13608  -0.11607  -0.10278  -0.09319
      4   4.000   -0.21119  -0.16717  -0.13577  -0.11603  -0.10277  -0.09318
      5   5.000   -0.19799  -0.16430  -0.13526  -0.11595  -0.10276  -0.09318
      6   6.000   -0.18479  -0.16079  -0.13450  -0.11581  -0.10273  -0.09318
      7   7.000   -0.17159  -0.15664  -0.13345  -0.11557  -0.10268  -0.09317
      8   8.000   -0.15839  -0.15185  -0.13204  -0.11519  -0.10259  -0.09315
      9   9.000   -0.14519  -0.14643  -0.13024  -0.11466  -0.10244  -0.09311
     10  10.000   -0.13199  -0.14037  -0.12799  -0.11390  -0.10221  -0.09304
      ...

B_raw DESIGN MATRIX

Category   Score   Deg 1   Deg 2   Deg 3   Deg 4   Deg 5   Deg 6
      0   0.000      0.0      0.0      0.0      0.0      0.0      0.0
      1   1.000      1.0      1.0      1.0      1.0      1.0      1.0
      2   2.000      2.0      4.0      8.0     16.0     32.0     64.0
      3   3.000      3.0     9.0     27.0     81.0    243.0    729.0
      4   4.000      4.0    16.0    64.0    256.0   1024.0   4096.0
      5   5.000      5.0    25.0   125.0   625.0   3125.0  15625.0
      ...
*****/

```





TABLE 10.4. RG Output for Log-linear Presmoothing for Form Y: General Information and Design Matrices

```

/*****
ACT Math Y
Input filename:  actmathfreq.dat

Univariate Log-Linear Smoothing using the Multinomial Model

Number of score categories = 41
Number of persons (i.e., total of frequencies) =    4152
Polynomial degree = 6
Design matrix B is scaled powers of raw scores
such that for any column of B, sum(elements) = 0 and
sum(elements^2) = 1

Convergence criterion:
  |(n_mts_raw[i] - m_mts_raw[i])| <=    0.0000010000  for i = 1 to 6
Number of iterations to convergence = 5
Likelihood-ratio chi-square =    29.45347 with 34 degrees of freedom

COEFFICIENTS:

      Beta[ 1] =    160.80623      Beta[ 4] =   -2344.98568
      Beta[ 2] =   -794.55366      Beta[ 5] =    1554.38629
      Beta[ 3] =   1852.80902      Beta[ 6] =   -425.66355

B DESIGN MATRIX

Category   Score   Deg 1   Deg 2   Deg 3   Deg 4   Deg 5   Deg 6
      0   0.000   -0.26398  -0.17227  -0.13630  -0.11609  -0.10278  -0.09319
      1   1.000   -0.25078  -0.17195  -0.13629  -0.11609  -0.10278  -0.09319
      2   2.000   -0.23758  -0.17099  -0.13623  -0.11609  -0.10278  -0.09319
      3   3.000   -0.22438  -0.16940  -0.13608  -0.11607  -0.10278  -0.09319
      4   4.000   -0.21119  -0.16717  -0.13577  -0.11603  -0.10277  -0.09318
      5   5.000   -0.19799  -0.16430  -0.13526  -0.11595  -0.10276  -0.09318
      6   6.000   -0.18479  -0.16079  -0.13450  -0.11581  -0.10273  -0.09318
      7   7.000   -0.17159  -0.15664  -0.13345  -0.11557  -0.10268  -0.09317
      8   8.000   -0.15839  -0.15185  -0.13204  -0.11519  -0.10259  -0.09315
      9   9.000   -0.14519  -0.14643  -0.13024  -0.11466  -0.10244  -0.09311
     10  10.000   -0.13199  -0.14037  -0.12799  -0.11390  -0.10221  -0.09304
     ...

B_raw DESIGN MATRIX

Category   Score   Deg 1   Deg 2   Deg 3   Deg 4   Deg 5   Deg 6
      0   0.000      0.0      0.0      0.0      0.0      0.0      0.0
      1   1.000      1.0      1.0      1.0      1.0      1.0      1.0
      2   2.000      2.0      4.0      8.0     16.0     32.0     64.0
      3   3.000      3.0     9.0     27.0     81.0    243.0    729.0
      4   4.000      4.0    16.0    64.0    256.0   1024.0   4096.0
      5   5.000      5.0    25.0   125.0   625.0   3125.0  15625.0
     ...
*****/

```

TABLE 10.5. RG Output for Log-linear Fitted Frequencies and Proportions for Form Y

```

/*****/

```

Cate- gory	Score	Actual freqs: nct[]	Fitted freqs: mct[]	prop	fitted prop	fitted crfd	fitted prd
0	0.000	0	0.16871	0.00000	0.00004	0.00004	0.00203
1	1.000	1	1.10996	0.00024	0.00027	0.00031	0.01743
2	2.000	3	4.79113	0.00072	0.00115	0.00146	0.08849
3	3.000	13	14.62855	0.00313	0.00352	0.00499	0.32235
4	4.000	42	33.71508	0.01012	0.00812	0.01311	0.90453
5	5.000	59	62.01393	0.01421	0.01494	0.02804	2.05733
6	6.000	95	95.44080	0.02288	0.02299	0.05103	3.95346
7	7.000	131	127.88645	0.03155	0.03080	0.08183	6.64286
8	8.000	158	154.20922	0.03805	0.03714	0.11897	10.03996
9	9.000	161	171.93311	0.03878	0.04141	0.16038	13.96749
10	10.000	194	181.16464	0.04672	0.04363	0.20401	18.21964
11	11.000	164	183.56162	0.03950	0.04421	0.24822	22.61181
12	12.000	166	181.26672	0.03998	0.04366	0.29188	27.00522
13	13.000	197	176.22753	0.04745	0.04244	0.33433	31.31030
14	14.000	177	169.91606	0.04263	0.04092	0.37525	35.47870
15	15.000	158	163.30315	0.03805	0.03933	0.41458	39.49145
16	16.000	169	156.94698	0.04070	0.03780	0.45238	43.34803
17	17.000	132	151.10819	0.03179	0.03639	0.48877	47.05775
18	18.000	158	145.85228	0.03805	0.03513	0.52390	50.63386
19	19.000	151	141.12784	0.03637	0.03399	0.55789	54.08979
20	20.000	134	136.82224	0.03227	0.03295	0.59085	57.43697
21	21.000	137	132.79972	0.03300	0.03198	0.62283	60.68386
22	22.000	122	128.92689	0.02938	0.03105	0.65388	63.83568
23	23.000	110	125.08876	0.02649	0.03013	0.68401	66.89463
24	24.000	116	121.19732	0.02794	0.02919	0.71320	69.86051
25	25.000	132	117.19365	0.03179	0.02823	0.74143	72.73130
26	26.000	104	113.04427	0.02505	0.02723	0.76865	75.50392
27	27.000	104	108.73273	0.02505	0.02619	0.79484	78.17464
28	28.000	114	104.24754	0.02746	0.02511	0.81995	80.73943
29	29.000	97	99.56820	0.02336	0.02398	0.84393	83.19386
30	30.000	107	94.65104	0.02577	0.02280	0.86673	85.53273
31	31.000	88	89.41734	0.02119	0.02154	0.88826	87.74935
32	32.000	80	83.74698	0.01927	0.02017	0.90843	89.83466
33	33.000	79	77.48201	0.01903	0.01866	0.92709	91.77624
34	34.000	70	70.44670	0.01686	0.01697	0.94406	93.55766
35	35.000	61	62.49120	0.01469	0.01505	0.95911	95.15855
36	36.000	48	53.56356	0.01156	0.01290	0.97201	96.55612
37	37.000	47	43.80322	0.01132	0.01055	0.98256	97.72865
38	38.000	29	33.62630	0.00698	0.00810	0.99066	98.66109
39	39.000	32	23.74283	0.00771	0.00572	0.99638	99.35195
40	40.000	12	15.03557	0.00289	0.00362	1.00000	99.81894

```

    mts[ 1] -0.01347    -0.01347
    mts[ 2] -0.03186    -0.03186
    mts[ 3] -0.03957    -0.03957
    mts[ 4] -0.04282    -0.04282
    mts[ 5] -0.04408    -0.04408
    mts[ 6] -0.04441    -0.04441

    mts_raw[ 1] 18.97977    18.97977
    mts_raw[ 2]  8.93932     8.93932
    mts_raw[ 3]  0.35269     0.35269
    mts_raw[ 4]  2.14636     2.14636
    mts_raw[ 5]  1.78367     1.78367
    mts_raw[ 6]  6.33644     6.33644
/*****/

```

TABLE 10.6. RG Output for Equated Raw Scores  
using Log-linear Smoothing

```

/*****/
ACT Math---Equi---Log Linear Smoothing
Equipercntile Equating with Random Groups Design
and Polynomial Log-Linear Smoothing

Log-Linear Smoothing for new form X:
  number of degrees of smoothing = 6
Log-Linear Smoothing for old form Y:
  number of degrees of smoothing = 6
Input file for new form X: actmathfreq.dat
Input file for old form Y: actmathfreq.dat
-----
Raw Score (X)      y-equiv

    0.00000      -0.43843
    1.00000       0.12386
    2.00000       0.92930
    3.00000       1.82645
    4.00000       2.74098
    5.00000       3.65734
    6.00000       4.57102
    7.00000       5.47247
    8.00000       6.35771
    9.00000       7.27309
   10.00000       8.21428
   11.00000       9.18189
   12.00000      10.17898
   13.00000      11.20917
   14.00000      12.27496
   15.00000      13.37645
   16.00000      14.51108
   17.00000      15.67838
   18.00000      16.86379
   19.00000      18.05664
   20.00000      19.24691
   21.00000      20.42623
   22.00000      21.59111
   23.00000      22.73680
   24.00000      23.85954
   25.00000      24.95936
   26.00000      26.03737
   27.00000      27.09538
   28.00000      28.13566
   29.00000      29.16065
   30.00000      30.17291
   31.00000      31.17493
   32.00000      32.16911
   33.00000      33.15764
   34.00000      34.14242
   35.00000      35.12500
   36.00000      36.10645
   37.00000      37.08735
   38.00000      38.06763
   39.00000      39.04626
   40.00000      40.02023

      Mean      18.98092
      S.D.       8.93543
      Skew       0.35407
      Kurt        2.14639
/*****/

```

TABLE 10.7. RG Output for Unrounded Equated Scale Scores using Log-linear Smoothing

```

/*****/
ACT Math---Equi---Log Linear Smoothing

Name of file containing yct[] []: yctmath.TXT

Equated Scale Scores (unrounded)

Raw Score (x)      y-equiv

    0.00000      0.50000
    1.00000      0.50000
    2.00000      0.50000
    3.00000      0.50000
    4.00000      0.50000
    5.00000      0.50000
    6.00000      0.60849
    7.00000      1.14650
    8.00000      2.17559
    9.00000      3.54211
   10.00000      5.00215
   11.00000      6.36670
   12.00000      7.72870
   13.00000      9.20159
   14.00000     10.69653
   15.00000     12.08549
   16.00000     13.43373
   17.00000     14.82772
   18.00000     16.19751
   19.00000     17.43666
   20.00000     18.57340
   21.00000     19.66779
   22.00000     20.66317
   23.00000     21.51436
   24.00000     22.23464
   25.00000     22.89359
   26.00000     23.54121
   27.00000     24.19064
   28.00000     24.82562
   29.00000     25.36174
   30.00000     25.80213
   31.00000     26.23992
   32.00000     26.74786
   33.00000     27.34406
   34.00000     28.01982
   35.00000     28.82451
   36.00000     29.90320
   37.00000     31.33118
   38.00000     32.79310
   39.00000     34.25378
   40.00000     35.50352

      Mean      16.51261
      S.D.       8.37005
      Skew       -0.12930
      Kurt        2.04214
/*****/

```

TABLE 10.8. RG Output for Rounded Equated Scale Scores using Log-linear Smoothing

```

/*****/
ACT Math---Equi---Log Linear Smoothing

Name of file containing yct[]: yctmath.TXT

Equated Scale Scores (rounded)

Raw Score (x)      y-equiv

    0.00000         1
    1.00000         1
    2.00000         1
    3.00000         1
    4.00000         1
    5.00000         1
    6.00000         1
    7.00000         1
    8.00000         2
    9.00000         4
   10.00000         5
   11.00000         6
   12.00000         8
   13.00000         9
   14.00000        11
   15.00000        12
   16.00000        13
   17.00000        15
   18.00000        16
   19.00000        17
   20.00000        19
   21.00000        20
   22.00000        21
   23.00000        22
   24.00000        22
   25.00000        23
   26.00000        24
   27.00000        24
   28.00000        25
   29.00000        25
   30.00000        26
   31.00000        26
   32.00000        27
   33.00000        27
   34.00000        28
   35.00000        29
   36.00000        30
   37.00000        31
   38.00000        33
   39.00000        34
   40.00000        36

      Mean      16.54955
      S.D.       8.38499
      Skew       -0.12340
      Kurt        2.01213
/*****/

```

TABLE 10.9. RG Output for Bootstrap Estimates of Standard Errors for Equated Raw Scores using Log-linear Smoothing

/\*\*\*\*\*  
 ACT Math---Equi---Log Linear Smoothing

Bootstrap standard errors for raw scores

Number of bootstrap replications = 1000

Score (x)	fd(x)	Method 0:	y-equiv
		eraw	bse
0.00000	0	-0.43843	0.34528
1.00000	1	0.12386	0.51534
2.00000	1	0.92930	0.73855
3.00000	3	1.82645	0.51255
4.00000	9	2.74098	0.30944
5.00000	18	3.65734	0.22289
6.00000	59	4.57102	0.19285
7.00000	67	5.47247	0.18019
8.00000	91	6.35771	0.17395
9.00000	144	7.27309	0.17200
10.00000	149	8.21428	0.17201
11.00000	192	9.18189	0.17821
12.00000	192	10.17898	0.18866
13.00000	192	11.20917	0.19941
14.00000	201	12.27496	0.21738
15.00000	204	13.37645	0.23981
16.00000	217	14.51108	0.26109
17.00000	181	15.67838	0.27516
18.00000	184	16.86379	0.28598
19.00000	170	18.05664	0.29921
20.00000	201	19.24691	0.30935
21.00000	147	20.42623	0.31725
22.00000	163	21.59111	0.32259
23.00000	147	22.73680	0.32852
24.00000	140	23.85954	0.32968
25.00000	147	24.95936	0.33313
26.00000	126	26.03737	0.33614
27.00000	113	27.09538	0.33774
28.00000	100	28.13566	0.33481
29.00000	106	29.16065	0.33157
30.00000	107	30.17291	0.32602
31.00000	91	31.17493	0.31937
32.00000	83	32.16911	0.31504
33.00000	73	33.15764	0.31357
34.00000	72	34.14242	0.31097
35.00000	75	35.12500	0.30433
36.00000	50	36.10645	0.29614
37.00000	37	37.08735	0.29019
38.00000	38	38.06763	0.28681
39.00000	23	39.04626	0.26750
40.00000	15	40.02023	0.19352
Ave. bse			0.27647

\*\*\* means bse = 0

/\*\*\*\*\*

TABLE 10.10. RG Output for Bootstrap Estimates of Standard Errors for Unrounded Equated Scale Scores using Log-linear Smoothing

```

/*****
ACT Math---Equi---Log Linear Smoothing

Bootstrap standard errors for unrounded scale scores

Number of bootstrap replications = 1000

Name of file containing conversion table for Y: yctmath.TXT

      Method 0:      y-equiv
-----
Score (x)  fd(x)      essu      bse

0.00000    0        0.50000      ***
1.00000    1        0.50000      ***
2.00000    1        0.50000      ***
3.00000    3        0.50000      ***
4.00000    9        0.50000      ***
5.00000   18        0.50000    0.00480
6.00000   59        0.60849    0.03622
7.00000   67        1.14650    0.17266
8.00000   91        2.17559    0.25134
9.00000  144        3.54211    0.27233
10.00000  149        5.00215    0.24749
11.00000  192        6.36670    0.24286
12.00000  192        7.72870    0.26706
13.00000  192        9.20159    0.28656
14.00000  201       10.69653    0.28485
15.00000  204       12.08549    0.28588
16.00000  217       13.43373    0.31083
17.00000  181       14.82772    0.32738
18.00000  184       16.19751    0.31619
19.00000  170       17.43666    0.29597
20.00000  201       18.57340    0.29291
21.00000  147       19.66779    0.28549
22.00000  163       20.66317    0.25875
23.00000  147       21.51436    0.22629
24.00000  140       22.23464    0.20228
25.00000  147       22.89359    0.19918
26.00000  126       23.54121    0.20414
27.00000  113       24.19064    0.20854
28.00000  100       24.82562    0.19012
29.00000  106       25.36174    0.15644
30.00000  107       25.80213    0.14000
31.00000   91       26.23992    0.15111
32.00000   83       26.74786    0.17680
33.00000   73       27.34406    0.20227
34.00000   72       28.01982    0.23204
35.00000   75       28.82451    0.29177
36.00000   50       29.90320    0.38435
37.00000   37       31.33118    0.42788
38.00000   38       32.79310    0.42958
39.00000   23       34.25378    0.37031
40.00000   15       35.50352    0.30141

      Ave. bse                      0.26165
-----
*** means bse = 0
*****/

```

TABLE 10.11. RG Output for Bootstrap Estimates of Standard Errors for Rounded Equated Scale Scores using Log-linear Smoothing

```

/*****
ACT Math---Equi---Log Linear Smoothing

```

```

Bootstrap standard errors for rounded scale scores

```

```

Number of bootstrap replications = 1000

```

```

Name of file containing conversion table for Y: yctmath.TXT

```

Score (x)	fd(x)	Method 0:	y-equiv
		-----	-----
		essr	bse
0.00000	0	1	***
1.00000	1	1	***
2.00000	1	1	***
3.00000	3	1	***
4.00000	9	1	***
5.00000	18	1	***
6.00000	59	1	***
7.00000	67	1	0.08341
8.00000	91	2	0.29261
9.00000	144	4	0.50133
10.00000	149	5	0.20503
11.00000	192	6	0.45892
12.00000	192	8	0.39281
13.00000	192	9	0.36679
14.00000	201	11	0.44124
15.00000	204	12	0.29449
16.00000	217	13	0.49735
17.00000	181	15	0.39405
18.00000	184	16	0.39453
19.00000	170	17	0.50047
20.00000	201	19	0.48949
21.00000	147	20	0.45479
22.00000	163	21	0.43948
23.00000	147	22	0.49962
24.00000	140	22	0.28344
25.00000	147	23	0.14676
26.00000	126	24	0.49651
27.00000	113	24	0.24665
28.00000	100	25	0.21387
29.00000	106	25	0.36113
30.00000	107	26	0.14676
31.00000	91	26	0.19839
32.00000	83	27	0.26987
33.00000	73	27	0.40963
34.00000	72	28	0.18672
35.00000	75	29	0.35978
36.00000	50	30	0.44449
37.00000	37	31	0.54650
38.00000	38	33	0.51111
39.00000	23	34	0.49106
40.00000	15	36	0.50196
Ave. bse			0.38764

```

*** means bse = 0

```

```

/*****

```



## 10.6 Example: Common-item Nonequivalent Groups

Kolen and Brennan (2004, chap. 5) briefly discuss a few results for the internal-anchor example considered here that involves log-linear presmoothing with the common-item nonequivalent groups design. This example uses the same data as that employed in Section 4.4 for linear equating and in Section 6.6 for equipercentile equating without presmoothing, but here  $w1$  is set to 1.

For this example, Table 10.12 is a `main()` function that provides code for: (a) bivariate log-linear smoothing for  $X$  and  $V$  in population 1 and for  $Y$  and  $V$  in population 2; (b) equipercentile equating using the log-linear presmoothed raw scores; and (c) parametric bootstrap estimated standard errors for equated raw scores.

### 10.6.1 Presmoothing

For both forms, the data are read using `convertFtoW` and `ReadRawGet_BSTATS()`, with results stored in the `BSTATS` structures `xv` and `yv`. Then, `Wrapper_Smooth_BLL()` is called twice—once to perform log-linear presmoothing for the  $X \times V$  bivariate raw score distribution, and once to perform presmoothing for the  $Y \times V$  distribution. The smoothed results are stored in the `BLL_SMOOTH` structures `bllxv` and `bllyv`, respectively, which are declared at the top of `main()`.

For both forms, six power moments are fitted for both marginals, as well as one cross-product moment, identified as `cpm1`, which is declared and initialized at the top of `main()`:

```
int cpm1[1][2] = {{1,1}};
```

This statement means that the single cross-product moment is  $u^1v^1$ .

Suppose three cross-product moments were desired:  $u^1v^1$ ,  $u^2v^1$ , and  $u^1v^2$ ; and suppose they are defined by `cpm3`. Then, in `Wrapper_Smooth_BLL()`, `cuv` would be set to 3, and `cpm3` would be declared and initialized by the statement:

```
int cpm3[3][2] = {{1,1}, {2,1}, {1,2}};
```

Of course, `cpm1` in the calling sequence for `Wrapper_Smooth_BLL()` in `main()` would have to be replaced by `cpm3`.

Using `Print_BLL()`, the full set of output for Form X is provided in Tables 10.13–10.18. Specifically,

- Table 10.13 describes the log-linear model input characteristics (note that the  $X'$  is used in the same sense as  $U$  in Section 10.3.2, namely, scores on non-common items);
- Tables 10.14 and 10.15 provide the actual (i.e., observed) and fitted frequencies and marginals for the  $X' \times V$  matrices, respectively;
- Table 10.16 provides the actual and fitted marginal and cross-product moments for  $X'$  and  $V$ , for both the scaled design matrix  $B$  and the raw design matrix  $B_{\text{raw}}$ ;

- Table 10.17 provides the bivariate fitted distribution for the  $X \times V$  matrix; and
- for both  $X$  and  $V$ , Table 10.18 provides fitted frequencies, relative frequencies, cumulative relative frequencies, and percentile ranks.

For Form Y, only partial output is provided here, namely, Table 10.19 which gives the fitted marginals for  $Y$  and  $V$ .

### 10.6.2 Equipercntile Equating

`Wrapper_CL()` performs equipercntile equating of the log-linear pre-smoothed raw scores using the results in `b1lxv` and `b1lyv`, with the equated raw scores and their moments stored in the `ERAW_RESULTS` structure `rCEL` declared at the top of `main()`. Since `method` is specified as 'A', equating is performed for five procedures: FE, BH-FE, MFE, BH-MFE, and Chained (see Section 10.4.5).

Using `Print_CL()`, the log-linear smoothing input characteristics are described in Tables 10.20, and the output for the five procedures in Table 10.21. The `main()` function also calls `Print_SynDens()` to print the synthetic densities (see page 69), but that output is not provided here.

### 10.6.3 Bootstrap Standard Errors

Using the structure `pdREL` as input, `Wrapper_Bootstrap()` calls functions that perform parametric bootstrapping under the log-linear model, storing results in `tCEL`. The output is provided in Table 10.22.

TABLE 10.12. Main() Code to Illustrate Log-linear Presmoothing for Equipercentile Equating with Common-item Nonequivalent Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xv,yv;
    struct PDATA pdCEL;
    struct ERAW_RESULTS rCEL;
    struct BOOT_ERAW_RESULTS tCEL;
    struct BLL_SMOOTH bllxv, bllyv;

    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm1[1][2] = {{1,1}};

    long idum = -15L;                                /* beginning seed */
    FILE *outf;

    outf = fopen("Chap 10 CG out","w");

    /* CINEG Design: Kolen and Brennan (2004, chap. 5) example:
       Equipercentile equating with log-linear smoothing */

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'X','V",&xv);

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'Y','V",&yv);

    Wrapper_Smooth_BLL(&xv, 1, 6, 6, 1, cpm1, 1, 1, 0, .000001, NULL, &bllxv);
    Print_BLL(outf, "log-linear smoothing of x and v in pop 1",
               &xv, &bllxv, 0, 2, 2, 1);

    Wrapper_Smooth_BLL(&yv, 1, 6, 6, 1, cpm1, 1, 1, 0, .000001, NULL, &bllyv);
    Print_BLL(outf, "log-linear smoothing of y and v in pop 2",
               &yv, &bllyv, 0, 2, 2, 1);

    Wrapper_CL('C','A','L',1,1,.5584431,.5735077,&xv,&yv,&bllxv,&bllyv,
               0,&pdCEL,&rCEL);
    Print_CL(outf,"w1 = 1, log-linear smoothing",&pdCEL,&rCEL);
    Print_SynDens(outf,"w1 = 1, log-linear smoothing",&pdCEL,&rCEL);

    Wrapper_Bootstrap(&pdCEL,1000,&idum,&tCEL,NULL);
    Print_Boot_se_eraw(outf,"w1 = 1, log-linear smoothing",
                       &pdCEL,&rCEL,&tCEL,0);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 10.13. CG Output for Log-linear Presmoothing for Form X: General Information

```

/*****
log-linear smoothing of x and v in pop 1

```

```

Input filename: mondatx-temp

```

```

Bivariate Log-Linear Smoothing using the Multinomial Model

```

```

NOTE: In this output, variables are identified as
X, X', and V, where  $X = X' + V$ ; i.e., X is the
total score, X' is the non-common-item score, and
V is the common-item score.

```

```

Number of score categories for X' = 25
Number of score categories for V = 13
Total number of score categories = 325

```

```

Number of persons (i.e., total of frequencies) = 1655

```

```

Polynomial degree for X' = 6
Polynomial degree for V = 6
Number of cross-products (X',V) = 1
Cross-Product moments: (X'1,V1)
Number of columns in design matrix = 13

```

```

Design matrix B is scaled powers of raw scores
such that for any column of B, sum(elements) = 0 and
sum(elements^2) = 1

```

```

Convergence criterion:
|(n_mts_raw[i] - m_mts_raw[i])| <= 0.0000010000 for i = 1 to 13

```

```

Number of iterations to convergence = 6

```

```

Likelihood-ratio chi-square = 266.52490 with 311 degrees of freedom

```

```

COEFFICIENTS:

```

```

Beta[ 1] = 278.75747
Beta[ 2] = -1284.52615
Beta[ 3] = 2553.79434
Beta[ 4] = -3259.74592
Beta[ 5] = 2357.72858
Beta[ 6] = -739.28768
Beta[ 7] = 80.03203
Beta[ 8] = -336.03238
Beta[ 9] = 217.85262
Beta[10] = 13.37271
Beta[11] = -156.52101
Beta[12] = 71.62851
Beta[13] = 150.34441

```

```

/*****

```

TABLE 10.14. CG Output for Log-linear Presmoothing for Form X: Observed Frequencies

[illegible]

```

*****
/******
*****
FITTED FREQUENCIES
*****
/******
*****

```

Category	Score	V= 0	V= 1	V= 2	V= 3	V= 4	V= 5	V= 6	V= 7	V= 8	V= 9	V= 10	V= 11	V= 12
		0.000	1.000	2.000	3.000	4.000	5.000	6.000	7.000	8.000	9.000	10.000	11.000	12.000
X'= 0	0.000	0.041	0.092	0.106	0.070	0.030	0.009	0.002	0.000	0.000	0.000	0.000	0.000	0.000
X'= 1	1.000	0.243	0.618	0.803	0.603	0.291	0.097	0.024	0.004	0.001	0.000	0.000	0.000	0.000
X'= 2	2.000	0.779	2.236	3.285	2.791	1.521	0.576	0.160	0.033	0.005	0.001	0.000	0.000	0.000
X'= 3	3.000	1.559	5.056	8.397	8.067	4.971	2.129	0.668	0.158	0.028	0.004	0.000	0.000	0.000
X'= 4	4.000	2.180	7.993	15.010	16.302	11.358	5.499	1.951	0.521	0.106	0.016	0.002	0.000	0.000
X'= 5	5.000	2.322	9.627	20.438	25.097	19.770	10.820	4.340	1.312	0.300	0.052	0.007	0.001	0.000
X'= 6	6.000	2.009	9.416	22.603	31.379	27.947	17.294	7.843	2.680	0.694	0.135	0.020	0.002	0.000
X'= 7	7.000	1.479	7.840	21.278	33.397	33.629	23.527	12.063	4.660	1.365	0.300	0.049	0.006	0.001
X'= 8	8.000	0.959	5.748	17.635	31.295	35.628	28.181	16.336	7.135	2.362	0.588	0.109	0.015	0.002
X'= 9	9.000	0.561	3.800	13.181	26.445	34.038	30.440	19.950	9.851	3.687	1.038	0.217	0.034	0.004
X'= 10	10.000	0.301	2.304	9.035	20.495	29.825	30.155	22.345	12.475	5.279	1.679	0.397	0.069	0.009
X'= 11	11.000	0.150	1.297	5.749	14.745	24.259	27.731	23.232	14.664	7.016	2.524	0.674	0.133	0.020
X'= 12	12.000	0.070	0.683	3.426	9.935	18.481	23.885	22.623	16.145	8.734	3.551	1.073	0.240	0.041
X'= 13	13.000	0.031	0.340	1.925	6.312	13.275	19.397	20.772	16.760	10.250	4.713	1.609	0.407	0.078
X'= 14	14.000	0.013	0.160	1.025	3.800	9.036	14.928	18.074	16.487	11.400	5.926	2.288	0.655	0.142
X'= 15	15.000	0.005	0.072	0.519	2.175	5.847	10.922	14.950	15.419	12.054	7.084	3.092	1.000	0.246
X'= 16	16.000	0.002	0.031	0.250	1.184	3.600	7.601	11.764	13.718	12.125	8.056	3.975	1.454	0.404
X'= 17	17.000	0.001	0.012	0.114	0.612	2.101	5.017	8.779	11.573	11.565	8.688	4.847	2.004	0.630
X'= 18	18.000	0.000	0.005	0.049	0.297	1.153	3.113	6.158	9.178	10.369	8.807	5.555	2.597	0.923
X'= 19	19.000	0.000	0.002	0.019	0.133	0.585	1.785	3.992	6.726	8.592	8.251	5.884	3.110	1.249
X'= 20	20.000	0.000	0.001	0.007	0.054	0.266	0.919	2.325	4.430	6.397	6.945	5.600	3.346	1.520
X'= 21	21.000	0.000	0.000	0.002	0.019	0.105	0.408	1.166	2.511	4.100	5.033	4.588	3.100	1.591
X'= 22	22.000	0.000	0.000	0.001	0.005	0.033	0.146	0.473	1.152	2.127	2.951	3.042	2.323	1.349
X'= 23	23.000	0.000	0.000	0.000	0.001	0.008	0.039	0.143	0.393	0.819	1.285	1.498	1.294	0.849
X'= 24	24.000	0.000	0.000	0.000	0.000	0.001	0.007	0.028	0.089	0.209	0.371	0.489	0.477	0.354

TABLE 10.16. CG Output for Log-linear Presmoothing for Form X: Moments

```

/*****/
MOMENTS
          Based on      Based on
          Actual        Fitted
          nct[]         mct[]

X' moments based on B:

      mts[ 1]   -0.00989   -0.00989
      mts[ 2]   -0.01850   -0.01850
      mts[ 3]   -0.02184   -0.02184
      mts[ 4]   -0.02293   -0.02293
      mts[ 5]   -0.02304   -0.02304
      mts[ 6]   -0.02272   -0.02272

V moments based on B:

      mts[ 1]   -0.01325   -0.01325
      mts[ 2]   -0.02177   -0.02177
      mts[ 3]   -0.02463   -0.02463
      mts[ 4]   -0.02528   -0.02528
      mts[ 5]   -0.02502   -0.02502
      mts[ 6]   -0.02441   -0.02441

X'V moments based on B:

      mts[ 1]   -0.00778   -0.00778

X' (central) moments based on B_raw:

      mts_raw[ 1]  10.71420   10.71420
      mts_raw[ 2]   4.63037   4.63038
      mts_raw[ 3]   0.46809   0.46809
      mts_raw[ 4]   2.55356   2.55356
      mts_raw[ 5]   2.81329   2.81329
      mts_raw[ 6]   9.73607   9.73607

V (central) moments based on B_raw:

      mts_raw[ 1]   5.10634   5.10634
      mts_raw[ 2]   2.37602   2.37602
      mts_raw[ 3]   0.41168   0.41168
      mts_raw[ 4]   2.76829   2.76829
      mts_raw[ 5]   2.92285   2.92285
      mts_raw[ 6]  11.96367  11.96367

X'V (central) moments based on B_raw:

      mts_raw[ 1]   0.70564   0.70564
/*****/

```

Category	Score	V= 0	V= 1	V= 2	V= 3	V= 4	V= 5	V= 6	V= 7	V= 8	V= 9	V= 10	V= 11	V= 12
		0.000	1.000	2.000	3.000	4.000	5.000	6.000	7.000	8.000	9.000	10.000	11.000	12.000
X= 0	0.000	0.041	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
X= 1	1.000	0.243	0.092	0.100	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
X= 2	2.000	0.779	0.618	0.106	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
X= 3	3.000	1.559	2.236	0.803	0.070	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
X= 4	4.000	2.180	5.056	3.285	0.603	0.030	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
X= 5	5.000	2.322	7.993	8.397	2.791	0.291	0.009	0.000	0.000	0.000	0.000	0.000	0.000	0.000
X= 6	6.000	2.009	9.627	15.010	8.067	1.521	0.097	0.002	0.000	0.000	0.000	0.000	0.000	0.000
X= 7	7.000	1.479	9.416	20.438	16.302	4.371	0.576	0.024	0.000	0.000	0.000	0.000	0.000	0.000
X= 8	8.000	0.959	7.840	22.603	25.097	11.358	2.129	0.160	0.004	0.000	0.000	0.000	0.000	0.000
X= 9	9.000	0.561	5.748	21.278	31.379	19.770	5.499	0.668	0.033	0.001	0.000	0.000	0.000	0.000
X= 10	10.000	0.301	3.800	17.635	33.397	27.947	10.820	1.951	0.005	0.000	0.000	0.000	0.000	0.000
X= 11	11.000	0.150	2.304	13.181	31.295	33.629	17.294	4.340	0.521	0.028	0.001	0.000	0.000	0.000
X= 12	12.000	0.070	1.297	9.035	26.445	35.628	23.527	7.843	1.312	0.106	0.004	0.000	0.000	0.000
X= 13	13.000	0.031	0.683	5.749	20.495	34.038	28.181	12.063	2.680	0.300	0.016	0.000	0.000	0.000
X= 14	14.000	0.013	0.340	3.426	14.745	29.825	30.440	16.336	4.660	0.694	0.052	0.002	0.000	0.000
X= 15	15.000	0.005	0.160	1.925	9.935	24.259	30.155	19.950	7.135	1.365	0.135	0.007	0.000	0.000
X= 16	16.000	0.002	0.072	1.025	6.312	18.481	27.731	22.345	8.851	2.362	0.300	0.020	0.001	0.000
X= 17	17.000	0.001	0.031	0.519	3.800	13.275	23.885	23.232	12.475	3.687	0.588	0.049	0.002	0.000
X= 18	18.000	0.000	0.012	0.250	2.175	9.036	19.397	22.672	14.664	5.279	1.038	0.109	0.006	0.000
X= 19	19.000	0.000	0.005	0.114	1.184	5.847	14.928	20.772	16.145	7.016	1.679	0.217	0.015	0.001
X= 20	20.000	0.000	0.002	0.049	0.612	3.600	10.922	18.074	16.760	8.734	2.524	0.397	0.034	0.002
X= 21	21.000	0.000	0.001	0.019	0.297	2.101	7.601	14.950	16.487	10.250	3.551	0.674	0.069	0.004
X= 22	22.000	0.000	0.000	0.007	0.133	1.153	5.017	11.764	15.419	11.400	4.713	1.073	0.133	0.009
X= 23	23.000	0.000	0.000	0.002	0.054	0.585	3.113	8.779	13.718	12.054	5.926	1.609	0.240	0.020
X= 24	24.000	0.000	0.000	0.001	0.019	0.266	1.785	6.158	1					



TABLE 10.18. CG Output for Log-linear Presmoothing for Form X:  
Fitted Marginal Distributions of  $X$  and  $V$  in Population 1

/***** FITTED MARGINAL DISTRIBUTION OF X -----					
Category	Score	Freq	Rel Freq	Cum RFreq	Perc Rank
0	0.000	0.04110	0.00002	0.00002	0.00124
1	1.000	0.33569	0.00020	0.00023	0.01262
2	2.000	1.50309	0.00091	0.00114	0.06818
3	3.000	4.66767	0.00282	0.00396	0.25461
4	4.000	11.15326	0.00674	0.01070	0.73258
5	5.000	21.80280	0.01317	0.02387	1.72823
6	6.000	36.33262	0.02195	0.04582	3.48459
7	7.000	53.20775	0.03215	0.07797	6.18973
8	8.000	70.15062	0.04239	0.12036	9.91657
9	9.000	84.93588	0.05132	0.17168	14.60197
10	10.000	96.01458	0.05801	0.22969	20.06875
11	11.000	102.74264	0.06208	0.29178	26.07350
12	12.000	105.26548	0.06360	0.35538	32.35773
13	13.000	104.23710	0.06298	0.41836	38.68711
14	14.000	100.53192	0.06074	0.47911	44.87349
15	15.000	95.03150	0.05742	0.53653	50.78175
16	16.000	88.50194	0.05348	0.59000	56.32657
17	17.000	81.54419	0.04927	0.63927	61.46391
18	18.000	74.59044	0.04507	0.68434	66.18097
19	19.000	67.92383	0.04104	0.72539	70.48654
20	20.000	61.70636	0.03728	0.76267	74.40286
21	21.000	56.00638	0.03384	0.79651	77.95913
22	22.000	50.82186	0.03071	0.82722	81.18657
23	23.000	46.09878	0.02785	0.85507	84.11469
24	24.000	41.74535	0.02522	0.88030	86.76859
25	25.000	37.64467	0.02275	0.90304	89.16708
26	26.000	33.66854	0.02034	0.92339	91.32156
27	27.000	29.69565	0.01794	0.94133	93.23588
28	28.000	25.63575	0.01549	0.95682	94.90753
29	29.000	21.45762	0.01297	0.96979	96.33029
30	30.000	17.21283	0.01040	0.98019	97.49858
31	31.000	13.04240	0.00788	0.98807	98.41263
32	32.000	9.15540	0.00553	0.99360	99.08326
33	33.000	5.78351	0.00349	0.99709	99.53459
34	34.000	3.13081	0.00189	0.99898	99.80390
35	35.000	1.32602	0.00080	0.99979	99.93855
36	36.000	0.35397	0.00021	1.00000	99.98931
----- FITTED MARGINAL DISTRIBUTION OF V -----					
Category	Score	Freq	Rel Freq	Cum RFreq	Perc Rank
0	0.000	12.70401	0.00768	0.00768	0.38381
1	1.000	57.32994	0.03464	0.04232	2.49964
2	2.000	144.85850	0.08753	0.12984	8.60805
3	3.000	235.21419	0.14212	0.27197	20.09061
4	4.000	277.75896	0.16783	0.43980	35.58829
5	5.000	264.62525	0.15989	0.59969	51.97452
6	6.000	220.15939	0.13303	0.73272	66.62058
7	7.000	168.07481	0.10156	0.83427	78.34971
8	8.000	119.58677	0.07226	0.90653	87.04039
9	9.000	77.99740	0.04713	0.95366	93.00970
10	10.000	45.01231	0.02720	0.98086	96.72601
11	11.000	22.26703	0.01345	0.99431	98.75861
12	12.000	9.41142	0.00569	1.00000	99.71567
/*****					

TABLE 10.19. CG Output for Log-linear Presmoothing for Form Y:  
Fitted Marginal Distributions of Y and V in Population 2

/***** FITTED MARGINAL DISTRIBUTION OF Y -----					
Category	Score	Freq	Rel Freq	Cum RFreq	Perc Rank
0	0.000	0.01867	0.00001	0.00001	0.00057
1	1.000	0.15350	0.00009	0.00011	0.00583
2	2.000	0.70087	0.00043	0.00053	0.03191
3	3.000	2.19911	0.00134	0.00188	0.12043
4	4.000	5.28884	0.00323	0.00510	0.34900
5	5.000	10.46092	0.00639	0.01149	0.82976
6	6.000	17.83032	0.01089	0.02238	1.69335
7	7.000	27.04021	0.01651	0.03888	3.06302
8	8.000	37.34049	0.02280	0.06168	5.02825
9	9.000	47.79967	0.02918	0.09086	7.62715
10	10.000	57.54565	0.03513	0.12599	10.84282
11	11.000	65.93746	0.04025	0.16625	14.61215
12	12.000	72.62867	0.04434	0.21059	18.84188
13	13.000	77.53912	0.04734	0.25793	23.42576
14	14.000	80.77841	0.04932	0.30724	28.25841
15	15.000	82.56107	0.05040	0.35765	33.24435
16	16.000	83.13750	0.05076	0.40840	38.30230
17	17.000	82.74830	0.05052	0.45892	43.36597
18	18.000	81.59986	0.04982	0.50874	48.38270
19	19.000	79.85507	0.04875	0.55749	53.31112
20	20.000	77.63283	0.04739	0.60488	58.11844
21	21.000	75.01166	0.04579	0.65068	62.77792
22	22.000	72.03463	0.04398	0.69465	67.26651
23	23.000	68.71442	0.04195	0.73660	71.56288
24	24.000	65.03845	0.03971	0.77631	75.64569
25	25.000	60.97512	0.03723	0.81354	79.49226
26	26.000	56.48241	0.03448	0.84802	83.07766
27	27.000	51.52064	0.03145	0.87947	86.37445
28	28.000	46.07028	0.02813	0.90760	89.35342
29	29.000	40.15422	0.02451	0.93211	91.98542
30	30.000	33.85925	0.02067	0.95278	94.24469
31	31.000	27.34492	0.01669	0.96948	96.11295
32	32.000	20.82937	0.01272	0.98219	97.58347
33	33.000	14.57333	0.00890	0.99109	98.66414
34	34.000	8.91688	0.00544	0.99653	99.38118
35	35.000	4.34009	0.00265	0.99918	99.78585
36	36.000	1.33778	0.00082	1.00000	99.95916
----- FITTED MARGINAL DISTRIBUTION OF V -----					
Category	Score	Freq	Rel Freq	Cum RFreq	Perc Rank
0	0.000	10.42217	0.00636	0.00636	0.31814
1	1.000	37.52806	0.02291	0.02927	1.78182
2	2.000	88.86849	0.05425	0.08353	5.64008
3	3.000	154.46420	0.09430	0.17783	13.06782
4	4.000	212.45529	0.12970	0.30753	24.26804
5	5.000	244.21519	0.14909	0.45663	38.20792
6	6.000	244.32187	0.14916	0.60578	53.12053
7	7.000	218.38355	0.13332	0.73911	67.24463
8	8.000	175.74707	0.10729	0.84640	79.27548
9	9.000	125.49251	0.07661	0.92301	88.47083
10	10.000	76.40978	0.04665	0.96966	94.63390
11	11.000	36.97963	0.02258	0.99224	98.09512
12	12.000	12.71219	0.00776	1.00000	99.61196
/*****					

TABLE 10.20. CG Output for Log-linear Raw-score Equating: General Information

```

/*****
w1 = 1, log-linear smoothing

Equipercntile Equating with CINEG Design
(Internal Anchor; w1 = 1.00000)
and Polynomial Log-Linear Smoothing

Input file for XV and pop 1: mondatx-temp
Input file for YV and pop 2: mondaty-temp

NOTE: In this output:
(1) variables for new form and pop 1 are identified as
    X, X', and V, where  $X = X' + V$ ; i.e., X is the
        total score, X' is the non-common-item score, and
        V is the common-item score; and
(2) variables for old form and pop 2 are identified as
    Y, Y', and V, where  $Y = Y' + V$ ; i.e., Y is the
        total score, Y' is the non-common-item score, and
        V is the common-item score.

Number of persons for X =      1655

Number of score categories for X = 37
Number of score categories for V = 13
Polynomial degree for X' = 6
Polynomial degree for V = 6
Number of cross-products X'V = 1
    Cross-Product moments:    (X'1,V1)

Number of persons for Y =      1638

Number of score categories for Y = 37
Number of score categories for V = 13
Polynomial degree for Y' = 6
Polynomial degree for V = 6
Number of cross-products Y'V = 1
    Cross-Product moments:    (Y'1,V1)
*****/

```

TABLE 10.21. CG Output for Log-linear Raw-score Equating

/\*\*\*\*\*

Braun-Holland (BH) Under Frequency Estimation (FE):

Intercept: b = 0.66901; Slope: a = 1.01969

Braun-Holland (BH) Under Modified Frequency Estimation (MFE):

Intercept: b = 0.47626; Slope: a = 1.01105

-----

## Equated Raw Scores

Raw Score (X)	Method 0: FE	Method 1: BH-FE	Method 2: MFE	Method 3: BH-MFE	Method 4: ChainedE
0.00000	0.40298	0.66901	0.32362	0.47626	0.40298
1.00000	1.44788	1.68870	1.33664	1.48731	1.49509
2.00000	2.47108	2.70839	2.33955	2.49836	2.52393
3.00000	3.49452	3.72807	3.34634	3.50942	3.57274
4.00000	4.51156	4.74776	4.36033	4.52047	4.65156
5.00000	5.53402	5.76745	5.38108	5.53152	5.61257
6.00000	6.56610	6.78714	6.40737	6.54257	6.61851
7.00000	7.60814	7.80683	7.43822	7.55363	7.61083
8.00000	8.65987	8.82651	8.47295	8.56468	8.59701
9.00000	9.72038	9.84620	9.50951	9.57573	9.59672
10.00000	10.78798	10.86589	10.54724	10.58679	10.61276
11.00000	11.86034	11.88558	11.58977	11.59784	11.59306
12.00000	12.93449	12.90526	12.63492	12.60889	12.65051
13.00000	14.00709	13.92495	13.68007	13.61995	13.67647
14.00000	15.07466	14.94464	14.72237	14.63100	14.67110
15.00000	16.13379	15.96433	15.75894	15.64205	15.76228
16.00000	17.18150	16.98401	16.78715	16.65311	16.78226
17.00000	18.21545	18.00370	17.80492	17.66416	17.79040
18.00000	19.23414	19.02339	18.81092	18.67521	18.85980
19.00000	20.23707	20.04308	19.80476	19.68626	19.86008
20.00000	21.22485	21.06277	20.78713	20.69732	20.84392
21.00000	22.19916	22.08245	21.75979	21.70837	21.87843
22.00000	23.16275	23.10214	22.72550	22.71942	22.85841
23.00000	24.11919	24.12183	23.68791	23.73048	23.82005
24.00000	25.07266	25.14152	24.65119	24.74153	24.83335
25.00000	26.02752	26.16120	25.61971	25.75258	25.81315
26.00000	26.98775	27.18089	26.59741	26.76364	26.79401
27.00000	27.95636	28.20058	27.58727	27.77469	27.81693
28.00000	28.93467	29.22027	28.59047	28.78574	28.82482
29.00000	29.92144	30.23995	29.60570	29.79679	29.85995
30.00000	30.91231	31.25964	30.62826	30.80785	30.90777
31.00000	31.89955	32.27933	31.64968	31.81890	31.94582
32.00000	32.87286	33.29902	32.65789	32.82995	32.97272
33.00000	33.81948	34.31871	33.63720	33.84101	33.96998
34.00000	34.71637	35.33839	34.56045	34.85206	34.79821
35.00000	35.49077	36.35808	35.44045	35.86311	35.49173
36.00000	36.32130	37.37777	36.29567	36.87417	36.32130
Mean	16.80381	16.80103	16.47456	16.47167	16.54995
S.D.	6.64097	6.65634	6.58422	6.59998	6.62123
Skew	0.49382	0.57991	0.52947	0.57991	0.56677
Kurt	2.59898	2.72166	2.66421	2.72166	2.68401

/\*\*\*\*\*

TABLE 10.22. CG Output for Bootstrap Estimates of Standard Errors for Log-linear Raw-score Equating

*****										
Method 0: FE			Method 1: BH-FE		Method 2: MFE		Method 3: BH-MFE		Method 4: ChainE	
-----			-----		-----		-----		-----	
(x)	fd(x)	eraw	bse	eraw	bse	eraw	bse	eraw	bse	eraw
0	0.00000	0	0.40298	0.52747	0.66901	0.25276	0.32362	0.52932	0.47626	0.26070
1	0.00000	0	1.44788	0.92078	1.68870	0.24009	1.33664	0.92547	1.48731	0.24818
2	0.00000	1	2.47108	0.96743	2.70839	0.22762	2.33955	0.96698	2.49836	0.23589
3	0.00000	5	3.49452	0.60028	3.72807	0.21541	3.34634	0.63526	3.50942	0.22385
4	0.00000	9	4.51156	0.41756	4.74776	0.20349	4.36033	0.43848	4.52047	0.21211
5	0.00000	13	5.53402	0.33222	5.76745	0.19193	5.38108	0.34974	5.53152	0.20072
6	0.00000	36	6.56610	0.27213	6.78714	0.18078	6.40737	0.28722	6.54257	0.18975
7	0.00000	51	7.60814	0.22917	7.80683	0.17014	7.43822	0.24291	7.55363	0.17926
8	0.00000	77	8.65987	0.20641	8.82651	0.16009	8.47295	0.21691	8.56468	0.16935
9	0.00000	86	9.72038	0.19980	9.84620	0.15077	9.50951	0.20273	9.57573	0.16014
10	0.00000	92	10.78798	0.20280	10.86589	0.14231	10.54724	0.20222	10.58679	0.15174
11	0.00000	113	11.86034	0.20590	11.88558	0.13488	11.58977	0.20587	11.59784	0.14429
12	0.00000	128	12.93449	0.20520	12.90526	0.12865	12.63492	0.20597	12.60889	0.13796
13	0.00000	90	14.00709	0.20552	13.92495	0.12381	13.68007	0.20639	13.61995	0.13290
14	0.00000	113	15.07466	0.20965	14.94464	0.12052	14.72237	0.21078	14.63100	0.12925
15	0.00000	80	16.13379	0.20731	15.96433	0.11891	15.75894	0.21181	15.64205	0.12715
16	0.00000	97	17.18150	0.21358	16.98401	0.11905	16.78715	0.21661	16.65311	0.12667
17	0.00000	81	18.21545	0.22442	18.00370	0.12094	17.80492	0.22674	17.66416	0.12783
18	0.00000	73	19.23414	0.22867	19.02339	0.12449	18.81092	0.23389	18.67521	0.13058
19	0.00000	50	20.23707	0.23793	20.04308	0.12957	19.80476	0.24321	19.68626	0.13483
20	0.00000	73	21.22485	0.24738	21.06277	0.13600	20.78713	0.25188	20.69732	0.14043
21	0.00000	54	22.19916	0.25699	22.08245	0.14361	21.75979	0.26054	21.70837	0.14725
22	0.00000	54	23.16275	0.25688	23.10214	0.15222	22.72550	0.26372	22.71942	0.15510
23	0.00000	41	24.11919	0.26191	24.12183	0.16166	23.68791	0.26575	23.73048	0.16386
24	0.00000	35	25.07266	0.26468	25.14152	0.17181	24.65119	0.26563	24.74153	0.17338
25	0.00000	42	26.02752	0.27478	26.16120	0.18254	25.61971	0.27374	25.75258	0.18353
26	0.00000	31	26.98775	0.29268	27.18089	0.19376	26.59741	0.29233	26.76364	0.19423
27	0.00000	31	27.95636	0.29859	28.20058	0.20539	27.58727	0.29932	27.77469	0.20539
28	0.00000	23	28.93467	0.31218	29.22027	0.21736	28.59047	0.30934	28.78574	0.21693
29	0.00000	22	29.92144	0.33182	30.23995	0.22961	29.60570	0.32968	29.79679	0.22880
30	0.00000	17	30.91231	0.35089	31.25964	0.24211	30.62826	0.34949	30.80785	0.24095
31	0.00000	8	31.89955	0.37863	32.27933	0.25482	31.64968	0.37296	31.81890	0.25334
32	0.00000	14	32.87286	0.41816	33.29902	0.26771	32.65789	0.41220	32.82995	0.26593
33	0.00000	7	33.81948	0.46572	34.31871	0.28075	33.63720	0.45142	33.84101	0.27870
34	0.00000	6	34.71637	0.53883	35.33839	0.29393	34.56045	0.52603	34.85206	0.29163
35	0.00000	1	35.49077	0.57923	36.35808	0.30722	35.44045	0.60697	35.86311	0.30469
36	0.00000	1	36.32130	0.40378	37.37777	0.32062	36.29567	0.40983	36.87417	0.31787
Ave.		bse	0.24422		0.15185		0.24741		0.15782	
*** means bse = 0										
*****										



# 11

## Cubic Spline Postsmoothing

Postsmoothing methods adjust the equating function  $\hat{e}_Y(x)$ .<sup>1</sup> The post-smoothed equivalents should appear smooth without departing too much from the unsmoothed equivalents. While polynomials could be used for postsmoothing, cubic splines are preferred because of their flexibility and simplicity. Cubic spline smoothing dates back to Schönberg (1964). *Equating Recipes* implements the solution proposed by Reinsch (1967),<sup>2</sup> described by Kolen (1984), and briefly summarized by Kolen and Brennan (2004).

In this chapter, the discussion sometimes states (or appears to assume) that the scores under consideration are number-correct raw scores ranging from 0 to  $K_X$  for Form X or 0 to  $K_Y$  for Form Y. This simplifies some discussions. The methodology and the code, however, have no such restriction. In particular, as in previous chapters, raw scores may be any set of real numbers with a constant difference between scores in the ordered set of scores.<sup>3</sup>

---

<sup>1</sup>In practice, it is the raw score equating results that are adjusted by postsmoothing. Then if scale scores are under consideration, they are based on the postsmoothed equated raw scores. In principle, the scale scores could be smoothed directly; i.e., unsmoothed equated raw scores could be used to obtain scale scores, and then the scale scores could be smoothed.

<sup>2</sup>de Boor (1978) also discusses smoothing cubic splines, but there are slight differences between his approach and that of Reinsch (1967).

<sup>3</sup>Technically, the methodology applies in even more general cases.

## 11.1 Smoothing Cubic Splines

In psychometric applications, the smoothing cubic spline is fit over a restricted range of score points to avoid unnecessary influence from scores with few examinees or poorly estimated standard errors. In general, Kolen (1984) recommends that score points with percentile ranks below .5 or above 99.5 be excluded. Let  $x_{low} = x_0$  be the lowest score point and  $x_{high} = x_n$  be the highest score point in the restricted range of the Form X raw scores. Then, for the distribution of Form X raw scores,

$$x_{low} = x_0 < x_1 < \cdots < x_n = x_{high},$$

the *cubic spline function*  $g(x)$  is a function from the interval  $[x_0, x_n]$  to  $\mathbb{R}$ , the set of all real numbers. It is defined piecewise for each subinterval  $[x_i, x_{i+1}]$ ,  $i = 0, 1, 2, \dots, n-1$  so that

$$g(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad (11.1)$$

on  $[x_i, x_{i+1}]$  for some coefficients  $a_i, b_i, c_i$ , and  $d_i$ . These coefficients are allowed to differ from one subinterval to another subinterval with the constraint that  $g(x) \in C^2[x_0, x_n]$ , where  $C^2$  stands for the class of functions with continuous second derivatives. That is, the overall cubic spline function  $g(x)$  and its derivatives  $g'(x)$  and  $g''(x)$  are all continuous.

The *smoothing cubic spline function*  $d_Y(x)$  for  $\hat{e}_Y(x)$  over  $[x_0, x_n]$  is the cubic spline function that minimizes

$$\int_{x_0}^{x_n} [g''(x)]^2 dx \quad (11.2)$$

among all cubic splines  $g(x)$  that satisfy

$$\sum_{i=0}^n \left[ \frac{g(x_i) - \hat{e}_Y(x_i)}{\widehat{se}[\hat{e}_Y(x_i)]} \right]^2 \leq S. \quad (11.3)$$

Here,  $\widehat{se}[\hat{e}_Y(x_i)]$  is the estimated standard error of equipercentile equating at  $x_i$  and  $S \geq 0$  is a given number.

Note that Kolen and Brennan (2004, Equation 3.12) use a scaled version of Equation 11.3, namely,

$$\frac{1}{n+1} \sum_{i=0}^n \left[ \frac{g(x_i) - \hat{e}_Y(x_i)}{\widehat{se}[\hat{e}_Y(x_i)]} \right]^2 \leq S'. \quad (11.4)$$

This matters only for the functions discussed later that include an  $\mathbf{s}$  parameter, since  $\mathbf{s}$  is defined with respect to  $S'$  in Equation 11.4, not  $S$  in Equation 11.3.

If  $S = 0$ , then the cubic spline function  $g(x)$  interpolates  $\hat{e}_Y(x_i)$  at  $x = x_i$ . Thus, the smoothing cubic spline function  $d_Y(x)$  is the function that minimizes Equation 11.2 among all functions that interpolate  $\hat{e}_Y(x_i)$  at  $x_i$ . In general, for fixed  $S > 0$ , the larger  $\widehat{se}[\hat{e}_Y(x_i)]$ , the more  $g(x_i)$  could deviate from  $\hat{e}_Y(x_i)$ . In this sense, the  $\widehat{se}[\hat{e}_Y(x_i)]$  controls the deviation of  $d_Y(x_i)$  from  $\hat{e}_Y(x_i)$ . The constant  $S$  could be used to control all  $\widehat{se}[\hat{e}_Y(x_i)]$  simultaneously. Reinsch (1967) suggests the parameter  $S$  be chosen to belong to the interval

$$n+1 - \sqrt{2(n+1)} \leq S \leq n+1 + \sqrt{2(n+1)},$$



where  $n$  is the number of nodes in the data set. However, Kolen (1984) and Kolen and Brennan (2004) recommend choosing a final value for  $S'$  after examining the results for various values for  $S'$  in a specific equating context, rather than choosing  $S$  according to some a priori rule.

The smoothing cubic spline function  $d_Y(x)$  can be extended over the entire score range,  $[-0.5, K_X + 0.5]$ , using linear interpolation. Using the notation  $\hat{d}_Y(x)$  for this extension,  $\hat{d}_Y(x)$  is defined piecewise by

$$\hat{d}_Y(x) = \begin{cases} l_{low}(x + 0.5) - 0.5, & -0.5 \leq x < x_{low} \\ d_Y(x), & x_{low} \leq x \leq x_{high} \\ l_{high}(x - x_{high}) + d_Y(x_{high}), & x_{high} < x \leq K_X + 0.5 \end{cases} \quad (11.5)$$

where  $l_{low}$  is the slope of the linear line passing through  $(-0.5, -0.5)$  and  $(x_{low}, \hat{d}_X(x_{low}))$  and  $l_{high}$  is the slope of the linear line passing through  $(x_{high}, \hat{d}_X(x_{high}))$  and  $(K_X + 0.5, \hat{d}_X(K_X + 0.5))$ . They are given by

$$l_{low} = \frac{d_Y(x_{low}) + 0.5}{x_{low} + 0.5},$$

and

$$l_{high} = \frac{d_Y(x_{high}) - (K_Y + 0.5)}{x_{high} - (K_Y + 0.5)}.$$

One problem with  $\hat{d}_Y(x)$  is that it is not symmetric. A more nearly symmetric function can be defined as the average of two functions. That is,

$$\hat{d}_Y^*(x) = \frac{\hat{d}_Y(x) + \hat{d}_X^{-1}(x)}{2}, \quad -0.5 \leq x < K_X + 0.5,$$

where  $\hat{d}_Y(x)$  is the postsmoothing function to convert Form X to the Form Y scale and  $\hat{d}_X^{-1}(x)$  is the inverse of the postsmoothing function  $\hat{d}_X(y)$  to convert Form Y to the Form X scale.

### 11.1.1 Solution Method for Smoothing Cubic Splines

The optimal solution  $d_Y(x)$  that minimizes Equation 11.2 and satisfies Equation 11.3 is determined by finding the coefficients  $a_i$ ,  $b_i$ ,  $c_i$ , and  $d_i$  on each subinterval  $[x_i, x_{i+1}]$ ,  $i = 0, 1, 2, \dots, n-1$ . There are four coefficients on each subinterval and  $n$  subintervals. Thus, we have to find  $4n$  coefficients to determine  $d_Y(x)$ . Reinsch (1967) presents the solution using the methods of the calculus of variations. If we use the notation

$$\begin{aligned} d_Y^{(k)}(x_i)_+ &= \lim_{h \rightarrow 0+} d_Y^{(k)}(x_i + h), \\ d_Y^{(k)}(x_i)_- &= \lim_{h \rightarrow 0+} d_Y^{(k)}(x_i - h), \\ d_Y''(x_0)_- &= d_Y'''(x_0)_- = 0, \\ d_Y''(x_n)_+ &= d_Y'''(x_n)_+ = 0, \end{aligned}$$

the optimal solution  $d_Y(x)$  satisfies the following conditions:

$$d_Y(x_i)_- = d_Y(x_i)_+, \text{ for } i = 1, 2, \dots, n-1 \quad (11.6)$$

$$d_Y'(x_i)_- = d_Y'(x_i)_+, \text{ for } i = 1, 2, \dots, n-1 \quad (11.7)$$

$$d_Y''(x_i)_- = d_Y''(x_i)_+, \text{ for } i = 0, 1, 2, \dots, n \quad (11.8)$$

$$d_Y^{(3)}(x_i)_- = d_Y^{(3)}(x_i)_+ + 2p \frac{d_Y(x_i) - \widehat{e}_Y(x_i)}{\widehat{se}[\widehat{e}_Y(x_i)]^2}, \text{ for } i = 0, 1, \dots, n \quad (11.9)$$

$$d_Y^{(4)}(x) = 0, \text{ for } x_0 < x < x_n, \quad (11.10)$$

where the parameter  $p$  is the Lagrangian parameter that minimizes

$$\int_{x_0}^{x_n} \left( g''(x) \right)^2 dx + p \left\{ \sum_{i=0}^n \left[ \frac{g(x_i) - \widehat{e}_Y(x_i)}{\widehat{se}[\widehat{e}_Y(x_i)]} \right]^2 + z^2 - S \right\}$$

for the auxiliary variable  $z$ .

By inserting Equation 11.1 into Equations 11.6–11.9, we can obtain  $4n$  relations among the spline coefficients. These  $4n$  relations can be summarized as

$$c = \left( \mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T} \right)^{-1} p \mathbf{Q}^T y, \quad (11.11)$$

$$a = y - p^{-1} \mathbf{D}^2 \mathbf{Q} c, \quad (11.12)$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \text{ for } i = 0, 1, \dots, n-1, \quad (11.13)$$

$$b_i = \frac{a_{i+1} - a_i}{h_i} - c_i h_i - d_i h_i^2, \text{ for } i = 0, 1, \dots, n-1. \quad (11.14)$$

Here,  $c_0 = c_n = 0$  and

$$\begin{aligned} h_i &= x_{i+1} - x_i, \\ y &= (y_0, y_1, \dots, y_n)^T, \\ c &= (c_1, c_2, \dots, c_{n-1})^T, \\ a &= (a_0, a_1, \dots, a_n)^T. \end{aligned}$$

The matrix  $\mathbf{D}$  is the  $(n+1) \times (n+1)$  diagonal matrix defined by

$$\mathbf{D}_{i,j} = \begin{cases} \delta_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}, \quad (11.15)$$

for  $i, j = 0, 1, \dots, n$ . In matrix form,

$$\mathbf{D} = \begin{bmatrix} \delta_0 & 0 & \dots & 0 \\ 0 & \delta_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \delta_n \end{bmatrix}.$$

The matrix  $\mathbf{T}$  is the  $(n-1) \times (n-1)$  tridiagonal matrix whose non-zero entries are defined by

$$\mathbf{T}_{i,i} = \frac{2}{3}(h_{i-1} + h_i), \quad (11.16)$$

$$\mathbf{T}_{i,i+1} = \frac{h_i}{3}, \quad (11.17)$$

$$\mathbf{T}_{i+1,i} = \frac{h_i}{3}, \quad (11.18)$$

for  $i = 1, 2, \dots, n-1$ . In matrix form,

$$\mathbf{T} = \begin{bmatrix} \frac{2}{3}(h_0 + h_1) & \frac{h_1}{3} & \dots & 0 \\ \frac{h_1}{3} & \frac{2}{3}(h_1 + h_2) & \dots & 0 \\ 0 & \frac{h_2}{3} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{2}{3}(h_{n-2} + h_{n-1}) \end{bmatrix}.$$

The matrix  $\mathbf{Q}$  is the  $(n+1) \times (n-1)$  tridiagonal matrix whose non-zero entries are defined by

$$\mathbf{Q}_{i-1,i} = \frac{1}{h_{i-1}}, \quad (11.19)$$

$$\mathbf{Q}_{i,i} = -\left(\frac{1}{h_{i-1}} + \frac{1}{h_i}\right), \quad (11.20)$$

$$\mathbf{Q}_{i+1,i} = \frac{1}{h_i}, \quad (11.21)$$

for  $i = 1, \dots, n-1$ . In matrix form,

$$\mathbf{Q} = \begin{bmatrix} \frac{1}{h_0} & 0 & \dots & 0 \\ -\left(\frac{1}{h_0} + \frac{1}{h_1}\right) & \frac{1}{h_1} & \dots & 0 \\ \frac{1}{h_1} & -\left(\frac{1}{h_1} + \frac{1}{h_2}\right) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{h_{n-2}} \\ 0 & 0 & \dots & -\left(\frac{1}{h_{n-2}} + \frac{1}{h_{n-1}}\right) \\ 0 & 0 & \dots & \frac{1}{h_{n-1}} \end{bmatrix}. \quad (11.22)$$

These matrices  $\mathbf{D}$ ,  $\mathbf{T}$ , and  $\mathbf{Q}$  can be computed based on the input data  $x_i$ ,  $i = 0, 1, 2, \dots, n$ . In most psychometric applications, we can assume that  $h_i = x_{i+1} - x_i = 1$ ,  $i = 0, 1, 2, \dots, n-1$ , so the matrices  $\mathbf{D}$ ,  $\mathbf{T}$ ,  $\mathbf{Q}$  have simpler values than the ones given above. Once we find the Lagrangian parameter  $p$ , we can find  $c$  from Equation 11.11,  $a$  from Equation 11.12,  $d$  from Equation 11.13, and  $b$  from Equation 11.14. Reinsch (1967) proves that the parameter  $p$  is the solution of the following equation

$$F(p) = \sqrt{S}, \quad (11.23)$$

where

$$F(p) = \left\| \mathbf{DQ} \left( \mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T} \right)^{-1} \mathbf{Q}^T \mathbf{y} \right\|_2,$$

and the Euclidean norm  $\|\cdot\|_2$  for a vector  $x = (x_0, x_1, \dots, x_n)^T$  is defined by

$$\|x\|_2 = \sqrt{x_0^2 + x_1^2 + \dots + x_n^2}.$$

### 11.1.2 Description of the Algorithm

The algorithm consists of two parts. The first part is to compute the Lagrangian parameter  $p$  that satisfies Equation 11.23. To find  $p$ , we can use Newton's method. Let  $u = (\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T})^{-1} \mathbf{Q}^T y$ . Then Newton's method to compute  $p$  is given by

$$\begin{aligned} p_{k+1} &= p_k - \left( F(p_k) - \sqrt{S} \right) / \frac{d}{dp} F(p_k), \\ &= p_k - \left( (F(p_k))^2 - \sqrt{S} F(p_k) \right) / \left( F(p_k) \frac{d}{dp} F(p_k) \right), \end{aligned} \quad (11.24)$$

where

$$F^2(p_k) = u^T \mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} u, \quad (11.25)$$

and

$$\begin{aligned} F(p) \frac{d}{dp} F(p) &= u^T \mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} \frac{du}{dp}, \\ &= -u^T \mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} (\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T})^{-1} \mathbf{T} u, \\ &= p u^T \mathbf{T} (\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T})^{-1} \mathbf{T} u - u^T \mathbf{T} u. \end{aligned} \quad (11.26)$$

Since the matrix  $\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T}$  is a positive definite matrix for any  $p \geq 0$ , we can find the Cholesky decomposition  $\mathbf{R}^T \mathbf{R}$  of  $\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T}$ . Thus, the vector  $u = (\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T})^{-1} \mathbf{Q}^T y$  can be found by solving the linear system

$$\mathbf{R}^T \mathbf{R} u = \mathbf{Q}^T y. \quad (11.27)$$

The linear system specified by Equation 11.27 can be solved by a forward substitution with  $\mathbf{R}^T$  and a backward substitution with  $\mathbf{R}$ . For more details, see Golub (1996). Once we have the Lagrangian parameter  $p$ , the second part of the algorithm is to compute the coefficient vectors  $a$ ,  $b$ ,  $c$ , and  $d$  using Equation 11.11, Equation 11.12, Equation 11.13, and Equation 11.14. The algorithm to find  $p$  and coefficient vectors  $a$ ,  $b$ ,  $c$ , and  $d$  is given in Figure 11.1.

Lines 2 and 3 set up matrices  $\mathbf{Q}$ ,  $\mathbf{Q}^T$ ,  $\mathbf{T}$ , and compute matrices  $\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q}$  and  $\mathbf{Q}^T y$ . These matrices are used to compute  $u$  in lines 6 and 7. Lines 4 through 10 correspond to the first part to find the Lagrangian parameter  $p$ . In line 6, Cholesky decomposition  $\mathbf{R}^T \mathbf{R}$  of the positive definite matrix  $\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T}$  is computed and in line 7,  $u$  is computed using the decomposition. In line 8,  $F(p)^2$  is set to the variable  $e$  and  $u^T \mathbf{T} u$  is set to  $f$ . In line 9,  $u^T \mathbf{T} (\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T})^{-1} \mathbf{T} u$  is set to the variable  $g$ . Finally, the variable  $p_{new}$  is updated in line 10 using  $e$ ,  $f$ , and  $g$  according to Equation 11.24. Once we have the parameter  $p$ , the second part of the algorithm is to compute  $a$ ,  $b$ ,  $c$ , and  $d$  in line 12.

1. Set  $p = 1$  and  $p_{new} = 0$
2. Set up matrix  $\mathbf{Q}$ ,  $\mathbf{Q}^T$ , and  $\mathbf{T}$
3. Compute  $\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q}$  and  $\mathbf{Q}^T y$
4. While  $|p_{new} - p| > \text{error}$
5.      $p = p_{new}$
6.     Cholesky decompose  $\mathbf{Q}^T \mathbf{D}^2 \mathbf{Q} + p \mathbf{T}$  into  $\mathbf{R}^T \mathbf{R}$
7.     Compute  $u$  from  $\mathbf{R}^T \mathbf{R} u = \mathbf{Q}^T y$
8.     Set  $e = (\mathbf{D} \mathbf{Q} u)^T \mathbf{D} \mathbf{Q} u$  and  $f = u^T \mathbf{T} u$
9.     Compute  $g = w^T w$  where  $\mathbf{R}^T w = \mathbf{T} u$
10.    Set  $p_{new} = p + (e - \sqrt{Se}) / (f - pg)$
11. End while
12. Compute  $a, b, c, d$

FIGURE 11.1. Cubic spline algorithm.

## 11.2 Functions

The functions for cubic spline smoothing are in `CubicSpline.c`, with the corresponding function prototypes in `CubicSpline.h`. The function call structure is

- `Wrapper_Smooth_CubSpl()`
  - `postSmooth()`
  - \* `sspline()`
  - `inversePostSmooth()`
  - \* `sspline()`
- `Print_CubSpl()`

`sspline()` calls other functions not discussed here (see code for details).

### 11.2.1 `Wrapper_Smooth_CubSpl()`

The wrapper function for performing cubic spline postsmoothing is `Wrapper_Smooth_CubSpl()`:

```
void Wrapper_Smooth_CubSpl(char design,
                           struct PDATA *xtoy, struct ERAW_RESULTS *r_xtoy,
                           double *se_xtoy, struct CS_SMOOTH *cs_xtoy,
                           struct PDATA *ytox, struct ERAW_RESULTS *r_ytox,
                           double *se_ytox, struct CS_SMOOTH *cs_ytox,
                           double prlow, double prhigh, double s, int rep,
                           struct PDATA *inall, struct ERAW_RESULTS *r)
```

Listed below are the input variables and their descriptions for `Wrapper_Smooth_CubSpl()`. In these descriptions “X to Y” is an abbreviation for “putting X on the scale of Y”; similarly, “Y to X” is an abbreviation for “putting Y on the scale of X.”

- ▷ **design** = 'R', 'S', or 'C' (see discussion at the end of this section);
- ▷ **xtoy** = PDATA structure for  $X$  to  $Y$ ;
- ▷ **r\_xtoy** = ERAW\_RESULTS structure for  $X$  to  $Y$ ;
- ▷ **se\_xtoy** = vector containing standard errors for  $X$  to  $Y$ ;
- ▷ **cs\_xtoy** = CS\_SMOOTH structure for  $X$  to  $Y$ ;
- ▷ **ytox** = PDATA structure for  $Y$  to  $X$ ;
- ▷ **r\_ytox** = ERAW\_RESULTS structure for  $Y$  to  $X$ ;
- ▷ **se\_ytox** = vector containing standard errors for  $Y$  to  $X$ ;
- ▷ **cs\_ytox** = CS\_SMOOTH structure for  $Y$  to  $X$ ;
- ▷ **prlow** = percentile rank associated with lowest score used for cubic spline smoothing;
- ▷ **prhigh** = percentile rank associated with highest score used for cubic spline smoothing;
- ▷ **s** = smoothing or “flexibility” parameter (**s** is to be understood in the sense of Equation 11.4); and
- ▷ **rep** = replication number for bootstrap (should be set to 0 for actual equating).

The output variables are:

- ▷ **inall** = PDATA structure for the cubic spline postsmoothing; and
- ▷ **r** = ERAW\_RESULTS structure that stores equated raw scores and their moments for the cubic spline postsmoothing.

Note that **inall** does not include as members the **CS\_SMOOTH** structures associated with  $X$  to  $Y$  and  $Y$  to  $X$ ; rather, these **CS\_SMOOTH** structures are in the **PDATA** structures **xtoy** and **ytox**, respectively. Also, note that the cubic spline postsmoothed results in **r** are for the average of the  $X$  to  $Y$  equating and the inverse of the  $Y$  to  $X$  equating. (See the example in Section 11.3 for explicit details.)

The first parameter in **Wrapper\_Smooth\_CubSpl()** is **design**. Logically, the design type is not needed for cubic spline postsmoothing. However, there are several reasons for requiring it. First, every **PDATA** structure contains **design**, and it is best that it be identified so that the structure members provide a full description of the equating performed. Second, standard errors are required for cubic spline postsmoothing, and they are design specific. Third, knowing the design simplifies obtaining the range [**low**, **high**] within which the cubic spline is determined. This is because percentile ranks are stored in the **USTATS** and **BSTATS** structure. (Recall that RG uses two **USTATS** structures, SG uses one **BSTATS** structure, and CG uses two **BSTATS** structures.) Fourth, requiring the design parameter forces the user to be careful about the functions that are called prior to calling **Wrapper\_Smooth\_CubSpl()**. As discussed more fully below, **Wrapper\_Smooth\_CubSpl('R' ...)** must be paired with two calls to **Wrapper\_RN()**, **Wrapper\_Smooth\_CubSpl('S' ...)** must be paired with two calls to **Wrapper\_SN()**, and **Wrapper\_Smooth\_CubSpl('C' ...)** must be paired with two calls to **Wrapper\_RN()**.

For the 'R' design, two prior calls to `ReadRawGet_USTATS()` are required, one returning the address of a structure, say, `&x`, and the other returning the address of a structure, say, `&y`. One call to `Wrapper_RN()` should use `&x` followed by `&y`. The other call should use `&y` followed by `&x`.

For the CINEG design, two calls to `ReadRawGet_BSTATS()` are required, one returning the address of a structure, say, `&xv`, and the other returning the address of a structure, say, `&yv`. One call to `Wrapper_CN()` should use `&xv` followed by `&yv`. The other call should use `&yv` followed by `&xv`. The method for the  $X$  to  $Y$  and the  $Y$  to  $X$  CINEG designs must be the same. The method variable can be 'E', 'F', or 'C' as discussed in Section 6.5. Note that equivalents for only one type of equipercetile equating can be smoothed per call to `Wrapper_Smooth_CubSpl()`. This restriction considerably simplifies the code.

Care must be taken with the 'S' design. It is assumed here that there have been two calls to `ReadRawGet_BSTATS()`. The first call reads data in the order  $x$  then  $y$  with results stored in, say, `xy`. The second call reads data in order  $y$  then  $x$ , with results stored in a different structure, say, `yx`. Then `Wrapper_SN()` needs to be called twice, once using `&xy` with results stored in structures, say, `pdxy` and `rxy`, and once using `&yx` with results stored in structures, say, `pdyx` and `ryx`.

### 11.2.2 Print\_CubSpl()

The print function associated with `Wrapper_Smooth_CubSpl()` is:

```
void Print_CubSpl(FILE *fp, char tt[], struct PDATA *xtoy,
                  struct PDATA *ytox, struct PDATA *inall,
                  struct ERAW_RESULTS *r, int parm)
```

The input variables are:

- ▷ `fp` = output file pointer;
- ▷ `tt[]` = user-specified title;
- ▷ `xtoy` = PDATA structure for  $X$  to  $Y$ , which includes the associated CS\_SMOOTH structure;
- ▷ `ytox` = PDATA structure for  $Y$  to  $X$ , which includes the associated CS\_SMOOTH structure;
- ▷ `inall` = PDATA structure for the cubic spline results (the structure returned by `Wrapper_Smooth_CubSpl()`);
- ▷ `r` = ERAW\_RESULTS structure that stores equated raw scores and their moments for cubic spline postsmoothing (the structure returned by `Wrapper_Smooth_CubSpl()`); and
- ▷ `parm` = 0 (do not print cubic spline coefficients); = 1 (print coefficients for  $X$  to  $Y$ ); = 2 (print coefficients for both  $X$  to  $Y$  and  $Y$  to  $X$ ).

Again, note that the cubic spline postsmoothed results in `r` are for the average of the  $X$  to  $Y$  equating and the inverse of the  $Y$  to  $X$  equating.

### 11.2.3 Other Functions

The principal functions used by `Wrapper_Smooth_CubSpl()` are `sspline()`, `postSmooth()`, and `inversePostSmooth`. They are described next.

#### `sspline()`

`sspline()` uses  $x_i$ ,  $\hat{e}_Y(x_i)$ , and  $\hat{se}[\hat{e}_Y(x_i)]$ , for  $i = 0, 1, 2, \dots, n$  ( $n$  is `high - low`) as input along with a smoothing parameter value (as defined in Equation 11.4). The function computes the coefficients  $a_i$ ,  $b_i$ ,  $c_i$ , and  $d_i$  of the smoothing cubic spline. It calls `chsol()` to find the Cholesky factorization of the matrix as well as other basic linear algebra functions. A more detailed function call structure is given below:

```
void sspline(double *vectX, double *vectY, double *vectD,
             int num, double s, double *cmat)
```

The input variables are described next. It is important to understand that for this function “raw scores” means the `high - low + 1` scores in the `[low, high]` range.

- ▷ `vectX` = vector containing raw scores as elements;
- ▷ `vectY` = vector containing equated raw scores as elements;
- ▷ `vectD` = vector containing standard errors of equated raw scores as elements
- ▷ `num` = dimension of the vectors `vectX`, `vectY`, and `vectD`;
- ▷ `s` = fidelity constant, called the smoothing constant in Kolen and Brennan (2004), expressed as in Equation 11.4, not Equation 11.3;

The output variable is:

- ▷ `cmat` = vector of size `4*(num-1)` that contains the cubic spline coefficients  $a_i$ ,  $b_i$ ,  $c_i$ , and  $d_i$  (see Equation 11.1) ordered in the following manner:

$$\underbrace{a_0, \dots, a_{num-2}}_{num-1}, \underbrace{b_0, \dots, b_{num-2}}_{num-1}, \underbrace{c_0, \dots, c_{num-2}}_{num-1}, \underbrace{d_0, \dots, d_{num-2}}_{num-1}.$$

#### `postSmooth()`

The function `postSmooth()` implements the postsmoothing function  $\hat{d}_Y(x)$  using `sspline()`. That is, `postSmooth()` is for postsmoothing the equivalents that result from equating  $X$  to the scale of  $Y$ . `postSmooth()` is defined piecewise from  $-0.5$  to  $K_X + 0.5$  according to Equation 11.5. Note that for this function the phrase “raw scores” in the following variable descriptions is to be understood in the sense of non-negative integers from 0 to  $nsx - 1$  where  $nsx$  is the total number of score categories for  $X$ .



```
void postSmooth(double *xvalues, double *yvalues,
               double *dyi, int num1, double s, int xlow,
               int xhigh, double ky, double *vectX,
               int num2, double *vectY, double *cmat)
```

The input variables are:

- ▷ **xvalues** = vector containing raw scores for  $X$  as its elements;
- ▷ **yvalues** = vector containing equated raw scores on the scale of  $Y$ , i.e.,  $\hat{e}_Y(x)$ , as its elements;
- ▷ **dyi** = vector containing standard errors of equated raw scores on the scale of  $Y$ , i.e.,  $\hat{se}[\hat{e}_Y(x)]$ , as its elements.;
- ▷ **num1** = dimension of the vectors **xvalues**, **yvalues**, and **dyi** (i.e.,  $nsx$ );
- ▷ **s** = fidelity constant that is called the smoothing constant in Kolen and Brennan (2004), expressed as in Equation 11.4, not Equation 11.3;
- ▷ **xlow** = raw score associated with the lowest percentile rank for  $X$  (**prlow**) for obtaining cubic spline results (usually, **prlow** is 0.5);
- ▷ **xhigh** = raw score associated with the highest percentile rank for  $X$  (**prhigh**) for obtaining cubic spline results (usually, **prhigh** is 99.5);
- ▷ **ky** = number of possible score categories minus 1 that is associated with Form  $Y$ , which is  $K_Y$  in Kolen and Brennan (2004);
- ▷ **vectX** = vector containing  $x$  coordinates where  $\hat{d}_Y(x)$  is evaluated (for typical equating purposes **vectX** should be the same as **xvalues**);
- ▷ **num2** = dimension of vectors **vectX** and **vectY** (for typical equating purposes **num2** should be  $nsx$ );

The output variables are:

- ▷ **vectY** = vector containing  $\hat{d}_Y(x)$  values at **vectX**; and
- ▷ **cmat** = vector of size  $4*(\text{num1}-1)$  that contains the  $a_i$ ,  $b_i$ ,  $c_i$ , and  $d_i$  coefficients of the smoothing cubic spline  $d_Y(x)$  (see Equation 11.1) ordered in the following manner:

$$\underbrace{a_0, \dots, a_{\text{num1}-2}}_{\text{num1}-1} \underbrace{b_0, \dots, b_{\text{num1}-2}}_{\text{num1}-1} \underbrace{c_0, \dots, c_{\text{num1}-2}}_{\text{num1}-1} \underbrace{d_0, \dots, d_{\text{num1}-2}}_{\text{num1}-1}$$

**inversePostSmooth()**

The function **inversePostSmooth()** implements the inverse postsmoothing function  $\hat{d}_X^{-1}(x)$  defined piecewise from  $-0.5$  to  $K_Y + 0.5$ . That is, **postSmooth()** postsmooths the equivalents that result from equating  $Y$  to the scale of  $YX$ , and gets the inverse (i.e., the  $Y$  values) associated with the raw score values of  $X$ . Note that for this function the phrase “raw scores” in the following variable descriptions is to be understood in the sense of non-negative integers from 0 to  $nsy - 1$  where  $nsy$  is the total number of score categories for  $Y$ .

```
void inversePostSmooth(double *yvalues, double *xvalues,
                      double *dxi, int num1, double s,
                      int ylow, int yhigh, double kx,
                      double *vectX, int num2,
                      double *vectY, double *cmat)
```

The input variables are:

- ▷ **yvalues** = vector containing raw scores for  $Y$  as its elements;
- ▷ **xvalues** = vector containing equated raw scores on the scale of  $X$ , i.e.,  $\hat{e}_X(y)$ , as its elements;
- ▷ **dxi** = vector containing standard errors of equated raw scores on the scale of  $X$ , i.e.,  $\hat{se}[\hat{e}_X(y)]$ , as its elements;
- ▷ **num1** = dimension of the vectors **yvalues**, **xvalues**, and **dxi** (i.e.,  $nsy$ );
- ▷ **s** = fidelity constant that is called the smoothing constant in Kolen and Brennan (2004), expressed as in Equation 11.4, not Equation 11.3;
- ▷ **ylow** = raw score associated with the lowest percentile rank for  $Y$  (**prlow**) for obtaining cubic spline results (usually, **prlow** is 0.5);
- ▷ **yhigh** = raw score associated with the highest percentile rank for  $Y$  (**prhigh**) for obtaining cubic spline results (usually, **prhigh** is 99.5);
- ▷ **kx** = number of possible score categories minus 1 that is associated with Form Y, which is  $K_X$  in Kolen and Brennan (2004);
- ▷ **vectX** = vector containing  $X$  coordinates where  $\hat{d}_X^{-1}(x)$ , the inverse of  $\hat{d}_X(x)$ , is evaluated (usually the elements are  $0, 1, 2, \dots, nsx - 1$ ); and
- ▷ **num2** = dimension of vectors **vectX** and **vectY** (for typical equating purposes **num2** should be the number of score categories in  $X$ ).

The output variables are:

- ▷ **vectY** = vector containing  $\hat{d}_X^{-1}(x)$  values at **vectX**; and
- ▷ **cmat** = vector of size  $4*(num1-1)$  that contains the  $a_i$ ,  $b_i$ ,  $c_i$ , and  $d_i$  coefficients of the smoothing cubic spline  $d_X(y)$  ordered in the following manner:

$$\underbrace{a_0, \dots, a_{num1-2}}_{num1-1} \underbrace{b_0, \dots, b_{num1-2}}_{num1-1} \underbrace{c_0, \dots, c_{num1-2}}_{num1-1} \underbrace{d_0, \dots, d_{num1-2}}_{num1-1}$$

### 11.3 Random Groups Example

Table 11.1 is a **main()** function that illustrates cubic spline smoothing for the random groups design based on an example discussed by Kolen and Brennan (2004, pp. 84–97) with **s** = .2. This example is based on the same data set used to illustrate:

- linear equating (see Section 3.2 that begins on page 39),

- equipercentile equating (see Section 5.2 that begins on page 58),
- analytic standard errors (see Section 7.3 that begins on page 76),
- bootstrap standard errors (see Section 8.2 that begins on page 86),
- beta-binomial presmoothing (see Section 9.7 that begins on page 98), and
- log-linear presmoothing (see Section 10.5 that begins on page 124).

The preliminary steps to implement cubic spline smoothing, as illustrated in Table 11.1 are:

- read the raw data for Form X and store statistics in a `USTATS` structure `x`;
- read the raw data for Form Y and store statistics in a `USTATS` structure `y`;
- call `Wrapper_RN()` to get raw-score equivalents for  $X$  on the scale of  $Y$ ;
- call `Wrapper_RN()` to get raw-score equivalents for  $Y$  on the scale of  $X$ ;
- get the standard errors for  $X$  on the scale of  $Y$  (i.e., the standard errors of the  $Y$  equivalents); and
- get the standard errors for  $Y$  on the scale of  $X$  (i.e., the standard errors of the  $X$  equivalents).

Then, `Wrapper_Smooth_CubSpl()` produces the cubic spline raw score results, which are printed using `Print_CubSpl()`. Finally, the corresponding scale scores results are produced and printed by `Wrapper_ESS()` and `Print_ESS()`, respectively.

Probably the most challenging aspect of the code in Table 11.1 for cubic spline postsMOOTHING is keeping track of the various structures. The reader is advised to study carefully the structure declarations at the top of Table 11.1 and the calling sequence for `Wrapper_Smooth_CubSpl()`. Keep in mind that the final cubic spline results are based on two equatings, which (more than) doubles the number of structures involved. There are structures for equating  $X$  to  $Y$  (`pd_xtoy`, `r_xtoy`, and `cs_xtoy`), for equating  $Y$  to  $X$  (`pd_ytox`, `r_ytox`, and `cs_ytox`), and for the final results (`pd_CubSpl` and `r_CubSpl`).

Table 11.2 provides the cubic spline output for equating  $X$  to the scale of  $Y$ . Note that `low` = 5 since 5 is the lowest raw score that has a percentile rank greater than or equal to `prlow` = .5. Similarly, `high` = 39 since 39 is the highest raw score that has a percentile rank less than or equal to `prhigh` = 99.5. *Equating Recipes* obtains these results from the distribution of percentile ranks in the `USTATS` structure `&x` (see also Kolen & Brennan, 2004, p. 50). In short, the cubic spline nodes are associated with the raw scores in the range [5, 39].

Using the notational conventions in Kolen and Brennan (2004, pp. 86–88), the cubic spline coefficients in the [5, 38] range are the entries in the columns labelled `v0`, `v1`, `v2`, and `v3`, (the same as  $a_i$ ,  $b_i$ ,  $c_i$ , and  $d_i$  in previous parts of this chapter) with `v0` being the cubic spline smoothed equivalents associated with the raw scores. For a raw score of 39, `v0` is the sum of the coefficients for a raw score of 38. Outside the [5, 39] range, the values in the `v0` column are obtained by linear interpolation between  $-.5$  and  $3.48982$  (`v0` for 5) at the low end of the

score scale, and between 39.14107 (`v0` for 39) and 40.5 at the high end of the score scale.<sup>4</sup>

In Table 11.2, `uee` indicates the unsmoothed equipercntile equivalents. These are the same results as reported in Section 5.2. Also, `se` indicates standard errors, and these are the same results reported in Section 7.3. The entries in the column labelled `dev^2` are the squares of  $(v0-uee)/se$  (i.e., the squared terms in Equation 11.4, which is the same as Equation 3.12 in Kolen & Brennan, 2004).

Table 11.3 is a similarly formatted table for the equating of  $Y$  to the scale of  $X$ . Note in particular that `low` for  $Y$  is 4, whereas `low` for  $X$  is 5, as reported in Table 11.2.

Table 11.4 provides the final cubic spline results. The second column is identical to the `v0` column in Table 11.2, the third column is the inverse of the cubic spline smoothing associated with equating  $Y$  to the scale of  $X$  (see Table 11.3),<sup>5</sup> and the final column is the average of the second and third columns.

The results in the last column are the final results. They can be compared with the `s=.2` column in Table 3.7 in Kolen and Brennan (2004, p. 92). The differences are generally less than  $|.003|$ . For various reasons, the *Equating Recipes* results are likely to be more accurate, although the differences are clearly not very large for practical purposes.

Table 11.5 provides equated scale scores (both unrounded and rounded). They can be compared with the `s=.2` column in Table 3.10 in Kolen and Brennan (2004, p. 96). Again, the differences are not very large for practical purposes.

---

<sup>4</sup>More generally, interpolation occurs between `min-inc/2` and `score(low,min,inc)` (on the low end of the score scale), and between `score(high,min,inc)` and `max+inc/2` (on the high end of the score scale).

<sup>5</sup>The process of obtaining the inverse is as follows: (a) get the cubic-spline smoothed equipercntile equivalents for the equating of  $Y$  to the scale of  $X$ ; and (b) for the  $X$  equivalents associated with the actual raw scores, get the corresponding  $Y$  score. Step (b) requires solving a non-linear equation. Newton's method could be used to do so, but *Equating Recipes* uses bisection.

TABLE 11.1. Main() Code to Illustrate Cubic Spline Postsmoothing with Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct PDATA pd_xtoy, pd_ytox, pd_CubSpl;
    struct ERAW_RESULTS r_xtoy, r_ytox, r_CubSpl;
    struct ESS_RESULTS s_CubSpl;
    struct CS_SMOOTH cs_xtoy, cs_ytox;

    double se_xtoy[41],se_ytox[41];          /* analytic se's */

    FILE *outf;

    outf = fopen("Chap 11 out","w");

    /* Cubic-Spline Postsmoothing with Random Groups Design:
       Kolen and Brennan (2004, chap. 2, pp. 84-97) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X',&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y',&y);

    Wrapper_RN('R','E','N',&x,&y,0,&pd_xtoy,&r_xtoy);
    Wrapper_RN('R','E','N',&y,&x,0,&pd_ytox,&r_ytox);

    SE_EPequate(y.ns, y.n, y.crfd, x.ns, x.inc, x.n, x.prd, se_xtoy);
    SE_EPequate(x.ns, x.n, x.crfd, y.ns, y.inc, y.n, y.prd, se_ytox);

    /* get and print raw-score cubic spline results */

    Wrapper_Smooth_CubSpl('R',&pd_xtoy, &r_xtoy, se_xtoy, &cs_xtoy,
                          &pd_ytox, &r_ytox, se_ytox, &cs_ytox,
                          .5, 99.5, .2, 0,
                          &pd_CubSpl, &r_CubSpl);

    Print_CubSpl(outf, "ACT Math---Equipercentile---Cubic Spline",
                 &pd_xtoy, &pd_ytox, &pd_CubSpl, &r_CubSpl, 2);

    /* get and print scale-score cubic spline results */

    Wrapper_ESS(&pd_CubSpl,&r_CubSpl,0,40,1,"yctmath.TXT",1,1,36,&s_CubSpl);
    Print_ESS(outf,"ACT Math---Equipercentile---Cubic Spline",
              &pd_CubSpl,&s_CubSpl);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 11.2. Output for Equated Raw Scores for Cubic Spline Smoothing that puts  $X$  on the Scale of  $Y$ 

```

/*****
ACT Math---Equipercentile---Cubic Spline

Cubic-Spline Postsmoothing Based on Random Groups Design

Putting X on scale of Y

Input file for x: actmathfreq.dat
Input file for y: actmathfreq.dat

s = 0.20000

Number of raw-score categories = 41
prlow = 0.50000; low = 5.00000 (in location 5)
prhigh = 99.50000; high = 39.00000 (in location 39)

Raw Sc X      uee      v0      v1      v2      v3      se      dev^2

0.00000  0.00000 -0.13729
1.00000  0.97956  0.58813
2.00000  1.64622  1.31355
3.00000  2.28563  2.03897
4.00000  2.89320  2.76440
5.00000  3.62047  3.48982  0.94573  0.00000  0.00043  0.14781  0.78132
6.00000  4.49965  4.43599  0.94704  0.00130  0.00051  0.25411  0.06278
7.00000  5.51484  5.38483  0.95116  0.00282  0.00088  0.15818  0.67547
8.00000  6.31242  6.33970  0.95946  0.00547  0.00083  0.19691  0.01920
9.00000  7.22424  7.30546  0.97290  0.00797  0.00064  0.17612  0.21269
10.00000  8.16067  8.28697  0.99076  0.00990  0.00034  0.17312  0.53227
11.00000  9.18270  9.28796  1.01156  0.01090  0.00014  0.19516  0.29091
12.00000  10.18590  10.31056  1.03377  0.01131 -0.00014  0.17995  0.47989
13.00000  11.25130  11.35549  1.05596  0.01088 -0.00029  0.23109  0.20328
14.00000  12.38963  12.42204  1.07685  0.01002 -0.00033  0.24312  0.01777
15.00000  13.39289  13.50858  1.09591  0.00904 -0.00051  0.21385  0.29267
16.00000  14.52401  14.61302  1.11246  0.00751 -0.00059  0.27635  0.10374
17.00000  15.71690  15.73239  1.12569  0.00573 -0.00061  0.26173  0.00350
18.00000  16.82344  16.86320  1.13532  0.00389 -0.00064  0.33835  0.01381
19.00000  18.00922  18.00178  1.14120  0.00199 -0.00063  0.28261  0.00069
20.00000  19.16472  19.14433  1.14328  0.00010 -0.00061  0.29473  0.00479
21.00000  20.36760  20.28710  1.14164 -0.00174 -0.00056  0.32987  0.05955
22.00000  21.45563  21.42645  1.13649 -0.00341 -0.00054  0.31827  0.00841
23.00000  22.68712  22.55899  1.12805 -0.00503 -0.00048  0.38646  0.10993
24.00000  23.91566  23.68154  1.11658 -0.00645 -0.00034  0.35546  0.43378
25.00000  25.02916  24.79133  1.10265 -0.00747 -0.00015  0.30133  0.62295
26.00000  26.16123  25.88635  1.08725 -0.00792 -0.00000  0.36831  0.55697
27.00000  27.26329  26.96568  1.07139 -0.00793  0.00017  0.35323  0.70987
28.00000  28.18006  28.02931  1.05604 -0.00742  0.00029  0.30691  0.24129
29.00000  29.14243  29.07821  1.04205 -0.00656  0.00033  0.34220  0.03523
30.00000  30.13048  30.11402  1.02990 -0.00559  0.00034  0.28963  0.00323
31.00000  31.12970  31.13867  1.01975 -0.00456  0.00033  0.32680  0.00075
32.00000  32.13571  32.15418  1.01162 -0.00356  0.00032  0.33093  0.00312
33.00000  33.07807  33.16256  1.00546 -0.00260  0.00026  0.30477  0.07686
34.00000  34.01719  34.16568  1.00104 -0.00183  0.00014  0.30798  0.23248
35.00000  35.10160  35.16504  0.99781 -0.00140  0.00009  0.30435  0.04344
36.00000  36.24255  36.16154  0.99529 -0.00112  0.00015  0.32400  0.06251
37.00000  37.12476  37.15586  0.99349 -0.00068  0.00012  0.27137  0.01313
38.00000  38.13209  38.14879  0.99249 -0.00032  0.00011  0.34301  0.00237
39.00000  39.08073  39.14107
40.00000  39.90055  40.04702
0.20179  0.08939
0.27872

*****/

```

TABLE 11.3. Output for Equated Raw Scores for Cubic Spline Smoothing that puts  $Y$  on the Scale of  $X$ 

```

/*****
Putting Y on scale of X

s =    0.20000

Number of raw-score categories = 41
prlow = 0.50000; low = 4.00000 (in location 4)
prhigh = 99.50000; high = 39.00000 (in location 39)

Raw Sc Y      uee      v0      v1      v2      v3      se      dev^2

0.00000  0.00000  0.19217
1.00000  1.02132  1.57651
2.00000  2.70219  2.96086
3.00000  4.16085  4.34520
4.00000  5.62915  5.72954  1.00931  0.00000 -0.00023  0.14536  0.47694
5.00000  6.51900  6.73862  1.00862 -0.00068 -0.00049  0.20202  1.18178
6.00000  7.65995  7.74607  1.00579 -0.00214 -0.00059  0.19847  0.18828
7.00000  8.78731  8.74912  0.99973 -0.00392 -0.00052  0.15894  0.05772
8.00000  9.82237  9.74441  0.99033 -0.00548 -0.00041  0.18702  0.17372
9.00000  10.84027  10.72886  0.97815 -0.00671 -0.00022  0.16662  0.44709
10.00000  11.80416  11.70008  0.96407 -0.00737 -0.00008  0.18646  0.31159
11.00000  12.77620  12.65670  0.94910 -0.00760  0.00006  0.20233  0.34880
12.00000  13.66450  13.59827  0.93410 -0.00740  0.00014  0.20503  0.10433
13.00000  14.60442  14.52511  0.91972 -0.00698  0.00023  0.21204  0.13988
14.00000  15.55656  15.43808  0.90644 -0.00631  0.00036  0.20672  0.32848
15.00000  16.36135  16.33857  0.89490 -0.00523  0.00038  0.21053  0.01171
16.00000  17.27560  17.22862  0.88559 -0.00408  0.00042  0.25564  0.03377
17.00000  18.13206  18.11054  0.87868 -0.00283  0.00043  0.25260  0.00726
18.00000  18.99106  18.98682  0.87432 -0.00153  0.00044  0.27389  0.00024
19.00000  19.87098  19.86005  0.87257 -0.00022  0.00045  0.23056  0.00225
20.00000  20.65062  20.73284  0.87347  0.00112  0.00041  0.31464  0.06828
21.00000  21.60072  21.60783  0.87692  0.00234  0.00040  0.28043  0.00064
22.00000  22.42907  22.48749  0.88280  0.00354  0.00037  0.27603  0.04480
23.00000  23.24411  23.37420  0.89098  0.00464  0.00030  0.29890  0.18944
24.00000  24.07286  24.27012  0.90115  0.00552  0.00019  0.30610  0.41527
25.00000  24.97270  25.17699  0.91278  0.00611  0.00007  0.28235  0.52349
26.00000  25.86125  26.09595  0.92521  0.00632 -0.00004  0.31841  0.54331
27.00000  26.74735  27.02743  0.93773  0.00620 -0.00015  0.34267  0.66804
28.00000  27.78598  27.97120  0.94966  0.00574 -0.00022  0.36935  0.25149
29.00000  28.86410  28.92638  0.96047  0.00508 -0.00025  0.32911  0.03580
30.00000  29.86396  29.89168  0.96988  0.00433 -0.00026  0.30537  0.00824
31.00000  30.86923  30.86564  0.97777  0.00355 -0.00026  0.33307  0.00012
32.00000  31.86362  31.84669  0.98408  0.00277 -0.00025  0.33679  0.00253
33.00000  32.91191  32.83329  0.98886  0.00201 -0.00022  0.34775  0.05112
34.00000  33.98258  33.82393  0.99221  0.00134 -0.00014  0.31320  0.25658
35.00000  34.91384  34.81734  0.99447  0.00091 -0.00008  0.26140  0.13628
36.00000  35.75723  35.81264  0.99606  0.00068 -0.00010  0.33852  0.02680
37.00000  36.83476  36.80929  0.99713  0.00038 -0.00009  0.37499  0.00462
38.00000  37.89490  37.80670  0.99762  0.00011 -0.00004  0.28067  0.09874
39.00000  38.88289  38.80440  0.31889  0.06058
40.00000  40.08295  39.93480  0.20125

*****/

```

TABLE 11.4. Output for Equated Raw Scores for Cubic Spline Smoothing

/\*\*\*\*\*/  
Cubic Spline Results

In the following table  $dY(x)$  represents cubic spline smoothed equivalents for putting  $X$  on the scale of  $Y$ ,  $dX^{-1}(x)$  represents the inverse of the cubic spline for putting  $Y$  on the scale of  $X$ , and "average" is the average of the two.  
(see Kolen & Brennan, 2004, pp. 84-97)

Raw Score (x)	$dY(x)$	$dX^{-1}(x)$	average
0.00000	-0.13729	-0.13882	-0.13805
1.00000	0.58813	0.58355	0.58584
2.00000	1.31355	1.30591	1.30973
3.00000	2.03897	2.02828	2.03363
4.00000	2.76440	2.75064	2.75752
5.00000	3.48982	3.47301	3.48141
6.00000	4.43599	4.26797	4.35198
7.00000	5.38483	5.25920	5.32202
8.00000	6.33970	6.25262	6.29616
9.00000	7.30546	7.25120	7.27833
10.00000	8.28697	8.25846	8.27271
11.00000	9.28796	9.27773	9.28284
12.00000	10.31056	10.31185	10.31120
13.00000	11.35549	11.36276	11.35913
14.00000	12.42204	12.43153	12.42679
15.00000	13.50858	13.51834	13.51346
16.00000	14.61302	14.62252	14.61777
17.00000	15.73239	15.74216	15.73727
18.00000	16.86320	16.87424	16.86872
19.00000	18.00178	18.01507	18.00842
20.00000	19.14433	19.16040	19.15236
21.00000	20.28710	20.30573	20.29641
22.00000	21.42645	21.44664	21.43654
23.00000	22.55899	22.57912	22.56906
24.00000	23.68154	23.69970	23.69062
25.00000	24.79133	24.80585	24.79859
26.00000	25.88635	25.89622	25.89129
27.00000	26.96568	26.97074	26.96821
28.00000	28.02931	28.03032	28.02981
29.00000	29.07821	29.07662	29.07741
30.00000	30.11402	30.11163	30.11282
31.00000	31.13867	31.13735	31.13801
32.00000	32.15418	32.15572	32.15495
33.00000	33.16256	33.16854	33.16555
34.00000	34.16568	34.17741	34.17155
35.00000	35.16504	35.18364	35.17434
36.00000	36.16154	36.18807	36.17481
37.00000	37.15586	37.19125	37.17356
38.00000	38.14879	38.19375	38.17127
39.00000	39.14107	39.17304	39.15705
40.00000	40.04702	40.05768	40.05235
Mean			18.97155
S.D.			8.89787
Skew			0.36461
Kurt			2.19308

/\*\*\*\*\*/



TABLE 11.5. Output for Unrounded Equated Scale Scores using Cubic Spline Smoothing

```

/*****
ACT Math---Equipercntile---Cubic Spline

Name of file containing yct[] []: yctmath.TXT

Equated Scale Scores (unrounded)

Raw Score (x)      Method 0:
                   equiv

0.00000            0.50000
1.00000            0.50000
2.00000            0.50000
3.00000            0.50000
4.00000            0.50000
5.00000            0.50000
6.00000            0.56688
7.00000            1.00113
8.00000            2.08622
9.00000            3.55044
10.00000           5.08533
11.00000           6.50325
12.00000           7.91744
13.00000           9.41730
14.00000          10.89385
15.00000          12.24807
16.00000          13.56073
17.00000          14.89822
18.00000          16.20313
19.00000          17.39047
20.00000          18.48415
21.00000          19.55102
22.00000          20.53785
23.00000          21.39605
24.00000          22.12957
25.00000          22.79780
26.00000          23.45302
27.00000          24.11191
28.00000          24.76857
29.00000          25.32535
30.00000          25.77662
31.00000          26.22166
32.00000          26.73950
33.00000          27.34935
34.00000          28.04266
35.00000          28.87662
36.00000          30.00249
37.00000          31.45967
38.00000          32.94896
39.00000          34.39407
40.00000          35.57023

Mean              16.53407
S.D.              8.32753
Skew              -0.12885
Kurt              2.09164
*****/

```



# 12

## Kernel Equating

The kernel method of equating was first proposed by Holland and Thayer (1989), and was recently described more fully by von Davier, Holland and Thayer (2004). The term “kernel equating” does not refer to a single equating method for a particular equating design, but rather to a framework of methods that apply to nearly all the different equating designs. The essential building blocks of this framework include the log-linear smoothing procedure which is applied to a univariate or bivariate distribution of discrete scores and a Gaussian kernel procedure that transforms two discrete score distributions into two continuous probability distributions. Equipercentile equating is then carried out on the continuous distributions. The steps in between vary according to different equating designs, but are all encapsulated in a concept called the Design Function. The authors claim that this framework of equating methods has some advantages over the traditional percentile rank (PR)-based equipercentile equating methods. Below is a more detailed description of the framework.<sup>1</sup>

### 12.1 The Kernel Equating Framework

The kernel equating framework in von Davier et al. (2004) can be described by decomposing the equating process into the following five steps (von Davier et al., 2004, pp. 45–47):

---

<sup>1</sup>The *Equating Recipes* code for the kernel method is somewhat newer and less tested than code for more traditional methods. We anticipate, therefore, that the code may be improved over time.

*Step 1: Pre-smoothing.* This step estimates score distributions for the populations from which test data were collected. Typically, the log-linear smoothing procedure (Holland & Thayer, 1987; Rosenbaum & Thayer, 1987) is used to carry out this step because of its flexibility and desirable statistical properties such as preserving the moments. For the random groups (RG) design (called the equivalent groups, EG, design by von Davier et al., 2004), a univariate log-linear smoothing procedure is used. For the single groups (SG) design, the counter-balanced (CB) design, and the common-item nonequivalent groups (CINEG) design (called the non-equivalent groups with anchor test, NEAT, design by von Davier et al., 2004), the bivariate log-linear smoothing procedure is used. The resulting distributions are smoothed discrete distributions.

*Step 2: Estimating score probabilities.* In this step, the univariate or bivariate score distributions from the previous step are transformed to the marginal distributions of the new and old test forms for a common target population. von Davier et al. (2004) used the terminology “Design Function” to connote this transformation process. For different data collection designs, different Design Functions will be used. For example, for the RG design, an identity function is used as the Design Function, which effectively means doing nothing for this step. For the other designs, the Design Function can be complicated. They are all described in Chapter 3 of von Davier et al. (2004).

*Step 3: Continuization.* This step transforms the discrete score distributions from the previous step into two continuous distributions. The PR function in traditional equipercentile equating can be viewed as a special type of continuous function because it is a piece-wise linear continuous function. von Davier et al. (2004) proposed using the Gaussian kernel method to fit a continuous distribution to the discrete score distribution, which is described in more detail later in this section. (As discussed more fully in Chapter 14, recently Wang (2007, 2008) proposed an alternative continuization method for this step.)

*Step 4: Equating.* This step performs equipercentile equating using the continuous distributions from Step 3. Denote the random variables for the test scores for Form X as  $X$  and for Form Y as  $Y$ , and denote cumulative distributions of  $X$  and  $Y$  for the target population as  $F(X)$  and  $G(Y)$ . Then the equipercentile equating function is

$$e_Y(X) = G^{-1}(F(X)), \quad (12.1)$$

*Step 5: Calculating the Standard Error of Equating.* With this general framework and the kernel continuization method, von Davier et al. (2004) derived an elegant general formula for estimating the standard error of equating (SEE) based on the  $\delta$ -method. This general formula can be applied to all equating designs and is composed of three components with each of them relating to a different step in

this process. Wang (2007, 2008) described procedures for computing the SEE for alternative continuization procedures.

In Step 3, von Davier et al. (2004) used the Gaussian kernel method to fit a continuous distribution to an already smoothed discrete distribution. The Gaussian kernel method is known to distort moments higher than the first (i.e., the mean). In order to preserve the first two moments, von Davier et al. (2004) proposed an adjusted Gaussian kernel method described below.

Let  $x_j, j = 0, \dots, J$  denote the score points for  $X$ ,  $r_j$  be the discrete relative frequency at score  $x_j$ ,  $\Phi$  and  $\phi$  denote the cdf and pdf of the standard normal distribution,  $h_X$  be a bandwidth parameter, and  $\mu_X$  and  $\sigma_X^2$  denote the mean and variance of  $X$  over target population  $T$ . The von Davier et al. (2004) Gaussian kernel has a distribution with the cdf given by

$$F_{h_X}(x) = \sum_j r_j \Phi(R_{jX}(x)), \quad (12.2)$$

where

$$R_{jX}(x) = \frac{x - a_X x_j - (1 - a_X)\mu_X}{a_X h_X}, \quad (12.3)$$

and

$$a_X^2 = \frac{\sigma_X^2}{\sigma_X^2 + h_X^2}. \quad (12.4)$$

The pdf of this kernel distribution is

$$f_{h_X}(x) = \sum_j r_j \phi(R_{jX}(x)) \frac{1}{a_X h_X}, \quad (12.5)$$

which has the same mean and standard deviation as the discrete distribution of  $X$ . The skewness and kurtosis, however, will be different by a factor of  $(a_X)^j$ , where  $j = 3$  for skewness and  $j = 4$  for kurtosis. Note that  $a_X$  is a number between 0 and 1. As  $h$  increases,  $a_X$  decreases. This means that a larger  $h$  will result in a more symmetric distribution with smaller kurtosis, which implies more distortion of shape in the kernel distribution compared to the discrete distribution.

von Davier et al. (2004, pp. 62–63) introduced two penalty functions to find the optimal  $h$  parameter that minimizes some weighted combination of these penalty functions. The first penalty function is defined as:

$$PEN_1(h_X) = \sum_j \left( \hat{r}_j - \hat{f}_{h_X}(x_j) \right)^2. \quad (12.6)$$

The second penalty function is defined as:

$$PEN_2(h_X) = \sum_j A_j (1 - B_j), \quad (12.7)$$

where  $A_j = 1$  if  $f'_{h_X}(x) < 0$  a little to the left of  $x_j$ , and  $B_j = 0$  if  $f'_{h_X}(x) > 0$  a little to the right of  $x_j$ . This penalty function has a value of 1 at a score point  $x_j$  if  $f_{h_X}(x)$  is “U-shaped” around it.

The combined penalty function is a weighted sum of the two penalty functions described above:

$$PEN(h_X) = PEN_1(h_X) + K \times PEN_2(h_X). \quad (12.8)$$

$K$  can be set to 1 because the value from the first penalty function is quite small, and any non zero value from the second penalty function will dominate the combined penalty function.

## 12.2 Functions for the Random Groups Design

The wrapper and print functions for kernel equating with the random groups design are in `Kernel_Equate.c`, with the function prototypes in `Kernel_Equate.h`.

The wrapper function for the random groups design is `Wrapper_RK()`. It calls `KernelEquateSEERG()`, which in turn calls `KernelEquate()`. Then `Wrapper_RK()` calls `MomentsFromFD()`, which is described on page 20. After `Wrapper_RK()` returns to `main()`, `Print_RK()` can be called. So, the calling sequence is

- `Wrapper_RK()`
  - `EqKernelEquateSEERG()`
  - \* `KernelEquate()`
  - `MomentsFromFD()`
- `Print_RK()`

`Wrapper_RK()` takes a few arguments which are structures. It then passes many data fields to the functions `KernelEquateSEERG()` and `KernelEquate()`, which do the main computational work.

### 12.2.1 `Wrapper_RK()`

`Wrapper_RK()` calls functions that performs kernel equating for the random groups design. The function prototype for `Wrapper_RK()` is:

```
void Wrapper_RK(char design, char method, char smoothing,
                struct USTATS *x, struct USTATS *y,
                struct ULL_SMOOTH *ullx,
                struct ULL_SMOOTH *ully, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' designating the random groups design;
- ▷ `method` = 'E' for equipercentile equating;
- ▷ `smoothing` = 'K' for kernel 'smoothing';
- ▷ `x` = pointer to a `USTATS` (new form);
- ▷ `y` = pointer to a `USTATS` (old form);
- ▷ `ullx` = pointer to a `ULL_SMOOTH` structure (new form);
- ▷ `ully` = pointer to a `ULL_SMOOTH` structure (old form);
- ▷ `rep` = replication number for bootstrap; should be set to 0 for actual equating;;

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_RK()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

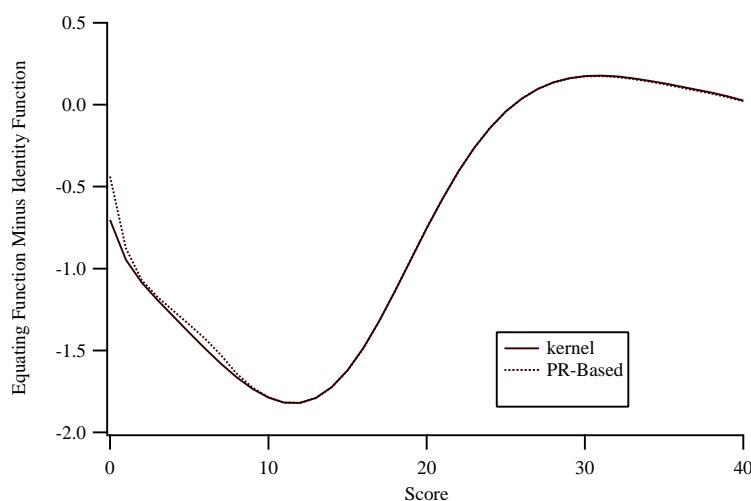


FIGURE 12.1. Kernel and percentile-rank based equipercentile equating under a RG Design.

### 12.2.2 Print\_RK()

The function prototype for `Print_RK()` is:

```
void Print_RK(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the kernel equating results for the random groups design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

### 12.2.3 Example

Table 12.1 is a `main()` function that illustrates kernel equating with the random groups design based on an example discussed by Kolen and Brennan (2004, chap. 2, p. 48). In this example, `Wrapper_Smooth_ULL()` was called to perform univariate log-linear smoothing for both the  $X$  and  $Y$  distributions, and then `Wrapper_RK()` was called to perform kernel equating.

The output from `Print_RK()` is provided in Table 12.2. The top of the output provides general information about the log-linear smoothing. The bottom part provides the equated raw scores for kernel equating, as well as their moments based on using the new-form frequencies. Figure 12.1 plots the kernel equating results together with the percentile rank based equipercentile equating results.

Note that the lowest equated raw score is  $-.70313$ . Recall that, for number-correct scoring (as is the case in this example), the lowest possible equated score using traditional percentile-rank procedures is  $-.5$  (see page 22 and the comments in the code for `perc_point()`). This inconsistency arises because for kernel equating with a Gaussian kernel there are no lower-limit or upper-limit constraints; i.e., a normal distribution is defined over  $(-\infty, +\infty)$ .

## 12.3 Functions for the Single Group Design

The wrapper function for kernel equating under the single group design is `Wrapper_SK()`. It calls `KernelEquateSG()`, and then calls `MomentsFromFD()`. After `Wrapper_SK()` returns to `main()`, `Print_SK()` can be called. So, the calling sequence is

- `Wrapper_SK()`
  - `KernelEquateSG()`
  - `MomentsFromFD()`
- `Print_SK()`

### 12.3.1 `Wrapper_SK()`

`Wrapper_SK()` calls functions that performs kernel equating for the random groups design. The function prototype for `Wrapper_SK()` is:

```
void Wrapper_SK(char design, char method, char smoothing,
                struct BSTATS *xy, struct BLL_SMOOTH *bllxy,
                int rep, struct PDATA *inall,
                struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'S' designating the single group design;
- ▷ `method` = 'E' for equipercentile equating;
- ▷ `smoothing` = 'K' for kernel 'smoothing';
- ▷ `xy` = pointer to a `BSTATS` (both forms);
- ▷ `bllxy` = pointer to a `BLL_SMOOTH` structure (both form); and
- ▷ `rep` = replication number for bootstrap; should be set to 0 for actual equating.

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_SK()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

`Wrapper_SK()` populates the elements of the `PDATA` structure `inall`, dynamically allocates space for the `eraw[][]` and `mts[][]` matrices in the `ERAW_RESULTS` structure `r`, calls a function that does the actual equating, stores results in `r`, and calls the function `MomentsFromFD()` that gets the equated raw-score moments and stores them in `r`.



### 12.3.2 Print\_SK()

The function prototype for `Print_SK()` is:

```
void Print_SK(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the kernel equating results for the single group design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

### 12.3.3 Example

Table 12.3 is a `main()` function that illustrates kernel equating with the single group design based on a dummy example using data in Kolen and Brennan (2004, chap. 4, p. 121). In this example, the `Wrapper_Smooth_BLL()` was called to perform bivariate log-linear smoothing for the joint distribution of  $X$  and  $Y$ , and then `Wrapper_SK()` was called to perform kernel equating.

The output from `Print_SK()` is provided in Table 12.4. The top of the output provides general information about the log-linear smoothing. The bottom part provides the equated raw scores for kernel equating, as well as their moments based on using the new-form frequencies.

## 12.4 Functions for the Common-Item Nonequivalent Groups Design

The wrapper function is `Wrapper_CK()`. Depending on the `method` argument, it calls `KernelEquateNEATPS()` or `KernelEquateNEATChn()`. Then `Wrapper_CK()` calls `MomentsFromFD()`. After `Wrapper_RK()` returns to `main()`, `Print_RK()` can be called. So, the calling sequence is

- `Wrapper_CK()`
  - `KernelEquateNEATChn()`, or
  - `KernelEquateNEATPS()`
  - `MomentsFromFD()`
- `Print_CK()`

`Wrapper_CK()` is a “high-level” function that uses structures so that only a few arguments need to be passed to the functions `KernelEquateSEERG()` and `KernelEquate()` (as well `MomentsFromFD()`). These latter functions do most of the computational work. They have a large number of arguments that do not use structures, which should make it easier for users to modify the functions, if desired.

### 12.4.1 Wrapper\_CK()

`Wrapper_CK()` calls functions that performs kernel equating for the random groups design. The function prototype for `Wrapper_CK()` is:

```
void Wrapper_CK(char design, char method, char smoothing,
                double w1, int anchor, double rv1, double rv2,
                struct BSTATS *xv, struct BSTATS *yv,
                struct BLL_SMOOTH *bllxv,
                struct BLL_SMOOTH *bllyv, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' designating the random groups design;
- ▷ `method` = see equating methods list below;
- ▷ `smoothing` = 'K' for kernel 'smoothing';
- ▷ `w1` = synthetic population weight for the new form;
- ▷ `rv1` = reliability of common item set (new form);
- ▷ `rv2` = reliability of common item set (old form);
- ▷ `xv` = pointer to a `BSTATS` (new form);
- ▷ `yv` = pointer to a `BSTATS` (old form);
- ▷ `bllxv` = pointer to a `BLL_SMOOTH` structure (new form);
- ▷ `bllyv` = pointer to a `BLL_SMOOTH` structure (old form); and
- ▷ `rep` = replication number for bootstrap; should be set to 0 for actual equating.

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_CK()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

Two arguments of the wrapper function `Wrapper_CK()` need particular attention. First, `method` must be one of the following:

- 'E' = FE
- 'F' = MFE
- 'G' = FE and MFE
- 'C' = Chained
- 'H' = FE and Chained
- 'A' = FE, MFE, and Chained,

where FE is frequency estimation, and MFE is modified frequency estimation. Second, if MFE is requested ('F', 'G', or 'A'), then reliability estimates must be provided for `rv1` and `rv2`.

`Wrapper_CK()` populates the elements of the `PDATA` structure `inall`, dynamically allocates space for the `eraw[] []` and `mts[] []` matrices in the `ERAW_RESULTS` structure `r`, calls a function that does the actual equating and stores results in `r`, and calls the function `MomentsFromFD()` that gets the equated raw-score moments and stores them in `r`.

### 12.4.2 Print\_CK()

The function prototype for `Print_CK()` is:

```
void Print_CK(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the kernel equating results for the common-item nonequivalent groups design design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

### 12.4.3 Example

Table 12.5 is a `main()` function that illustrates kernel equating with the common-item nonequivalent groups design design based on an example discussed by Kolen and Brennan (2004, chap. 5, pp. 147–151). Here, however, `w1 = .5`, rather than `w1 = 1` as in the Kolen and Brennan (2004) example. In this example, the function `Wrapper_Smooth_BLL()` is called twice to perform bivariate log-linear smoothing for the two bivariate distributions, and then `Wrapper_CK()` is called to perform kernel equating.

The output from `Print_CK()` is provided in Table 12.6. The top of the output provides general information about the log-linear smoothing. The bottom part provides the equated raw scores for the kernel equating, as well as their moments based on using the new-form frequencies.

Figure 12.2 contains plots of the kernel equating for both frequency estimation and chained equipercentile methods. It also plots percentile-rank based equipercentile equatings for comparison.

TABLE 12.1. Main() Code to Illustrate Kernel Equating with the Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct ULL_SMOOTH ullx, ully;
    struct PDATA pdREK;
    struct ERAW_RESULTS rREK;
    struct ESS_RESULTS sREK;

    FILE *outf;

    outf = fopen("Chap 12 RG out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004): Chapter 2 example:
       Equipercntile equating (see pp. 52, 53, 57, 60) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y",&y);

    Wrapper_Smooth_ULL(&x, 6, 1, 1, 0, .000001, NULL, &ullx);
    Print_ULL(outf, "ACT Math X", &x, &ullx, 2, 2);

    Wrapper_Smooth_ULL(&y, 6, 1, 1, 0, .000001, NULL, &ully);
    Print_ULL(outf, "ACT Math Y", &y, &ully, 2, 2);

    /* Kernel equating for Random Groups Design */

    Wrapper_RK('R','E','K', &x, &y, &ullx, &ully, 0, &pdREK,&rREK);
    Print_RK(outf,"ACT Math---Kernel---Log Linear Smoothing",&pdREK, &rREK);

    Wrapper_ESS(&pdREK,&rREK,0,40,1,"yctmath.TXT",1,1,36,&sREK);
    Print_ESS(outf,"ACT Math---Equipercntile",&pdREK,&sREK);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 12.2. Output for Equated Raw Scores using Kernel Equating Under a RG Design

```
*****
ACT Math---Kernel Equating---Log Linear Smoothing

Kernel Equating with Random Groups Design and Polynomial Log-Linear Smoothing
Log-Linear Smoothing for new form X: number of degrees of smoothing = 6
Log-Linear Smoothing for old form Y: number of degrees of smoothing = 6
Input file for new form X: actmathfreq.dat
Input file for old form Y: actmathfreq.dat
-----
```

Raw Score (X)	y-equiv
0.00000	-0.70313
1.00000	0.05371
2.00000	0.91431
3.00000	1.80695
4.00000	2.70721
5.00000	3.60825
6.00000	4.51118
7.00000	5.41908
8.00000	6.33547
9.00000	7.26482
10.00000	8.21192
11.00000	9.18190
12.00000	10.17978
13.00000	11.21013
14.00000	12.27610
15.00000	13.37840
16.00000	14.51470
17.00000	15.67904
18.00000	16.86227
19.00000	18.05408
20.00000	19.24488
21.00000	20.42629
22.00000	21.59163
23.00000	22.73647
24.00000	23.85880
25.00000	24.95863
26.00000	26.03695
27.00000	27.09548
28.00000	28.13642
29.00000	29.16214
30.00000	30.17504
31.00000	31.17773
32.00000	32.17260
33.00000	33.16175
34.00000	34.14700
35.00000	35.13004
36.00000	36.11176
37.00000	37.09290
38.00000	38.07289
39.00000	39.05136
40.00000	40.02563
Mean	18.97894
S.D.	8.94050
Skew	0.35266
Kurt	2.14788

```
*****
```

TABLE 12.3. Main() Code to Illustrate Kernel Equating with the Single Group Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xy;
    struct BLL_SMOOTH bllxy;
    struct PDATA pdSEK;
    struct ERAW_RESULTS rSEK;
    struct ESS_RESULTS sSEK;
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm3[3][2] = {{1,1},{1,2},{2,1}};

    FILE *outf;

    outf = fopen("Chap 12 SG out","w");

    /* Single Groups. Dummy example using Chapter 4 BSTATS for old form
       with v serving as x in struct BSTATS xy; log-linear smoothing */

    convertFtoW("mondaty.dat",2,fieldsACT,"mondaty-temp");
    ReadRawGet_BSTATS("mondaty-temp",2,1,0,12,1,0,36,1,'X','Y',&xy);
    Print_BSTATS(outf,"Ch 4 dummy: v->x and y from ch 4 serve as x and y",&xy,0);

    /* log-linear smoothing of the bivariate distribution */
    Wrapper_Smooth_BLL(&xy, 0, 6, 6, 3, cpm3, 1, 1, 0, .000001, NULL, &bllxy);
    Print_BLL(outf,"Ch 4 dummy: v->x and y from ch 4 serve as x and y",
               &xy, &bllxy, 0, 2, 2, 1);

    /* Kernel equating for Single Group Design */

    Wrapper_SK('S','E','K', &xy, &bllxy, 0, &pdSEK, &rSEK);
    Print_SK(outf,"Kernel SG design: v->x and y from ch 4 serve as x and y",
             &pdSEK,&rSEK);

    Wrapper_ESS(&pdSEK,&rSEK,0,40,1,"yctmath.TXT",1,1,36,&sSEK);
    Print_ESS(outf,"ACT Math---Equipercentile",&pdSEK,&sSEK);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 12.4. Output for Equated Raw Scores using Kernel Equating Under a SG Design

```

*****

Kernel SG design: v->x and y from ch 4 serve as x and y

Input filename:  mondaty-temp

Kernel Equating with Single Group Design
and Polynomial Log-Linear Smoothing

Bivariate Log-Linear Smoothing using the Multinomial Model

Number of score categories for X  = 13
Number of score categories for Y  = 37
Total number of score categories = 481

Number of persons (i.e., total of frequencies) =    1638

Polynomial degree for X = 6
Polynomial degree for Y = 6
Number of cross-products = 3
Cross-Product moments:   (X1,Y1),  (X1,Y2),  (X2,Y1)

-----

Equated Raw Scores

Raw Score (X)      Method 0:
                   y-equiv

    0.00000         4.24899
    1.00000         6.19888
    2.00000         8.25270
    3.00000        10.56964
    4.00000        13.15651
    5.00000        15.93228
    6.00000        18.82373
    7.00000        21.82985
    8.00000        24.94289
    9.00000        27.97827
   10.00000        30.62910
   11.00000        32.81702
   12.00000        34.76323

    Mean           18.67180
    S.D.            6.88585
    Skew            0.20992
    Kurt            2.28762

*****

```

TABLE 12.5. Main() Code to Illustrate Kernel Equating with the CINEG Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xv,yv;
    struct BLL_SMOOTH bllxv, bllyv;
    struct PDATA pdCHK;
    struct ERAW_RESULTS rCHK;
    struct ESS_RESULTS sCHK;
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm1[1][2] = {{1,1}};

    FILE *outf;

    outf = fopen("Chap 12 CG out","w");

    /* CG Design: Kolen and Brennan (2004, chap. 5) example with w1=.5:
       Equipercntile equating with log-linear smoothing;
       Note: Kolen and Brennan (2004, p. 151) uses w1=1 */

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'X','V",&xv);

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'Y','V",&yv);

    Print_BSTATS(outf,"xv",&xv,1);
    Print_BSTATS(outf,"yv",&yv,1);

    /* Bivariate log-linear smoothing */

    Wrapper_Smooth_BLL(&xv, 1, 6, 6, 1, cpm1, 1, 1, 0, .000001, NULL, &bllxv);
    Print_BLL(outf, "Chapter 5: log-linear smoothing of x and v in pop 1",
               &xv, &bllxv, 0, 2, 2, 1);

    Wrapper_Smooth_BLL(&yv, 1, 6, 6, 1, cpm1, 1, 1, 0, .000001, NULL, &bllyv);
    Print_BLL(outf, "Chapter 5: log-linear smoothing of y and v in pop 2",
               &yv, &bllyv, 0, 2, 2, 1);

    /* Kernel equating for CINEG Design */

    Wrapper_CK('C','H','K', .5, 0, 0, 0, &xv,&yv, &bllxv,&bllyv,
               0,&pdCHK,&rCHK);

    Print_CK(outf,"SAT: w1 = .5, Kernel NEATPS with log-linear smoothing",&pdCHK,&rCHK);
    Print_SynDens(outf,"SAT: w1 = .5, log-linear smoothing",&pdCHK,&rCHK);

    Wrapper_ESS(&pdCHK,&rCHK,-11,54,1,"Conversion-reordered.DAT",2,100,850,&sCHK);
    Print_ESS(outf,"SAT: w1 = .5, Kernel with log-linear smoothing",&pdCHK,&sCHK);

    fclose(outf);
    return 0;
}
*****/

```



TABLE 12.6. Output for Equated Raw Scores using Kernel Equating Under a CINEG Design

```

*****
Ch. 5 Real Data: w1 = .5, Kernel CINEG with log-linear smoothing

Kernel Equating with CINEG Design (External Anchor; w1 = 0.50000)
and Polynomial Log-Linear Smoothing
Input file for XV and pop 1: mondatx-temp
Input file for YV and pop 2: mondaty-temp

Number of persons for X =      1655
Number of score categories for X = 37; Number of score categories for V = 13
Polynomial degree for X = 6; Polynomial degree for V = 6
Number of cross-products  XV = 1; Cross-Product moments:    ( X1,V1)

Number of persons for Y =      1638
Number of score categories for Y = 37; Number of score categories for V = 13
Polynomial degree for Y = 6; Polynomial degree for V = 6
Number of cross-products  YV = 1; Cross-Product moments:    ( Y1,V1)

Raw Score (X)           FE      ChainedE

    0.00000      0.21094      0.43555
    1.00000      1.30029      1.44629
    2.00000      2.34192      2.46826
    3.00000      3.38635      3.49866
    4.00000      4.43079      4.53783
    5.00000      5.47820      5.54997
    6.00000      6.53018      6.55721
    7.00000      7.58716      7.56163
    8.00000      8.64945      8.56035
    9.00000      9.71687      9.57232
   10.00000     10.78864     10.58346
   11.00000     11.86334     11.60320
   12.00000     12.93890     12.63416
   13.00000     14.01261     13.66399
   14.00000     15.08139     14.70294
   15.00000     16.14188     15.74325
   16.00000     17.19101     16.77676
   17.00000     18.22623     17.81066
   18.00000     19.24584     18.83570
   19.00000     20.24912     19.84768
   20.00000     21.23670     20.85438
   21.00000     22.21037     21.85034
   22.00000     23.17290     22.83542
   23.00000     24.12815     23.82019
   24.00000     25.08037     24.80294
   25.00000     26.03408     25.78561
   26.00000     26.99324     26.78034
   27.00000     27.96090     27.78341
   28.00000     28.93848     28.79739
   29.00000     29.92474     29.82796
   30.00000     30.91539     30.86134
   31.00000     31.90297     31.89665
   32.00000     32.87756     32.90704
   33.00000     33.82935     33.87006
   34.00000     34.74954     34.75165
   35.00000     35.62622     35.55884
   36.00000     36.45166     36.33936

      Mean      16.80661      16.53442
      S.D.       6.64932      6.62189
      Skew       0.48954      0.55772
      Kurt       2.60018      2.67700
*****

```

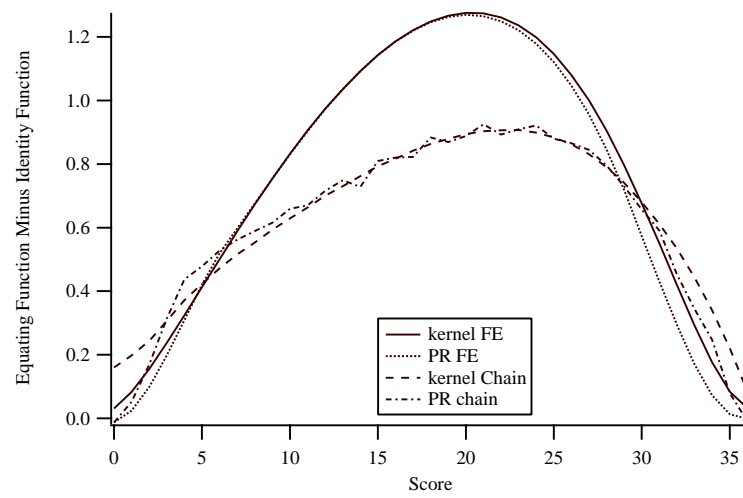


FIGURE 12.2. Kernel and percentile-rank based equating using the frequency estimation and chained equipercntile methods under a CINEG design.

# 13

## Standard Error of Kernel Equating

The previous chapter describes the kernel equating method (Holland & Thayer, 1989; von Davier, Holland & Thayer, 2004). In the five-step kernel equating framework, the last step is to compute the standard error of equating (SEE). SEE is defined as the standard deviation of the equated scores over many random replications of sampling and equating from the same population(s) (Kolen & Brennan, 2004, chap. 7). SEE is the random part of equating error; whereas bias is the systematic part of equating error. Holland, King, and Thayer (1989) first derived the SEE for the kernel method based on the  $\delta$ -method. von Davier et al. (2004) laid out a more systematic description of the derivation.<sup>1</sup>

### 13.1 Standard Error of Kernel Equating Based on the Delta Method

Holland, King, and Thayer (1989) and von Davier et al. (2004) derived a general expression for SEEs for different equating designs based on the  $\delta$ -method (see Section 7.1). In the kernel equating framework, the sampling variance of the estimated parameters of the log-linear model can be found using asymptotic theorems (Holland & Thayer, 1987, 2000). After the log-linear model estimates are obtained, there are two layers of functions in kernel equating, one related to the Design Function, the other related to continuization with a Gaussian kernel and

---

<sup>1</sup>The *Equating Recipes* code for estimating standard errors for kernel equating is somewhat newer and less tested than code for more traditional methods. We anticipate, therefore, that the code may be improved over time.

equipercentile equating. Thus the derivation of the SEEs of the kernel equating method “divides the problem in three” and ends up with an expression with three components:

$$SEE_Y(x) = \|J_{e_Y} J_{DF} C\|, \quad (13.1)$$

where  $\|\cdot\|$  is the Euclidian norm of a vector such that for any vector  $v$ ,  $\|v\| = \sqrt{\sum_j v_j^2}$ . The three components on the right-hand side of the above equation are related to three different steps in the kernel equating method, with the continuization and equating combined into one step. The  $C$  matrix is related to the first step of the five-step kernel equating framework.  $C$  is a matrix that characterizes the variance-covariance matrix of estimates of score distributions from the log-linear model,  $\hat{R}$  and  $\hat{S}$ , by the following relationship (von Davier et al., Equation 5.11):

$$\Sigma_{\hat{R}, \hat{S}} = C C^t, \quad (13.2)$$

where  $C^t$  means the transpose of  $C$ . The  $C$  matrix is a compound matrix composed of  $C_R$  and  $C_S$  as

$$C = \begin{pmatrix} C_R & 0 \\ 0 & C_S \end{pmatrix}. \quad (13.3)$$

The procedure for computing the  $C_R$  and  $C_S$  matrices is described in von Davier et al. (2004, p. 52) and will not be repeated here.

$J_{DF}$  is a matrix which is the component that relates to the Design Function, which is associated with the second step of the five-step framework.  $J_{DF}$  can be expressed as (von Davier et al., 2004, Equation 5.10):

$$J_{DF} = \begin{pmatrix} \frac{\partial r}{\partial \hat{R}} & \frac{\partial r}{\partial \hat{S}} \\ \frac{\partial s}{\partial \hat{R}} & \frac{\partial s}{\partial \hat{S}} \end{pmatrix}, \quad (13.4)$$

where  $r$  and  $s$  are the marginal score probabilities for the new and old test forms for the target population. They are the results of applying the Design Function to the log-linear smoothed univariate or bivariate score distributions. The Design Function is the identity function for the random groups (RG) design, but for the other equating designs, the expression for the Design Function is complicated. The various expressions are described in detail in von Davier et al. (2004, chap. 5). Note that

$$J_{DF} C = \begin{pmatrix} \frac{\partial r}{\partial \hat{R}} C_R & \frac{\partial r}{\partial \hat{S}} C_S \\ \frac{\partial s}{\partial \hat{R}} C_R & \frac{\partial s}{\partial \hat{S}} C_S \end{pmatrix}, \quad (13.5)$$

or

$$J_{DF} C = \begin{pmatrix} U_R & U_S \\ V_R & V_S \end{pmatrix}. \quad (13.6)$$

$J_{e_Y}$  in Equation 13.1 is a vector that is related to the third (continuization) and fourth (equating) steps in the five-step framework. It can be expressed as (von Davier et al., 2004, Equation 5.9):

$$J_{e_Y} = \left( \frac{\partial e_Y}{\partial r}, \frac{\partial e_Y}{\partial s} \right), \quad (13.7)$$

where  $e_Y$  is the equating function. With kernel equating, the discrete score probabilities  $r$  and  $s$  are continuized using the Gaussian kernel to produce two continuous cdf's,  $F$  and  $G$ . Given  $F$  and  $G$ ,  $e_Y$  is defined as

$$e_Y(x; r, s) = G^{-1}(F(x; r); s). \quad (13.8)$$

The derivatives in Equation 13.7 can be expressed as (von Davier et al., Equation 5.16, 5.17):

$$\frac{\partial e_Y}{\partial r_j} = \frac{1}{G'} \frac{\partial F(x; r)}{\partial r_j}, \quad (13.9)$$

and

$$\frac{\partial e_Y}{\partial s_k} = -\frac{1}{G'} \frac{\partial G(e_Y(x); s)}{\partial s_k}, \quad (13.10)$$

where

$$G' = \frac{\partial G(e_Y(x); s)}{\partial y}. \quad (13.11)$$

The expressions for  $\frac{\partial F(x; r)}{\partial r_j}$  and  $\frac{\partial G(e_Y(x); s)}{\partial y}$  can be found in von Davier et al. (2004, Equations 5.21 and 5.22). So we have

$$J_{e_Y} = \frac{1}{G'} \left( \frac{\partial F}{\partial r}, -\frac{\partial G}{\partial s} \right), \quad (13.12)$$

After putting together all the elements, the general expression for the SEE for all designs is

$$SE_{E_Y}(x) = \frac{1}{G'} \left[ \left\| \frac{\partial F}{\partial r} U_R - \frac{\partial G}{\partial s} V_R \right\|^2 + \left\| \frac{\partial F}{\partial r} U_S - \frac{\partial G}{\partial s} V_S \right\|^2 \right]^{1/2}. \quad (13.13)$$

The expression in Equation 13.13 applies to all kernel equating methods under all designs except for chained kernel equating under the CINEG design. Chained kernel equating is different from the other methods in that it involves two single group (SG) kernel equatings and never defines the target population. Readers are referred to von Davier et al. (2004, section 5.4) for a detailed description of SEEs for chained kernel equating.

## 13.2 Functions for Random Groups Design

The C functions for the kernel equating SEEs are in `Kernel_Equate.c`, with the function prototypes in `Kernel_Equate.h`. These functions do not include the log-linear smoothing step, but do include the equating step. That means that either `Wrapper_Smooth_ULL()` or `Wrapper_Smooth_BLL()` needs to be called before the functions for SEEs are called.

`KernelEquateSEERG()` computes the kernel SEEs for the random groups design. This function calls `ComputeCmatrixGen()`, `Optimalhx()`, `KernelEquate()`, `PartialFPpartialr()`, `KernelContinuPdf()`, and `FrCrSqNorm()`. These functions in turn call other functions. The calling sequence is:

- `KernelEquateSEERG()`
  - `ComputeCmatrixGen()`
    - \* `er_qrdcmp()`
  - `Optimalhx()`
    - \* `Pen()`
  - `KernelEquate()`

```

    * KernelContinuCdf()
    * PeKernelInverseCdf()
- PartialFPartialr()
- KernelContinuPdf()
- FrCrSqNorm()

```

### 13.2.1 KernelEquateSEERG()

The function prototype for `KernelEquateSEERG()` is:

```

void KernelEquateSEERG(int nDistCatx, int degreeex,
                       long npx, double *scoresx, double *fdx,
                       int nDistCaty, int degreeey, long npy,
                       double *scoresy, double *fdy,
                       double *Equatedx, double *SEE)

```

The input variables are:

- ▷ `ncatx` = number of score categories (new form);
- ▷ `degreeex` = log-linear smoothing degree (new form);
- ▷ `npx` = sample size (new form);
- ▷ `scoresx` = discrete scores (new form);
- ▷ `fdx` = relative frequencies (new form);
- ▷ `ncaty` = number of score categories (old form);
- ▷ `degreeey` = log-linear smoothing degree (old form);
- ▷ `npy` = sample size (old form);
- ▷ `scoresy` = discrete scores (old form); and
- ▷ `fdy` = relative frequencies (old form).

The output variables are:

- ▷ `Equatex` = vector of equated scores; and
- ▷ `SEE` = vector of SEEs.

Note that `KernelEquateSEERG()` outputs the equating function itself. However, if only the equating function is needed, using `KernelEquate()` is much quicker. Computational time is especially relevant when procedures such as the bootstrapping are used to compute the SEE. The bootstrap procedure typically calls the equating function hundreds of times.

### 13.2.2 Example

Table 13.1 is a `main()` function that illustrates computing kernel equating SEEs for the random groups design based on an example discussed by Kolen and Brennan (2004, chap. 2, p. 48). In this example, `Wrapper_Smooth_ULL` is called to perform univariate log-linear smoothing for both the  $X$  and  $Y$  distributions, and then `KernelEquateSEERG()` is called to compute SEEs.

TABLE 13.1. Main() Code to Illustrate Kernel Equating with the Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct ULL_SMOOTH ullx, ully;
    double scoresx[41], scoresy[41], eq_krg[41], se_krg[41];
    int i;

    FILE *outf;

    outf = fopen("Chap 13 RG out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004): Chapter 2 example:
       Equipercntile equating (see pp. 52, 53, 57, 60) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X",&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y",&y);

    Wrapper_Smooth_ULL(&x, 6, 1, 1, 0, .000001, NULL, &ullx);
    Print_ULL(outf, "ACT Math X", &x, &ullx, 2, 2);

    Wrapper_Smooth_ULL(&y, 6, 1, 1, 0, .000001, NULL, &ully);
    Print_ULL(outf, "ACT Math Y", &y, &ully, 2, 2);

    /* Kernel equating SEE for Random Groups Design */

    for (i=0;i<x.ns;i++) scoresx[i] = x.min + i * x.inc;
    for (i=0;i<y.ns;i++) scoresy[i] = y.min + i * y.inc;

    KernelEquateSEERG(x.ns, ullx.c, x.n, scoresx, ullx.density,
                      y.ns, ully.c, y.n, scoresy, ully.density,
                      eq_krg, se_krg);
    Print_vector(outf, "ACT Math---kernel delta SE's",
                 se_krg, x.ns, "    Raw Score", "          SE");

    fclose(outf);
    return 0;
}
*****/

```

TABLE 13.2. Output for Kernel Equating SEE Under a RG Design

/\*\*\*\*\*/

ACT Math---kernel delta SE's

Raw Score	SE
0.00000	0.27994
1.00000	0.34578
2.00000	0.33077
3.00000	0.28464
4.00000	0.23932
5.00000	0.20225
6.00000	0.17474
7.00000	0.15746
8.00000	0.15004
9.00000	0.15046
10.00000	0.15604
11.00000	0.16478
12.00000	0.17584
13.00000	0.18928
14.00000	0.20538
15.00000	0.22384
16.00000	0.24323
17.00000	0.26128
18.00000	0.27637
19.00000	0.28861
20.00000	0.29875
21.00000	0.30685
22.00000	0.31283
23.00000	0.31707
24.00000	0.31987
25.00000	0.32112
26.00000	0.32064
27.00000	0.31857
28.00000	0.31548
29.00000	0.31200
30.00000	0.30842
31.00000	0.30435
32.00000	0.29893
33.00000	0.29139
34.00000	0.28197
35.00000	0.27267
36.00000	0.26667
37.00000	0.26528
38.00000	0.26151
39.00000	0.23151
40.00000	0.14497

/\*\*\*\*\*/



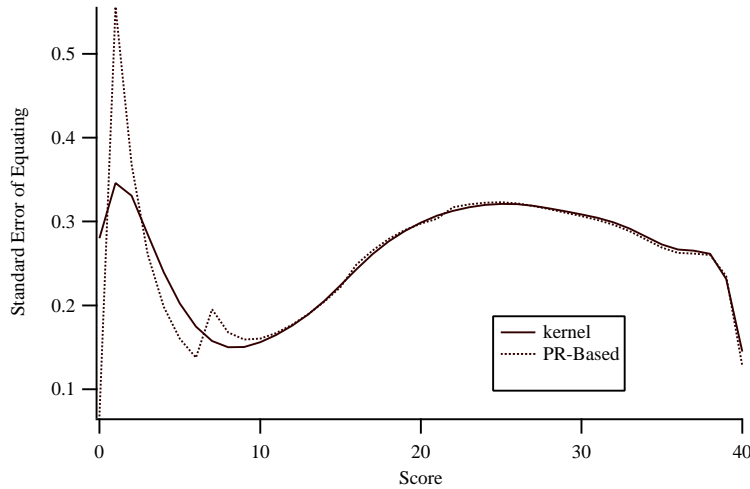


FIGURE 13.1. SEEs of Kernel and Percentile-Rank Based Equipercentile Equating under a RG Design

The SEEs are presented in Table 13.2 and plotted in Figure 13.1. For comparison purposes, the SEEs for the conventional percentile-rank based equipercentile equating with log-linear smoothing are also plotted in Figure 13.1. It can be seen that the SEEs for these two equating methods are very similar, although there are some differences at the lower end of score scale.

### 13.3 Functions for the Single Group Design

The function that computes the kernel SEE for the single group design is `KernelEquateSEESG()`, which calls functions `ComputeCmatrixGen()`, `Optimalhx()`, `KernelEquate()`, and some other functions. These functions in turn calls other functions. The calling sequence is

- `KernelEquateSEESG()`
  - `ComputeCmatrixGen()`
    - \* `er_qrdcmp()`
  - `Optimalhx()`
    - \* `Pen()`
  - `KernelEquate()`
    - \* `KernelContinuCdf()`
    - \* `PeKernelInverseCdf()`
  - `MatrixMultiMatrix()`
  - `vPMN()`
  - `vPT()`
  - `MatrixMultiVector()`
  - `VectorMultiMatrix()`

### 13.3.1 KernelEquateSEESG()

The function prototype for KernelEquateSEESG() is:

```
void KernelEquateSEESG((BLL_SMOOTH *bivar,
                        double *Equatedx, double *SEE)
```

The input variable is:

- ▷ **bivar** = pointer to a BLL\_SMOOTH structure.

The output variables are:

- ▷ **Equatedx** = vector equated scores; and
- ▷ **SEE** = vector for SEE.

### 13.3.2 Example

Table 13.3 is a `main()` function that illustrates computing kernel equating SEE for the single group design using an example discussed by Kolen and Brennan (2004, chap. 4, old form). In this example, the `Wrapper_Smooth_BLL` is called to perform bivariate log-linear smoothing for the joint distribution of  $X$  and  $Y$ , and then `KernelEquateSEESG()` is called to compute SEE.

Due to limited space, the output is not presented for this example.

## 13.4 Functions for the Common-Item Nonequivalent Groups Design

With the CINEG design, we consider two kernel equating methods: the frequency estimation method and the chained equipercentile method. The functions that compute the SEEs for these methods are described separately below. Then, a sample `main()` function for both methods is provided.

The function that computes the kernel SEEs for the common-item nonequivalent groups design using the frequency estimation method is named `KernelEquateSEENEATPS()`, which calls functions `ComputeCmatrixGen()`, `Optimalhx()`, `KernelEquate()`, and some other functions. These functions in turn call other functions. The calling sequence is:

- `KernelEquateSEENEATPS()`
  - `ComputeCmatrixGen()`
    - \* `er_qrdcmp()`
  - `Optimalhx()`
    - \* `Pen()`
  - `KernelEquate()`
    - \* `KernelContinuCdf()`
    - \* `PeKernelInverseCdf()`
  - `MatrixMultiMatrix()`
  - `vPMN()`
  - `vPT()`
  - `MatrixMultiVector()`
  - `VectorMultiMatrix()`

### 13.4.1 KernelEquateSENEATPS()

The function prototype for KernelEquateSENEATPS() is:

```
void KernelEquateSENEATPS(BLL_SMOOTH *bivar1,
                          BLL_SMOOTH *bivar2, double wts,
                          double *Equatedx, double *SEE)
```

The input variables are:

- ▷ **bivar1** = pointer to a BLL\_SMOOTH structure (new form);
- ▷ **bivar2** = pointer to a BLL\_SMOOTH structure (old form);
- ▷ **wts** = synthetic population weight of new form;

The output variables are:

- ▷ **Equatex** = vector of equated scores; and
- ▷ **SEE** = vector of SEEs.

The function that computes the kernel SEEs for the common-item nonequivalent groups design using the chained equipercntile method is named **KernelEquateSENEATChn()**, which calls the functions **KernelEquateSEESG()**, **ComputeCmatrixGen()**, **Optimalhx()**, **KernelEquate()**, and some other functions. These functions in turn call other functions. The calling sequence is:

- **KernelEquateSENEATChn()**
  - **KernelEquateSEESG()**
  - **ComputeCmatrixGen()**
    - \* **er\_qrdcmp()**
  - **Optimalhx()**
    - \* **Pen()**
  - **KernelEquate()**
    - \* **KernelContinuCdf()**
    - \* **PeKernelInverseCdf()**
  - **MatrixMultiMatrix()**
  - **vPMN()**
  - **vPT()**
  - **MatrixMultiVector()**
  - **VectorMultiMatrix()**

### 13.4.2 KernelEquateSENEATChn()

The function prototype for KernelEquateSENEATChn() is:

```
void KernelEquateSENEATChn(BLL_SMOOTH *bivar1, BLL_SMOOTH *bivar2,
                           double *Equatedx, double *SEE)
```

The input variables are:

- ▷ **bivar1** = pointer to a BLL\_SMOOTH structure (new form);
- ▷ **bivar2** = pointer to a BLL\_SMOOTH structure (old form);

The output variables are:

- ▷ **Equatex** = vector of equated scores; and
- ▷ **SEE** = vector of SEEs.

### 13.4.3 Example

Table 13.4 is a `main()` function that illustrates how to obtain kernel equating standard errors with the common-item nonequivalent groups design based on an example discussed by Kolen and Brennan (2004, chap. 5, pp. 147–151). Here, however, `w1 = .5`, rather than `w1 = 1` as in the Kolen and Brennan (2004) example. The `main()` function calls `Wrapper_Smooth_BLL()` twice to perform bivariate log-linear smoothing and then calls the functions `KernelEquateSEENEATPS()` and `KernelEquateSEENEATChn()` to compute the SEEs for both the frequency estimation and chained equipercentile methods, respectively.

The functions `KernelEquateSEENEATPS()` and `KernelEquateSEENEATChn()` produce output that is provided in Table 13.5. The table is reformatted to include bootstrap SEEs (based on bivariate log-linear smoothing) for comparison.

TABLE 13.3. Main() Code to Illustrate Kernel Equating SEE with the Single Group Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xy;
    struct BLL_SMOOTH bllxy;
    double eq_ksg[13], se_ksg[13];
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm3[3][2] = {{1,1},{1,2},{2,1}};

    FILE *outf;

    outf = fopen("Chap 13 SG out","w");

    /* Single Groups. Dummy example using Chapter 4 bstats for old form
       with v serving as x in struct BSTATS xy; log-linear smoothing */

    convertFtoW("mondaty.dat",2,fieldsACT,"mondaty-temp");
    ReadRawGet_BSTATS("mondaty-temp",2,1,0,12,1,0,36,1,'X','Y",&xy);
    Print_BSTATS(outf,"Ch 4 dummy: v->x and y from ch 4 serve as x and y",&xy,0);

    /* log-linear smoothing of the bivariate distribution */

    Wrapper_Smooth_BLL(&xy, 0, 6, 6, 3, cpm3, 1, 1, 0, .000001, NULL, &bllxy);

    /* Kernel equating SEE for Single Group Design */

    KernelEquateSEESG(&bllxy, eq_ksg, se_ksg);
    Print_vector(outf, "Ch 4 data old form--SG--kernel delta SE's",
                 se_ksg, xy.nsl, "    Raw Score", "          SE");

    fclose(outf);
    return 0;
}
*****/

```

TABLE 13.4. Main() Code to Illustrate Kernel Equating SEE with the CINEG Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xv,yv;
    struct BLL_SMOOTH bllxv, bllyv;
    double eq_kcfe[37], se_kcfe[37], eq_kcchn[37], se_kcchn[37];
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm1[1][2] = {{1,1}};

    FILE *outf;

    outf = fopen("Chap 13 CG out","w");

    /* CG Design: Kolen and Brennan (2004, chap. 5) example with w1=.5:
       Equipercntile equating with log-linear smoothing;
       Note: Kolen and Brennan (2004, p. 151) uses w1=1 */

    convertFtoW("mondtx.dat",2,fieldsACT,"mondtx-temp");
    ReadRawGet_BSTATS("mondtx-temp",1,2,0,36,1,0,12,1,'X','V",&xv);

    convertFtoW("mondty.dat",2,fieldsACT,"mondty-temp");
    ReadRawGet_BSTATS("mondty-temp",1,2,0,36,1,0,12,1,'Y','V",&yv);

    Print_BSTATS(outf,"xv",&xv,1);
    Print_BSTATS(outf,"yv",&yv,1);

    Wrapper_Smooth_BLL(&xv, 1, 6, 6, 1, cpm1, 1, 1, 0, .000001, NULL, &bllxv);
    Print_BLL(outf, "Chapter 5: log-linear smoothing of x and v in pop 1",
               &xv, &bllxv, 0, 2, 2, 1);

    Wrapper_Smooth_BLL(&yv, 1, 6, 6, 1, cpm1, 1, 1, 0, .000001, NULL, &bllyv);
    Print_BLL(outf, "Chapter 5: log-linear smoothing of y and v in pop 2",
               &yv, &bllyv, 0, 2, 2, 1);

    /* Kernel equating SEE with FE for CINEG Design */

    KernelEquateSENEATPS(&bllxv, &bllyv, .5, eq_kcfe, se_kcfe);
    Print_vector(outf, "Ch 5 Data--FE--kernel delta SE's",
                 se_kcfe, xv.ns1, " Raw Score", " SE");

    /* Kernel equating SEE with Chain for CINEG Design */

    KernelEquateSENEATChn(&bllxv, &bllyv, eq_kcchn, se_kcchn);
    Print_vector(outf, "Ch 5 Data--Chain--kernel delta SE's",
                 se_kcchn, xv.ns1, " Raw Score", " SE");

    fclose(outf);
    return 0;
}
*****/

```

TABLE 13.5. Analytical and Bootstrap SEE for Frequency Estimation and Chained Equipercntile Kernel Equating under a CINEG Design

Score	Analy. SEE FE	Bootst. SEE FE	Analy. SEE Chain	Bootst. SEE Chain
0	2.03401	0.52739	0.66495	0.52779
1	0.79349	0.91488	0.62664	0.92429
2	0.54616	0.96411	0.53975	0.98353
3	0.44093	0.59864	0.45488	0.64571
4	0.34020	0.41478	0.36738	0.49582
5	0.27546	0.32947	0.28537	0.34686
6	0.22654	0.27018	0.23527	0.31322
7	0.19552	0.22717	0.20874	0.25615
8	0.18753	0.20411	0.19680	0.23210
9	0.17979	0.19750	0.18780	0.23471
10	0.17282	0.19971	0.17843	0.21324
11	0.16868	0.20251	0.17433	0.22919
12	0.16706	0.20179	0.17431	0.21936
13	0.17358	0.20348	0.17907	0.21555
14	0.17928	0.20864	0.18628	0.24200
15	0.18354	0.20649	0.19082	0.22472
16	0.18696	0.21379	0.19466	0.23644
17	0.18998	0.22462	0.19893	0.27057
18	0.19617	0.23002	0.20389	0.25591
19	0.20365	0.24083	0.21263	0.26854
20	0.21094	0.25095	0.22323	0.30404
21	0.21853	0.25954	0.23154	0.29139
22	0.22297	0.25873	0.23834	0.29039
23	0.22749	0.26314	0.24300	0.31365
24	0.23086	0.26632	0.24553	0.29559
25	0.23619	0.27628	0.25213	0.31297
26	0.24655	0.29367	0.26385	0.34739
27	0.25617	0.30060	0.27740	0.34269
28	0.26848	0.31430	0.29323	0.37411
29	0.27724	0.33391	0.30348	0.39014
30	0.28464	0.35139	0.31347	0.41581
31	0.30898	0.37842	0.33640	0.44245
32	0.33522	0.41372	0.37740	0.49580
33	0.37652	0.45235	0.41576	0.51538
34	0.42326	0.51285	0.41383	0.52090
35	0.41425	0.47843	0.38756	0.59087
36	0.32473	0.28967	0.31868	0.68725

*Note.* The analytic standard errors for frequency estimation appear to be rather sensitive to the algorithm used for QR decomposition. For example, the *Numerical Recipes* function `genqrncmp()` gives results that are occasionally somewhat different (on the order of  $10^{-3}$ ) from `er_qrdcmp()` which is used in *Equating Recipes*.





# 14

## Continuized Log-Linear Equating

The kernel equating method described by von Davier, Holland and Thayer (2004) is a test equating framework that adds a continuization step to the conventional non-IRT equating procedures with log-linear presmoothing. This framework applies to practically all equating designs and methods. The advantage of this framework is that it modularizes the equating steps (see Chapter 12 for a description of the framework). The continuization step von Davier et. al. (2004) proposed involves an adjusted Gaussian kernel procedure which produces two continuous distributions for  $X$  and  $Y$ . The continuous distributions preserve the first two moments of their respective discrete distributions. Wang (2007, 2008) proposed an alternative continuization method called the continuized log-linear (CLL) method which directly uses the log-linear function from the smoothing step and makes it a continuous probability density function (pdf). An advantages of the CLL method is that it preserves all the moments of the discrete distribution. An important difference between the CLL method and the kernel method is that for the kernel method, the step that applies the Design Function (see von Davier et al., 2004, chap. 2 for a detailed description of the Design Function) precedes the continuization step, whereas in the CLL method the Design Function step comes after the continuization step. Below is a description of the CLL method for various equating designs.<sup>1</sup>

---

<sup>1</sup>The *Equating Recipes* code for continuized log-linear equating is somewhat newer and less tested than code for more traditional methods. We anticipate, therefore, that the code may be improved over time.

### 14.1 Continuized Log-linear Method for the Random Groups Design

For the random groups (RG) design, the Design Function is the identity function, which effectively means that this step can be omitted. The continuous probability density function (pdf) is taken directly from the log-linear smoothing and can be expressed as

$$f(x) = \frac{1}{D} \exp(\mathbf{b}^t \boldsymbol{\beta}), \quad (14.1)$$

where  $t$  means transpose,  $\mathbf{b}^t = (1, x, x^2, \dots, x^M)$  is a vector of polynomial terms in  $x$ ,  $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2, \dots, \beta_M)^t$  is a vector of parameters, and  $M$  is the order of the polynomial.  $D$  is a normalizing constant that ensures that  $f(x)$  is a pdf:

$$D = \int_l^u \exp(\mathbf{b}^t \boldsymbol{\beta}) dx,$$

where  $l$  and  $u$  are the lower and upper limits of integration. For the CLL procedure, they are set at  $-0.5$  and  $J + 0.5$ , respectively, where  $J + 1$  is the number of score categories.

The moments of the CLL distribution are approximately equal to those of the smoothed discrete distribution, as can be seen from the following relationship between the non-central moments of the CLL distribution and the smoothed discrete distribution:

$$\frac{\int_l^u x^i \exp(\mathbf{b}^t \boldsymbol{\beta}) dx}{\int_l^u \exp(\mathbf{b}^t \boldsymbol{\beta}) dx} \approx \frac{1}{N} \sum_{x=0}^J x^i \exp(\mathbf{b}^t \boldsymbol{\beta}), \quad (14.2)$$

where  $N$  is the sample size. This approximation holds because the right side of the equation is an expression for numerical integration of the left side with equally spaced quadrature points. The numerator and denominator of the left side can be expressed separately as:

$$\int_l^u x^i \exp(\mathbf{b}^t \boldsymbol{\beta}) dx \approx \sum_{x=0}^J x^i \exp(\mathbf{b}^t \boldsymbol{\beta}), \quad (14.3)$$

and

$$D = \int_l^u \exp(\mathbf{b}^t \boldsymbol{\beta}) dx \approx \sum_{x=0}^J \exp(\beta_0 + \beta_1 x + \dots + \beta_M x^M) = N. \quad (14.4)$$

This means that the normalizing constant is approximately equal to the sample size, which is known prior to equating. This result significantly simplifies computations.

Equations 14.2–14.4 are conceptually related to the trapezoidal rule (see Thisted, 1988, p. 264) with a subinterval length of 1. Here, however, the range of the continuous distribution is  $-0.5$  to  $J + 0.5$ , and the quadrature function is evaluated at the mid points of the subintervals rather than at

the end points (as is usual for the trapezoidal rule). This range is consistent with the range of the percentile rank method in conventional equipercentile equating (Kolen & Brennan, 2004). Because of the smoothness of the log-linear function, the approximation can be quite close as the number of quadrature points (i.e.,  $J + 1$ ) gets large.

CLL continuization seems to have several advantages over kernel continuization. First, CLL continuization is simpler and more direct. Second, it is smoother; it does not have the small bumpiness that is characteristic of kernel continuization. Third, it preserves all the moments of the discrete distribution to the precision of equally-spaced numerical integration with  $J + 1$  quadrature points.

## 14.2 Single Group, Counter Balanced Design

For the SG design, both Form X and Form Y are administered to the same group of examinees. For the counter-balanced (CB) version of this design, the whole group takes both Form X and Form Y; however, approximately half of the group take Form X first and then Form Y, and the other half takes Form Y first and then Form X. We will label the first half group as Group 1 and the second half as Group 2. The SG design can be viewed as a special case of CB design when there is only one Group.

We treat the log-linear functions from Step 1 in the von Davier et al. (2004) framework as if they were continuous functions and normalize them to be pdf's. For Group 1, the pdf can be expressed:

$$f_1(x, y) = \frac{1}{D_1} \exp(\mathbf{b}^t \boldsymbol{\beta}), \quad (14.5)$$

where

$$\mathbf{b}^t = (1, x, x^2, \dots, x^{M_X}, y, y^2, \dots, y^{M_Y}, xy, x^2y, xy^2, \dots, x^{C_X}y^{C_Y})$$

is a vector of polynomial terms in  $x$  and  $y$ ,

$$\boldsymbol{\beta} = (\beta_{00}, \beta_{01}, \beta_{02}, \dots, \beta_{0M_X}, \beta_{10}, \beta_{20}, \dots, \beta_{M_Y0}, \beta_{11}, \beta_{12}, \beta_{21}, \dots, \beta_{C_X C_Y})^t$$

is a vector of parameters,  $M_X$  and  $M_Y$  are the orders of marginal polynomial terms for  $X$  and  $Y$ , respectively,  $C_X$  and  $C_Y$  are the orders of the cross-product terms for  $X$  and  $Y$ , respectively, and  $D_1$  is a normalizing constant which ensures that  $f_1(x, y)$  is a pdf:

$$D_1 = \int_{l_Y}^{u_Y} \int_{l_X}^{u_X} \exp(\mathbf{b}^t \boldsymbol{\beta}) dx dy.$$

As for the RG design, it can be shown that the normalizing constant approximates the sample size.

The joint pdf of Group 2,  $f_2(x, y)$ , can be found in a similar fashion. Let  $w_X$  and  $w_Y$  be the weights for Group 1 relative to  $f(x)$  and  $f(y)$ , respectively. Then, the combined marginal distributions for  $X$  and  $Y$  are:

$$f(x) = w_X \int_{l_Y}^{u_Y} f_1(x, y) dy + (1 - w_X) \int_{l_Y}^{u_Y} f_2(x, y) dy, \quad (14.6)$$

and

$$f(y) = w_Y \int_{l_X}^{u_X} f_1(x, y) dx + (1 - w_Y) \int_{l_X}^{u_X} f_2(x, y) dx. \quad (14.7)$$

Numerical integration methods are used to carry out the necessary integrations. The remaining steps are the same as for the RG design.

### 14.3 Common-Item Nonequivalent Groups Design

For the common-item nonequivalent groups (CINEG) design, Group 1 from Population 1 takes Form X plus the anchor set V, Group 2 from Population 2 takes Form Y plus the anchor set V. The continuous bivariate pdf's,  $f_1(x, v)$  for  $X$  and  $V$ , and  $f_2(y, v)$  for  $Y$  and  $V$ , can be obtained in a manner similar to that in the previous section for the CB design. We consider two equating methods under the CINEG design: the frequency estimation method (also called post-stratification) and the chained equipercenile equating method. The frequency estimation method makes the following assumptions:

$$f_2(x|v) = f_1(x|v) = f_1(x, v)/f_1(v), \quad (14.8)$$

and

$$f_1(y|v) = f_2(y|v) = f_2(y, v)/f_2(v). \quad (14.9)$$

The marginal distributions can be found using the following expressions:

$$f_1(x) = \int_{l_V}^{u_V} f_1(x, v) dv, \quad (14.10)$$

$$f_2(y) = \int_{l_V}^{u_V} f_2(y, v) dv, \quad (14.11)$$

$$f_1(v) = \int_{l_X}^{u_X} f_1(x, v) dx, \quad (14.12)$$

and

$$f_2(v) = \int_{l_Y}^{u_Y} f_2(y, v) dy. \quad (14.13)$$

Let  $w_1$  be the weight for Population 1 in the target population,  $T$ . Then, given the marginal distributions in Equations 14.10–14.13, the marginal distributions of  $X$  and  $Y$  for the target population are:

$$f_T(x) = w_1 f_1(x) + (1 - w_1) \int_{l_V}^{u_V} f_1(x|v) f_2(v) dv, \quad (14.14)$$

and

$$f_T(y) = w_1 \int_{l_V}^{u_V} f_2(y|v) f_1(v) dv + (1 - w_1) f_2(y). \quad (14.15)$$

The rest of the equating procedure is the same as in the RG design.<sup>2</sup>

The chained equipercentile equating method first equates  $X$  to  $V$  using  $f_1(x)$  and  $f_1(v)$ , and then equates these equivalents to  $Y$  using  $f_2(v)$  and  $f_2(y)$ . These marginal distributions are given by Equations 14.10–14.13.

## 14.4 Standard Errors for CLL Equating under the Random Groups Design

von Davier et al. (2004) derived the following general expression for the asymptotic standard error of equating:

$$SEE_Y(x) = \| J_{eY} J_{DF} C \| . \quad (14.16)$$

This expression is composed of three parts, each relating to a different stage of the equating process.  $J_{eY}$  is related to continuization (step 3) and equating (step 4),  $J_{DF}$  is related to estimation of score probabilities (step 2), and  $C$  is related to pre-smoothing (step 1). Because the CLL method uses the log-linear function directly in the continuization step, the cumulative distribution functions (cdf's) for Form X and Form Y depend on the estimated parameter vectors,  $\hat{\beta}_X$  and  $\hat{\beta}_Y$ , of the log-linear models, rather than on the estimated score probabilities  $\hat{r}$  and  $\hat{s}$  in von Davier et al. (2004). Let  $F$  denote the cdf of Form X and  $G$  denote the cdf of Form Y. The equating function from  $X$  to  $Y$  can be expressed as

$$e_Y(x) = e_Y(x; \beta_X, \beta_Y) = G^{-1}(F(x; \beta_X); \beta_Y), \quad (14.17)$$

where

$$F(x; \beta_X) = \frac{\int_l^x \exp(\mathbf{b}^t \boldsymbol{\beta}) dx}{\int_l^u \exp(\mathbf{b}^t \boldsymbol{\beta}) dx}, \quad (14.18)$$

---

<sup>2</sup>Processing time is quite slow for frequency estimation with the continuized log-linear method and the common-item nonequivalent groups design when test lengths are relatively long, because the numerical procedure used for double integration is computationally intensive.

and

$$G(y; \beta_Y) = \frac{\int_l^y \exp(\mathbf{b}^t \boldsymbol{\beta}) dy}{\int_l^u \exp(\mathbf{b}^t \boldsymbol{\beta}) dy}. \quad (14.19)$$

Using the  $\delta$ -method and following a similar approach as in Holland, King and Thayer (1989), the square of SEE can be expressed as

$$\sigma_Y^2(x) = (\partial e_Y)^t \Sigma (\partial e_Y) \quad (14.20)$$

where

$$\Sigma = \begin{bmatrix} \Sigma_{\hat{\beta}_X} & \Sigma_{\hat{\beta}_X \hat{\beta}_Y} \\ \Sigma_{\hat{\beta}_X \hat{\beta}_Y} & \Sigma_{\hat{\beta}_Y} \end{bmatrix}, \quad (14.21)$$

and

$$(\partial e_Y) = \begin{bmatrix} \frac{\partial e_Y}{\partial \beta_X} \\ \frac{\partial e_Y}{\partial \beta_Y} \end{bmatrix}. \quad (14.22)$$

The elements of  $\Sigma$  can be obtained using the following equation:

$$\Sigma_{\hat{\beta}_X} = (B_X^t \Sigma_{mX} B_X)^{-1}, \quad (14.23)$$

where  $B_X$  is the design matrix for Form X using the log-linear model (see Holland & Thayer, 1987); and

$$\Sigma_{mX} = N(D_{p_X} - p_X p_X^t), \quad (14.24)$$

where  $p_X$  is the vector of probabilities in the multinomial categories for Form X and  $D_{p_X}$  is a diagonal matrix made from  $p_X$ .  $\Sigma_{\hat{\beta}_Y}$  can be obtained in a similar fashion. We also have

$$\Sigma_{\hat{\beta}_X \hat{\beta}_Y} = \mathbf{0}. \quad (14.25)$$

The elements of  $(\partial e_Y)$  can be obtained from the following equations:

$$\frac{\partial e_Y}{\partial \beta_{Xi}} = \frac{1}{\frac{\partial G(y; \beta_Y)}{\partial y}} \bigg|_{y=e_Y(x)} \frac{\partial F(x; \beta_X)}{\partial \beta_{Xi}}, \quad (14.26)$$

and

$$\frac{\partial e_Y}{\partial \beta_{Yi}} = - \frac{1}{\frac{\partial G(y; \beta_Y)}{\partial y}} \bigg|_{y=e_Y(x)} \frac{\partial G(y; \beta_Y)}{\partial \beta_{Yi}} \bigg|_{y=e_Y(x)}. \quad (14.27)$$

Given Equations 14.18 and 14.19, it is straightforward to obtain the derivatives in the above equations, but their expressions can be quite messy and thus are omitted here.

The general expression for the square of the SEE in Equation 14.20 applies to all designs. However, for designs other than the RG design, it can be quite complicated to calculate the derivatives in Equation 14.22. This matter is beyond the scope of this chapter.

## 14.5 Functions for the Random Groups Design

The wrapper and print functions for CLL equating with the random groups design are in `CLL_Equate.c`, with the function prototypes in `CLL_Equate.h`.

The wrapper function for the random groups design is `Wrapper_RC()`. It calls `CLLEquateEG()`, which in turn calls `CLLEquate()`. Then `Wrapper_RC()` calls `MomentsFromFD()`, which is described on page 20. After `Wrapper_RC()` returns to `main()`, `Print_RC()` can be called. So, the calling sequence is

- `Wrapper_RC()`
  - `CLLEquateEG()`
    - \* `CLLEGcdf()`
    - \* `CLLInverseCdf()`
  - `MomentsFromFD()`
- `Print_RC()`

### 14.5.1 `Wrapper_RC()`

`Wrapper_RC()` calls functions that perform CLL equating for the random groups design. The function prototype for `Wrapper_RC()` is:

```
void Wrapper_RC(char design, char method, char smoothing,
                struct USTATS *x, struct USTATS *y,
                struct ULL_SMOOTH *ullx,
                struct ULL_SMOOTH *ully, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' designating the random groups design;
- ▷ `method` = 'E' for equipercntile equating;
- ▷ `smoothing` = 'Z' for CLL 'smoothing';
- ▷ `x` = pointer to a `USTATS` (new form);
- ▷ `y` = pointer to a `USTATS` (old form);
- ▷ `ullx` = pointer to a `ULL_SMOOTH` structure (new form);
- ▷ `ully` = pointer to a `ULL_SMOOTH` structure (old form); and
- ▷ `rep` = replication number for bootstrap; should be set to 0 for actual equating.

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_RC()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

### 14.5.2 `Print_RC()`

The function prototype for `Print_RC()` is:

```
void Print_RC(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the CLL equating results for the random groups design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

### 14.5.3 *Example*

Table 14.1 is a `main()` function that illustrates CLL equating with the random groups design based on an example discussed by Kolen and Brennan (2004, chap. 2, p. 48). In this example, `Wrapper_Smooth_ULL()` is called twice to perform univariate log-linear smoothing for the  $X$  and  $Y$  distributions, and then `Wrapper_RC()` is called to perform CLL equating. Note that the `scale` parameter in the two calls to `Wrapper_Smooth_ULL()` must set to 0.

The output from `Print_RC()` is provided in Table 14.2. The top of the output provides general information about the log-linear smoothing. The bottom part provides the equated raw scores for the CLL equating, as well as their moments based on using the new-form frequencies. Figure 14.1 shows the plots of kernel, CLL, and percentile-rank based equipercentile equating for these data.

## 14.6 Functions for the Single Group Design

The wrapper function for the CLL equating under the single group design is `Wrapper_SC()`. It calls `CLLEquateSG()`, and then calls `MomentsFromFD()`.



After `Wrapper_SC()` returns to `main()`, `Print_SC()` can be called. So, the calling sequence is

- `Wrapper_SC()`
  - `CLLEquateSG()`
    - \* `CLLBivCdf()`
    - \* `CLLMargInverseYCdf()`
  - `MomentsFromFD()`
- `Print_SC()`

#### 14.6.1 `Wrapper_SC()`

`Wrapper_SC()` calls functions that performs CLL equating for the single group design. The function prototype for `Wrapper_SC()` is:

```
void Wrapper_SC(char design, char method, char smoothing,
                struct BSTATS *xy, struct BLL_SMOOTH *bllxy,
                int rep, struct PDATA *inall,
                struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'S' designating the single group design;
- ▷ `method` = 'E' for equipercetile equating;
- ▷ `smoothing` = 'Z' for CLL 'smoothing';
- ▷ `xy` = pointer to a `BSTATS` (both forms);
- ▷ `bllxy` = pointer to a `BLL_SMOOTH` structure (both form); and
- ▷ `rep` = replication number for bootstrap; should be set to 0 for actual equating.

The output variables are:

- ▷ `inall` = a `PDATA` structure that is populated by `Wrapper_SC()`; and
- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

`Wrapper_SC()` populates the elements of the `PDATA` structure `inall`, dynamically allocates space for the `eraw[][]` and `mts[][]` matrices in the `ERAW_RESULTS` structure `r`, calls a function that does the actual equating and stores results in `r`, and calls the function `MomentsFromFD()` that gets the equated raw-score moments and stores them in `r`.

### 14.6.2 Print\_SC()

The function prototype for `Print_SC()` is:

```
void Print_SC(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the CLL equating results for the single group design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

### 14.6.3 Example

Table 14.3 is a `main()` function that illustrates CLL equating with the single group design based on a dummy example using data in Kolen and Brennan (2004, chap. 2, p. 48). In this example, `Wrapper_Smooth_BLL()` is called to perform bivariate log-linear smoothing for the joint distribution of  $X$  and  $Y$ , and then `Wrapper_SK()` is called to perform CLL equating. Note that the `scale` parameter in `Wrapper_Smooth_BLL()` must set to 0.

The output from `Print_SC()` is provided in Table 14.4. The top of the output provides general information about the log-linear smoothing. The bottom part provides the equated raw scores for the CLL equating, as well as their moments based on using the new-form frequencies.

## 14.7 Functions for the Common-Item Nonequivalent Groups Design

The wrapper function is `Wrapper_CC()`. Depending on the `method` argument, `Wrapper_CC()` calls different functions such as `CLLEquateNEATPS()`, `CLLEquateNEATChn()`, or some other function. Then `Wrapper_CC()` calls `MomentsFromFD()`. After `Wrapper_CC()` returns to `main()`, `Print_CC()` can be called. So, the calling sequence is

- `Wrapper_CC()`
  - `CLLEquateNEATPS()`, or
  - \* `CLLNEATPSMargCdf()`

- \* CLLNEATPSInverseCdf()
- CLLEquateNEATChn()
- \* CLLBivCdf()
- \* CLLMargInverseYCdf()
- \* CLLMargInverseXCdf()
- MomentsFromFD()
- Print\_CC()

#### 14.7.1 Wrapper\_CC()

Wrapper\_CC() calls functions that performs CLL equating for the single group design. The function prototype for Wrapper\_CC() is:

```
void Wrapper_CC(char design, char method, char smoothing,
                double w1, int anchor, double rv1, double rv2,
                struct BSTATS *xv, struct BSTATS *yv,
                struct BLL_SMOOTH *bllxv,
                struct BLL_SMOOTH *bllyv, int rep,
                struct PDATA *inall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ **design** = 'R' designating the random groups design;
- ▷ **method** = see equating methods list below;
- ▷ **smoothing** = 'Z' for CLL “smoothing”;
- ▷ **w1** = synthetic population weight for the new form;
- ▷ **rv1** = reliability of common item set (new form);
- ▷ **rv2** = reliability of common item set (old form);
- ▷ **xv** = pointer to a BSTATS (new form);
- ▷ **yv** = pointer to a BSTATS (old form);
- ▷ **bllxv** = pointer to a BLL\_SMOOTH structure (new form);
- ▷ **bllyv** = pointer to a BLL\_SMOOTH structure (old form); and
- ▷ **rep** = replication number for bootstrap; should be set to 0 for actual equating.

The output variables are:

- ▷ **inall** = a PDATA structure that is populated by Wrapper\_CC(); and

- ▷ `r` = an `ERAW_RESULTS` structure that stores all equated raw-score results.

Two arguments of the wrapper function `Wrapper_CC()` need particular attention. First, `method` must be one of the following:

- `'E'` = FE
- `'F'` = MFE
- `'G'` = FE and MFE
- `'C'` = Chained
- `'H'` = FE and Chained
- `'A'` = FE, MFE, and Chained

Second, if MFE is requested (`'F'`, `'G'`, or `'A'`), then reliability estimates must be provided for `rv1` and `rv2`.

`Wrapper_CC()` populates the elements of the `PDATA` structure `inall`, dynamically allocates space for the `eraw[][]` and `mts[][]` matrices in the `ERAW_RESULTS` structure `r`, calls a function that does the actual equating and stores results in `r`, and calls the function `MomentsFromFD()` that gets the equated raw-score moments and stores them in `r`.

### 14.7.2 `Print_CC()`

The function prototype for `Print_CC()` is:

```
void Print_CC(FILE *fp, char tt[], struct PDATA *inall,
              struct ERAW_RESULTS *r)
```

This function writes the CLL equating results for the common-item nonequivalent groups design design. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ various variables in the `PDATA` structure `inall` (recall that `PDATA` is a structure that is used to pass variables from one function to another);
- ▷ the equated raw-score results in the `ERAW_RESULTS` structure `r`.

### 14.7.3 Example

Table 14.5 is a `main()` function that illustrates CLL equating with the common-item nonequivalent groups design based on an example discussed by Kolen and Brennan (2004, chap. 5, pp. 147–151). Here, however, `w1 = .5`, rather than `w1 = 1` as in the Kolen and Brennan (2004) example. In this example, `Wrapper_Smooth_BLL()` is called twice to perform bivariate log-linear smoothing for the two bivariate distributions, and then `Wrapper_CC()` is called to perform CLL equating. Note that the `scale` parameter in the two calls to `Wrapper_Smooth_BLL()` must set to 0.

The output from `Print_CC()` is provided in Table 14.6. The top of the output provides general information about the log-linear smoothing. The bottom part provides the equated raw scores for the CLL equating, as well as their moments based on using the new-form frequencies.

Figure 14.2 contains plots of the kernel, CLL, and percentile-rank based equatings using frequency estimation. Figure 14.3 contains plots of the kernel, CLL, and percentile-rank based equatings, using the chained equipercentile method.

TABLE 14.1. Main() Code to Illustrate CLL Equating with the Random Groups Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct USTATS x,y;
    struct ULL_SMOOTH ullx, ully;
    struct PDATA pdREZ;
    struct ERAW_RESULTS rREZ;
    struct ESS_RESULTS sREZ;

    FILE *outf;

    outf = fopen("Chap 14 RG out","w");

    /* Random Groups Design:
       Kolen and Brennan (2004): Chapter 2 example:
       Equipercetile equating (see pp. 52, 53, 57, 60) */

    ReadFdGet_USTATS("actmathfreq.dat",1,2,0,40,1,'X',&x);
    ReadFdGet_USTATS("actmathfreq.dat",1,3,0,40,1,'Y',&y);

    Wrapper_Smooth_ULL(&x, 6, 0, 1, 0, .000001, NULL, &ullx);
    Print_ULL(outf, "ACT Math X", &x, &ullx, 2, 2);

    Wrapper_Smooth_ULL(&y, 6, 0, 1, 0, .000001, NULL, &ully);
    Print_ULL(outf, "ACT Math Y", &y, &ully, 2, 2);

    /* CLL equating for Random Groups Design */

    Wrapper_RC('R','E','Z', &x, &y, &ullx, &ully, 0, &pdREZ,&rREZ);
    Print_RC(outf,"ACT Math--CLL--RG",&pdREZ, &rREZ);

    Wrapper_ESS(&pdREZ,&rREZ,0,40,1,"yctmath.TXT",1,1,36,&sREZ);
    Print_ESS(outf,"ACT Math---CLL--RG",&pdREZ,&sREZ);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 14.2. Output for Equated Raw Scores using CLL Equating Under a RG Design

```

*****
ACT Math--CLL--RG

CLL Equating with Random Groups Design and Polynomial Log-Linear Smoothing
Log-Linear Smoothing for new form X: number of degrees of smoothing = 6
Log-Linear Smoothing for old form Y: number of degrees of smoothing = 6
Input file for new form X: actmathfreq.dat
Input file for old form Y: actmathfreq.dat
-----
Raw Score (X)      y-equiv

    0.00000      0.01953
    1.00000      0.31250
    2.00000      1.00586
    3.00000      1.85791
    4.00000      2.74109
    5.00000      3.63190
    6.00000      4.52744
    7.00000      5.42961
    8.00000      6.34140
    9.00000      7.26700
   10.00000      8.21117
   11.00000      9.17919
   12.00000     10.17616
   13.00000     11.20651
   14.00000     12.27318
   15.00000     13.37666
   16.00000     14.51416
   17.00000     15.67959
   18.00000     16.86409
   19.00000     18.05725
   20.00000     19.24881
   21.00000     20.43022
   22.00000     21.59496
   23.00000     22.73911
   24.00000     23.86093
   25.00000     24.96010
   26.00000     26.03775
   27.00000     27.09549
   28.00000     28.13553
   29.00000     29.16031
   30.00000     30.17220
   31.00000     31.17386
   32.00000     32.16766
   33.00000     33.15567
   34.00000     34.13986
   35.00000     35.12161
   36.00000     36.10199
   37.00000     37.08130
   38.00000     38.05908
   39.00000     39.03351
   40.00000     39.99985

      Mean      18.97880
      S.D.       8.93773
      Skew       0.35232
      Kurt       2.14566
*****

```

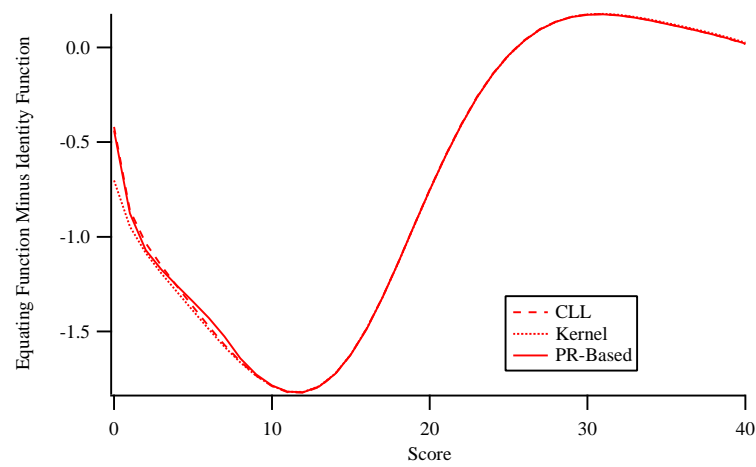


FIGURE 14.1. CLL, Kernel and Percentile-Rank Based Equipercentile Equating Functions under a RG Design



TABLE 14.3. Main() Code to Illustrate CLL Equating with the Single Group Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xy;
    struct BLL_SMOOTH bllxy;
    struct PDATA pdSEZ;
    struct ERAW_RESULTS rSEZ;
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm3[3][2] = {{1,1},{1,2},{2,1}};

    FILE *outf;

    outf = fopen("Chap 14 SG out","w");

    /* Single Groups. Dummy example using Chapter 4 bstats for old form
       with v serving as x in struct BSTATS xy */

    convertFtoW("mondaty.dat",2,fieldsACT,"mondaty-temp");
    ReadRawGet_BSTATS("mondaty-temp",2,1,0,12,1,0,36,1,'X','Y",&xy);
    Print_BSTATS(outf,"Ch 4 dummy: v->x and y from ch 4 serve as x and y",&xy,0);

    /* log-linear smoothing of the bivariate distribution */

    Wrapper_Smooth_BLL(&xy, 0, 6, 6, 3, cpm3, 0, 1, 0, .000001, NULL, &bllxy);

    /* CLL equating for Single Groups Design */

    Wrapper_SC('S','E','Z', &xy, &bllxy, 0, &pdSEZ, &rSEZ);
    Print_SC(outf,"CLL SG design: v->x and y from ch 4 serve as x and y",
             &pdSEZ,&rSEZ);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 14.4. Output for Equated Raw Scores using CLL Equating Under a SG Design

```

*****
CLL SG design: v->x and y from ch 4 serve as x and y

Input filename: mondaty-temp

CLL Equating with Single Group Design
and Polynomial Log-Linear Smoothing

Bivariate Log-Linear Smoothing using the Multinomial Model

Number of score categories for X = 13
Number of score categories for Y = 37
Total number of score categories = 481

Number of persons (i.e., total of frequencies) = 1638

Polynomial degree for X = 6
Polynomial degree for Y = 6
Number of cross-products = 3
Cross-Product moments: (X1,Y1), (X1,Y2), (X2,Y1)

-----

Equated Raw Scores

Raw Score (X)      Method 0:
                   y-equiv

    0.00000      4.07983
    1.00000      6.15295
    2.00000      8.23849
    3.00000     10.58148
    4.00000     13.18417
    5.00000     15.95821
    6.00000     18.83557
    7.00000     21.81992
    8.00000     24.91040
    9.00000     27.94104
   10.00000     30.61487
   11.00000     32.82349
   12.00000     34.82886

    Mean         18.67162
    S.D.          6.87848
    Skew          0.20528
    Kurt          2.30336

*****

```

TABLE 14.5. Main() Code to Illustrate CLL Equating with the CINEG Design

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct BSTATS xv,yv;
    struct BLL_SMOOTH bllxv, bllyv;
    struct PDATA pdCHZ;
    struct ERAW_RESULTS rCHZ;
    struct ESS_RESULTS sCHZ;
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};
    int cpm1[1][2] = {{1,1}};

    FILE *outf;

    outf = fopen("Chap 14 CG out","w");

    /* CG Design: Kolen and Brennan (2004, chap. 5) example with w1=.5:
       Equipercetile equating with log-linear smoothing;
       Note: Kolen and Brennan (2004, p. 151) uses w1=1 */

    convertFtoW("mondatx.dat",2,fieldsACT,"mondatx-temp");
    ReadRawGet_BSTATS("mondatx-temp",1,2,0,36,1,0,12,1,'X','V',&xv);

    convertFtoW("mondaty.dat",2,fieldsACT,"mondaty-temp");
    ReadRawGet_BSTATS("mondaty-temp",1,2,0,36,1,0,12,1,'Y','V',&yv);

    Print_BSTATS(outf,"xv",&xv,1);
    Print_BSTATS(outf,"yv",&yv,1);

    Wrapper_Smooth_BLL(&xv, 1, 6, 6, 1, cpm1, 0, 1, 0, .000001, NULL, &bllxv);
    Print_BLL(outf, "Chapter 5: log-linear smoothing of x and v in pop 1",
               &xv, &bllxv, 0, 2, 2, 1);

    Wrapper_Smooth_BLL(&yv, 1, 6, 6, 1, cpm1, 0, 1, 0, .000001, NULL, &bllyv);
    Print_BLL(outf, "Chapter 5: log-linear smoothing of y and v in pop 2",
               &yv, &bllyv, 0, 2, 2, 1);

    /* CLL equating for CINEG Design */

    Wrapper_CC('C','H','Z', .5, 0, 0, 0, &xv,&yv, &bllxv,&bllyv,
               0,&pdCHZ,&rCHZ);

    Print_CC(outf,"Ch 5 Data: w1 = .5, CLL CINEG",&pdCHZ,&rCHZ);
    Print_SynDens(outf,"Ch 5 Data: w1 = .5, CLL CINEG",&pdCHZ,&rCHZ);

    Wrapper_ESS(&pdCHZ,&rCHZ,-11,54,1,"Conversion-reordered.DAT",2,100,850,&sCHZ);
    Print_ESS(outf,"Ch 5 Data: w1 = .5, CLL CINEG",&pdCHZ,&sCHZ);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 14.6. Output for Equated Raw Scores using CLL Equating Under a CINEG Design

```

*****
Ch 5 Real Data: w1 = .5, CLL with log-linear smoothing

CLL Equating with CINEG Design (External Anchor; w1 = 0.50000)
and Polynomial Log-Linear Smoothing
Input file for XV and pop 1: mondatx-temp
Input file for YV and pop 2: mondaty-temp

Number of persons for X = 1655
Number of score categories for X = 37; Number of score categories for V = 13
Polynomial degree for X = 6; Polynomial degree for V = 6
Number of cross-products XV = 1; Cross-Product moments: ( X1,V1)

Number of persons for Y = 1638
Number of score categories for Y = 37; Number of score categories for V = 13
Polynomial degree for Y = 6; Polynomial degree for V = 6
Number of cross-products YV = 1; Cross-Product moments: ( Y1,V1)

Raw Score (X)          FE      ChainedE

    0.00000      0.07813      0.65625
    1.00000      1.23438      1.23438
    2.00000      2.31836      2.42676
    3.00000      3.36621      3.53784
    4.00000      4.41406      4.58118
    5.00000      5.46756      5.59290
    6.00000      6.52219      6.59106
    7.00000      7.58020      7.58415
    8.00000      8.64442      8.57724
    9.00000      9.71317      9.57541
   10.00000     10.78642     10.58148
   11.00000     11.86307     11.59546
   12.00000     12.93887     12.61734
   13.00000     14.01466     13.64825
   14.00000     15.08397     14.68425
   15.00000     16.14481     15.72222
   16.00000     17.19463     16.76076
   17.00000     18.22978     17.79449
   18.00000     19.24884     18.82174
   19.00000     20.25153     19.83995
   20.00000     21.23840     20.84828
   21.00000     22.21060     21.84814
   22.00000     23.17151     22.83784
   23.00000     24.12395     23.82133
   24.00000     25.07356     24.80652
   25.00000     26.02261     25.79283
   26.00000     26.97787     26.78253
   27.00000     27.93935     27.78578
   28.00000     28.90872     28.79242
   29.00000     29.88431     29.82051
   30.00000     30.85989     30.83844
   31.00000     31.83209     31.84790
   32.00000     32.80768     32.87091
   33.00000     33.71326     33.69745
   34.00000     34.71143     34.67078
   35.00000     36.13867     35.32568
   36.00000     36.43225     36.35095

      Mean      16.80244      16.52780
      S.D.       6.64321      6.61429
      Skew       0.48463      0.55926
      Kurt       2.59369      2.66671
*****

```

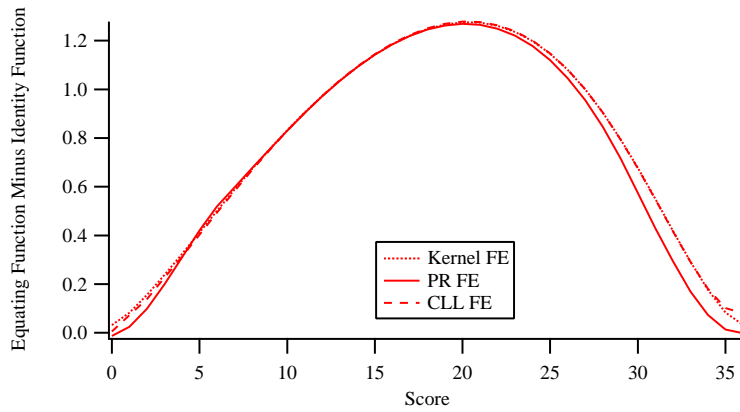


FIGURE 14.2. Kernel, percentile-rank based, and CLL equating functions using the frequency estimation method under a CINEG design.

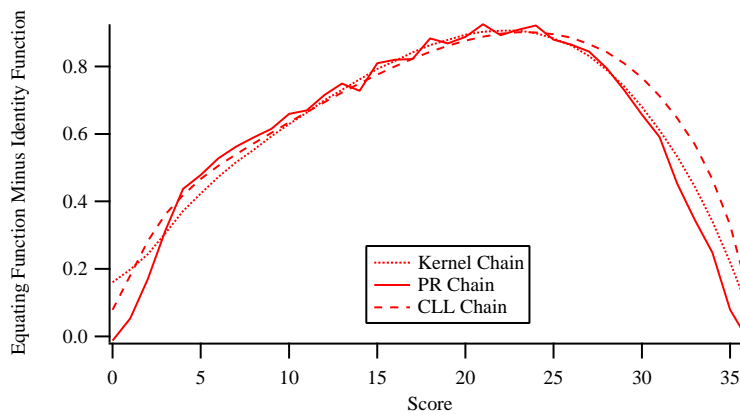


FIGURE 14.3. Kernel, percentile-rank based, and CLL equating functions using the chained equipercentile method under a CINEG design.



## Scale Transformation using Item Response Theory

Item response theory (IRT) is often used to equate forms of a test. Although examinees' performance on a test item may well be subject to multiple factors, it is often assumed that there is a single dominant factor underlying the performance so that unidimensional IRT models can be used. Accordingly, this chapter assumes that the scale used for measuring ability (denoted as  $\theta$ ) is unidimensional.

A critical step in the equating process using unidimensional IRT is to achieve comparability between item parameters on two scales that have been separately determined with different groups of examinees. This step typically consists of finding a linear transformation between the scales and then transforming IRT parameters on one scale to the other. When dealing with parameter estimates from sample data, however, the linear transformation should be properly estimated using a scale transformation method because of sampling error and possible model misfit. Kolen and Brennan (2004) describe four scale transformation methods that can be used for single-format tests (involving a single dichotomous or polytomous IRT model) under the common-item nonequivalent groups design. This chapter describes the four scale transformation methods more generally, so that they can be implemented for mixed-format tests as well as single-format tests. It should be noted that the notational schemes used in this chapter are slightly different from those used in Kolen and Brennan (2004).<sup>1</sup>

---

<sup>1</sup>This chapter is based largely on Kim and Kolen (2005)

## 15.1 IRT Models

Items may be scored dichotomously in two categories of incorrect versus correct, or scored polytomously in multiple ordered-response categories. The scoring scheme in a test is largely dependent on both the purpose of the test and the assessment domain being measured. A mixed-format test uses both dichotomously-scored and polytomously-scored items. For generality, this mixed scoring is assumed for a test throughout this chapter. To deal with both dichotomous and polytomous items consistently, the incorrect and correct responses of a dichotomous item are respectively referred to as the first and second response categories.

A unidimensional IRT model relates the probability of an examinee responding in each category of an item to his/her ability. The probabilistic relation, over the ability continuum  $-\infty < \theta < \infty$ , is realized through the category characteristic curve. The characteristic curve for category  $k$  of item  $j$  with  $m_j$  categories is symbolized as  $P_{jk}(\theta)$ , which represents the probability that a randomly selected examinee of ability  $\theta$  earns a score in category  $k$  of item  $j$ .

The probability represented by the category characteristic curve can be defined in various ways depending on how item parameters are conceptualized and what type of functional forms are used to characterize examinees' responses to an item. These various ways of defining the category characteristic curve lead to different IRT models. Kolen and Brennan (2004) describe four IRT models that can be used to model items that are scored in two or more ordered response categories: the three-parameter logistic (3PL), graded response (GR), nominal response (NR), and generalized partial credit (GPC) models. Each of these four IRT models is briefly described next. As in Kolen and Brennan (2004), response categories are designated using consecutive integers beginning with 1.

### 15.1.1 Three-Parameter Logistic Model

Birnbaum's (1968) 3PL model has been widely used as the most general form to model dichotomously-scored items. Under the 3PL model, the characteristic curve for the correct response (category 2) of item  $j$  is defined as

$$P_{j2}(\theta) = P(\theta; a_j, b_{j2}, c_j) = c_j + (1 - c_j) \frac{\exp[Da_j(\theta - b_{j2})]}{1 + \exp[Da_j(\theta - b_{j2})]} \quad (15.1)$$

where  $a_j$  is a discrimination parameter,  $b_{j2}$  is a difficulty parameter,  $c_j$  is a lower asymptote, and  $D$  is a scaling constant (typically 1.7). The expression "exp" in Equation 15.1 stands for the exponential function  $e^x$ , so the bracketed component corresponds to  $x$ . Note that the first category (i.e., incorrect response) does not have a difficulty parameter and the charac-



teristic curve for the first category is symbolized as  $P_{j1}(\theta)$ , which equals  $1 - P_{j2}(\theta)$ .

### 15.1.2 Graded Response Model

Samejima's (1969, 1997) GR model can be used to model item responses that are scored polytomously using ordered categories. Although originally developed as a normal ogive model, the GR model also has been presented as a logistic model, which is considered here, because of its closed form. The GR model is a difference model that requires a two-step process: (1) evaluating cumulative category characteristic curves and then (2) calculating the difference between adjacent cumulative category characteristic curves. The cumulative characteristic curve for category  $k$  of item  $j$  with  $m_j$  categories,  $\tilde{P}_{jk}(\theta)$ , represents the probability that a randomly selected examinee of ability  $\theta$  earns a score at or above category  $k$  of item  $j$ . Formally,

$$\tilde{P}_{jk}(\theta) = \tilde{P}(\theta; a_j, b_{jk}) = \begin{cases} 1 & k = 1 \\ \frac{\exp[Da_j(\theta - b_{jk})]}{1 + \exp[Da_j(\theta - b_{jk})]} & 2 \leq k \leq m_j, \\ 0 & k > m_j \end{cases}, \quad (15.2)$$

where  $a_j$  is a discrimination parameter,  $b_{jk}$  ( $k = 2, \dots, m_j$ ) are difficulty or location parameters, and  $D$  is a scaling constant (typically 1.7). Note that the first category does not have a difficulty parameter, as is the case for the 3PL model. The category characteristic curve is then given by the difference between two adjacent cumulative probabilities as follows:

$$P_{jk}(\theta) = P(\theta; a_j, b_{jk}, b_{j(k+1)}) = \tilde{P}_{jk}(\theta) - \tilde{P}_{j(k+1)}(\theta). \quad (15.3)$$

### 15.1.3 Nominal Response Model

Bock's (1972) NR model is a general divide-by-total model that can be used to model polytomous items whose response categories are not necessarily ordered. For the NR model, the category characteristic curve is expressed as

$$P_{jk}(\theta) = P(\theta; a_{j1}, \dots, a_{jm_j}, c_{j1}, \dots, c_{jm_j}) = \frac{\exp(a_{jk}\theta + c_{jk})}{\sum_{h=1}^{m_j} \exp(a_{jh}\theta + c_{jh})}, \quad (15.4)$$

where  $a_{jk}$  and  $c_{jk}$  ( $k = 1, 2, \dots, m_j$ ) are discrimination and intercept parameters for category  $k$  of item  $j$  with  $m_j$  categories. Since Equation 15.4 is invariant with respect to a translation of the term  $a_{jk}\theta + c_{jk}$  in both the numerator and denominator, there are sometimes two constraints imposed for model identification:

$$\sum_{k=1}^{m_j} a_{jk} = 0 \quad \text{and} \quad \sum_{k=1}^{m_j} c_{jk} = 0. \quad (15.5)$$

### 15.1.4 Generalized Partial Credit Model

Muraki's (1992, 1997) GPC model is a generalization of Masters' (1982) partial credit model, which is appropriate for the analysis of responses from performance assessments or ratings that are recorded in two or more ordered categories. The GPC model states that the category characteristic curve is given by

$$P_{jk}(\theta) = P(\theta; a_j, b_{j1}, \dots, b_{jk}, \dots, b_{jm_j}) \quad (15.6)$$

$$= \frac{\exp \left[ \sum_{v=1}^k Da_j(\theta - b_{jv}) \right]}{\sum_{h=1}^{m_j} \exp \left[ \sum_{v=1}^h Da_j(\theta - b_{jv}) \right]},$$

where  $a_j$  is a discrimination parameter,  $b_{jk}$  ( $k = 1, 2, \dots, m_j$ ) are item-category parameters, and  $D$  is a scaling constant (typically 1.7). The GPC model is over-parameterized and thus  $b_{j1}$  is often arbitrarily defined as 0. The item-category parameter  $b_{jk}$  is sometimes resolved into two parameters, a location parameter  $b_j$  and a category parameter  $d_{jk}$  such that

$$b_{jk} = b_j - d_{jk}, \quad (15.7)$$

to be suitable for a Likert-type item. As described by Muraki (1992), the NR model becomes equivalent to the GPC model if the following conditions are satisfied:

$$a_{jk} = kDa_j \quad \text{and} \quad c_{jk} = -Da_j \sum_{v=1}^k b_{jv}. \quad (15.8)$$

## 15.2 IRT Scale Transformation Methods

It is well known that the  $\theta$ -scale is undetermined up to a linear transformation (Lord, 1980). The practical meaning of this indeterminacy is that, to maintain the same fit of model to data, the numerical values of IRT item parameters should be properly expressed on the chosen  $\theta$ -scale that has an arbitrary origin (usually 0) and unit (usually 1). Under the marginal maximum likelihood estimation (MMLE) method (e.g., Bock & Aitkin, 1981), the choice of an origin and unit of a  $\theta$ -scale is usually based on the mean and standard deviation of the ability distribution for the group of examinees. The chosen "0, 1" scale, of course, can still be linearly transformed in an "offsetting" way such that both item and ability parameters are simultaneously changed so as to keep the same degree of model-to-data fit.

However, consider a situation where separate IRT estimation runs are conducted in each of two or more sampled groups of examinees, with their respective "0, 1" scales being used. Because the resulting scales are usually

not expected to be equivalent due to differences in ability of the sampled groups, the scales need to be linked through a linear scale transformation to each other so that all parameter estimates may be placed on a common scale. At this point, the IRT invariance property plays a major role in linking scales from such separate estimations, because it states that item parameters are invariant across groups. The invariance property suggests that scale linking through linear transformation can be done as long as a set of common items provides the link between two samples of data that are obtained from separate administrations of a test. Of course, the statistical method to “estimate” the linear scale transformation should take into account the sampling error and possible model misfit. The statistical method is referred to here as the scale transformation method.

### 15.2.1 Overview of IRT Scale Transformation

Suppose that a set of items is independently administered to two groups of examinees, old and new. Assume for generality that the  $n$  items comprise a mixed-format test and thus their responses are dichotomously or polytomously scored. Also assume that two separate “0, 1” scales,  $\theta_O$  (old scale) and  $\theta_N$  (new scale), determined respectively from the old and new groups, are used for test calibration. (Note that, in Kolen & Brennan, 2004,  $\theta_O$  and  $\theta_N$  are designated as  $\theta_J$  and  $\theta_I$ , respectively.) Then, the task is to find a linear function between  $\theta_O$  and  $\theta_N$  such as

$$\theta_O = A\theta_N + B, \quad (15.9)$$

so that the two “0, 1” scales may be compared and used interchangeably. The inverse of Equation 15.9 is

$$\theta_N = (\theta_O - B)/A. \quad (15.10)$$

When the scaling constants  $A$  (slope) and  $B$  (intercept) satisfy both Equation 15.9 and 15.10, the transformation is called “symmetric” (see Kim & Kolen, 2005, for more details).

Scale transformation methods attempt to find the appropriate slope and intercept of the linear function, which is expected to minimize linking error occurring during the process of linear transformation. Many scale transformation methods have been developed in the literature (see, e.g., Kim & Kolen, 2005, for a review). This chapter focuses on four scale transformation methods: (1) mean/sigma (Marco, 1977), (2) mean/mean (Loyd & Hoover, 1980), (3) Haebara (Haebara, 1980), and (4) Stocking-Lord (Stocking & Lord, 1983) methods. The first two methods are referred to as moment methods, and they attempt to find the scaling constants by matching two sets of item parameter estimates from separate calibrations of the  $n$  “common” items. In contrast, the Haebara and Stocking-Lord methods, referred to as characteristic curve methods, are based on the idea of matching two

separate sets of estimated item-category/test characteristic curves from the common items. The moment methods are symmetric in nature. Haebara (1980) provided a symmetric version of his method. Stocking and Lord (1983) originally presented a non-symmetric scale transformation method, but here a general symmetric version of their method is presented. The methodology for each of the four methods is provided below.

### 15.2.2 Moment Methods

Development of the moment methods requires specifying linear relations between old and new item parameters given the linear function  $\theta_O = A\theta_N + B$  in Equation 15.9.<sup>2</sup> The linear relations differ by IRT model. Under the 3PL, GR, and GPC models,

$$a_{jO} = a_{jN}/A \quad (15.11)$$

and

$$b_{jkO} = Ab_{jkN} + B. \quad (15.12)$$

Notice in Equations 15.11 and 15.12 that the subscripts  $O$  and  $N$  are added to the notation. Note that the lower asymptote  $c_j$  under the 3PL model is not dependent on the ability scale because it is on the probability metric (Stocking & Lord, 1983). Also note that under the GPC model, if the  $b_{jk}$  parameter is replaced with the  $b_j$  and  $d_{jk}$  parameters, as shown in Equation 15.7, the linear relations for the decomposed parameters between the old and new scales are

$$b_{jO} = Ab_{jN} + B \quad \text{and} \quad d_{jkO} = Ad_{jkN}.$$

Under the NR model,

$$a_{jkO} = a_{jkN}/A \quad (15.13)$$

and

$$c_{jkO} = c_{jkN} - (B/A)a_{jkN}. \quad (15.14)$$

If a reparameterization of  $a_{jk}\theta + c_{jk} = a_{jk}(\theta - b_{jk})$ , where  $b_{jk} = -c_{jk}/a_{jk}$ , is made, Equation 15.14 can be re-expressed in terms of  $b_{jk}$  as in Equation 15.12. Note that Equations 15.11 through 15.14 must hold for every item as long as the IRT models used fit item response data from the population under consideration. However, this is not the case for sample data (more specifically, for item parameter estimates) because of sampling error and possible model misfit.

---

<sup>2</sup>Note that, beginning with Equation 15.10, a discussion similar to the following can be made and lead to the same solutions, which qualifies the moment methods for “symmetric” methods.

The moment methods use summary statistics over item parameter estimates to estimate the linear function with sample data. The rationale underlying the moment methods is to express Equations 15.11 through 15.13 for an item in terms of a group of items. For example, taking the mean over item discriminations ( $a_j$  and  $a_{jk}$ ) of the  $n$  common items based on Equations 15.11 and 15.13 and then expressing the resulting relationship with respect to  $A$ ,

$$A = \frac{M(a_N)}{M(a_O)}. \quad (15.15)$$

Here,  $M(\cdot)$  is the operator for the arithmetic mean,  $a_N$  represents all of the discrimination parameters on the new scale, and  $a_O$  is the counterpart on the old scale. As another example, taking the mean and standard deviation over item difficulties ( $b_{jk}$ ) of the  $n$  items based on Equation 15.12 and then expressing the resulting relationships with respect to  $A$  and  $B$ ,

$$A = \frac{SD(b_O)}{SD(b_N)} \quad (15.16)$$

and

$$B = M(b_O) - AM(b_N), \quad (15.17)$$

where  $SD(\cdot)$  is the operator for the standard deviation. In Equations 15.16 and 15.17,  $b_N$  and  $b_O$  each include all of the difficulty parameters on the new or old scale *except* arbitrary ones such as  $b_{j1N}$  and  $b_{j1O}$  under the GPC model. Note that Equations 15.16 and 15.17 assume that the intercept parameter  $c_{jk}$  under the NR model is replaced with  $b_{jk}$  through the reparameterization  $a_{jk}\theta + c_{jk} = a_{jk}(\theta - b_{jk})$ .

*Mean/Sigma Method.* The mean/sigma (MS) method, described by Marco (1977), uses Equations 15.16 and 15.17 to estimate the scaling constants  $A$  and  $B$  as follows;

$$\hat{A}_{MS} = \frac{SD(\hat{b}_O)}{SD(\hat{b}_N)} \quad (15.18)$$

and

$$\hat{B}_{MS} = M(\hat{b}_O) - \hat{A}_{MS}M(\hat{b}_N), \quad (15.19)$$

where the symbol  $\hat{\cdot}$  is used to indicate estimates. Notice that two sets of item difficulty estimates from separate calibrations are used in Equations 15.18 and 15.19.

*Mean/Mean Method.* The mean/mean (MM) method (Loyd & Hoover, 1980) considers the item discriminations as well as the item difficulties. Based on Equations 15.15 and 15.17, the mean/mean estimates of  $A$  and  $B$  are

$$\hat{A}_{MM} = \frac{M(\hat{a}_N)}{M(\hat{a}_O)} \quad (15.20)$$

and

$$\hat{B}_{MM} = M(\hat{b}_O) - \hat{A}_{MM}M(\hat{b}_N). \quad (15.21)$$

Several issues may be addressed for the moment methods dealing with scale transformation with mixed-format tests. Among such issues, Kim and Lee (2006) pointed out the following two (see Kim & Kolen, 2005, for more details). First, appropriate attention should be given to the constraints imposed for model identification when the NR model is employed for mixed-format tests. If the constraints as shown in Equation 15.5 are imposed for the identification of the NR model, the sum of  $a_N$ -parameter estimates from the new scale and the counterpart from the old scale are zero and they do not contribute to the calculation of  $M(\hat{a}_N)$  and  $M(\hat{a}_O)$  in the mean/mean method. Second, when the constraints are used, the estimates of the reparameterized  $b_{jk} = -c_{jk}/a_{jk}$  might be unstable, though the reparameterization per se is theoretically legitimate. The instability can increase when both  $\hat{a}_{jk}$  and  $\hat{c}_{jk}$  are near zero, and thus the resulting  $\hat{b}_{jk}$  can be numerically unstable in magnitude and sign. In practice, the zero-sum constraints are typically used, and thus the moment methods are not recommended when some of the common items in a mixed-format test are calibrated using the NR model.

### 15.2.3 Characteristic Curve Methods

Estimates of the slope  $A$  and intercept  $B$  can be evaluated by minimizing criterion functions (or, so-called loss functions) defined based on the difference between estimated characteristic curves of the common items rather than item parameter estimates (Kim & Kolen, 2005). One potential problem with moment methods arises when two very different sets of item parameter estimates for an item produce almost identical item category characteristic curves. In this situation, characteristic curves, which consider all of the parameter estimates for an item simultaneously, can be used instead of item parameter estimates to obtain more stable estimates of  $A$  and  $B$  (Kim & Kolen, 2005; Kolen & Brennan, 2004).

The rationale for the characteristic curve methods begins with specifying scale linking through linear transformation in terms of item-category/test characteristic curves. By the invariance property of IRT, for category  $k$  of item  $j$ , two category characteristic curves,  $P_{jk}(\theta_N)$  on the new scale and  $P_{jk}(\theta_O)$  on the old scale, should perfectly overlap when they are linearly transformed to each other. Given the linear function  $\theta_O = A\theta_N + B$  or  $\theta_N = (\theta_O - B)/A$ , let  $P_{jkO}(\theta_O)$  and  $P_{jkN}(\theta_N)$  be the characteristic curves for category  $k$  of item  $j$  expressed respectively on  $\theta_O$  and  $\theta_N$ , with their respective “original” parameters placed on their own scales. Under the 3PL model, for example,

$$P_{j2O}(\theta_O) = P(\theta_O; a_{jO}, b_{j2O}, c_j) \quad \text{and} \quad P_{j2N}(\theta_N) = P(\theta_N; a_{jN}, b_{j2N}, c_j).$$

Further, let  $P_{jkN}^*(\theta_O)$  and  $P_{jkO}^\#(\theta_N)$  be the transformed category characteristic curves expressed respectively on  $\theta_O$  and  $\theta_N$ , with the parameters transformed from each other's scale. The scale transformation (new-to-old or old-to-new) of item parameters varies by IRT model (Kim & Lee, 2006). Under the 3PL, GR, and GPC models, from Equations 15.11 and 15.12, item parameters on the new scale are transformed to the old scale by

$$a_{jN}^* = a_{jN}/A \quad \text{and} \quad b_{jkN}^* = Ab_{jkN} + B,$$

and those on the old scale to the new scale by

$$a_{jO}^\# = Aa_{jO} \quad \text{and} \quad b_{jkO}^\# = (b_{jkO} - B)/A.$$

Under the NR model, from Equations 15.13 and 15.14, the new-to-old transformation is obtained using

$$a_{jkN}^* = a_{jkN}/A \quad \text{and} \quad c_{jkN}^* = c_{jkN} - (B/A)a_{jkN},$$

and the old-to-new transformation is obtained using

$$a_{jkO}^\# = Aa_{jkO} \quad \text{and} \quad c_{jkO}^\# = c_{jkO} + Ba_{jkO}.$$

Therefore, for example, the 3PL transformed category characteristic curves are

$$P_{j2N}^*(\theta_O) = P(\theta_O; a_{jN}^*, b_{j2N}^*, c_j) \quad \text{and} \quad P_{j2O}^\#(\theta_N) = P(\theta_N; a_{jO}^\#, b_{j2O}^\#, c_j).$$

Now, the perfect scale linking on the old scale means that

$$P_{jkO}(\theta_O) = P_{jkN}^*(\theta_O). \quad (15.22)$$

For the new scale, perfect scale linking means that

$$P_{jkN}(\theta_N) = P_{jkO}^\#(\theta_N). \quad (15.23)$$

Perfect scale linking can also be applied to test characteristic curves. A scoring function is needed to define the test characteristic curve for an  $n$ -item mixed-format test. Let  $U_{jk}$  refer to the integer score associated with category  $k$  of item  $j$ . Two scoring functions that are often used are  $U_{jk} = k - 1$  and  $U_{jk} = k$ . The test characteristic curve, then, is defined as

$$T(\theta) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jk} P_{jk}(\theta). \quad (15.24)$$

When perfect linking between  $\theta_O$  and  $\theta_N$  occurs, from Equations 15.22 and 15.23, the following relationships hold:

$$T_O(\theta_O) = T_N^*(\theta_O) \quad (15.25)$$

or

$$T_N(\theta_N) = T_O^\#(\theta_N), \quad (15.26)$$

where

$$T_O(\theta_O) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jk} P_{jkO}(\theta_O), \quad T_N^*(\theta_O) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jk} P_{jkN}^*(\theta_O),$$

$$T_N(\theta_N) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jk} P_{jkN}(\theta_N), \quad \text{and} \quad T_O^\#(\theta_N) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jk} P_{jkO}^\#(\theta_N).$$

As described earlier, however, with item parameter estimates, Equations 15.22 and 15.23 typically do not hold for all  $n$  items because of sampling error and model misfit. In this case, Equations 15.25 and 15.26 do not hold either. Therefore, the scaling constants  $A$  and  $B$  are estimated so that the equations hold as nearly as possible. An objective way of solving the problem is to define a criterion function as a measure of seriousness of the difference between the left- and right-hand members, for example, in either Equation 15.22 or 15.23 and to find values of  $A$  and  $B$  that minimize the criterion function. If an approach to the problem is based on Equations 15.22 and/or 15.23, it is referred to as the Haebara method (see Haebara, 1980). If an approach to the problem is based on Equations 15.25 and/or 15.26, it is referred to as the Stocking-Lord method (see Stocking & Lord, 1983).

*Haebara Method.* More specifically, the Haebara scaling constant estimates are the  $A$  and  $B$  that minimize the following criterion function

$$Q^*(A, B) = Q_1^*(A, B) + Q_2^*(A, B), \quad (15.27)$$

where

$$Q_1^*(A, B) = \frac{1}{S} \int_{-\infty}^{+\infty} \left\{ \sum_{j=1}^n \sum_{k=1}^{m_j} [\hat{P}_{jkO}(\theta_O) - \hat{P}_{jkN}^*(\theta_O)]^2 \right\} \psi_1(\theta_O) d\theta_O$$

and

$$Q_2^*(A, B) = \frac{1}{S} \int_{-\infty}^{+\infty} \left\{ \sum_{j=1}^n \sum_{k=1}^{m_j} [\hat{P}_{jkN}(\theta_N) - \hat{P}_{jkO}^\#(\theta_N)]^2 \right\} \psi_2(\theta_N) d\theta_N,$$

where  $S = \sum_{j=1}^n m_j$ ,  $\psi_1(\theta_O)$  is the probability density of  $\theta_O$ , and  $\psi_2(\theta_N)$  is the probability density of  $\theta_N$ . Notice that the symbol  $\hat{\phantom{x}}$  is used to indicate that item parameter estimates are used. The criterion function  $Q^*$  is a *nonlinear* equation in two unknowns,  $A$  and  $B$ , because the transformed item parameter estimates are functions of the  $A$  and  $B$ . The quantity  $S$



is a factor to standardize the criterion function. Since it does not affect the solutions,  $S$  can be ignored. A noteworthy point about Equation 15.27 is that in the case of dichotomous items the incorrect response categories contribute to the criterion function. As shown by Kim and Kolen (2005), when only a single dichotomous model (e.g., the 3PL model) is used for a test calibration, the criterion function  $Q^*$  simplifies to the function suggested by Haebara (1980), in which only the correct response categories are considered. The first function  $Q_1^*$  takes into account the linking error on the old scale  $\theta_O$ , and the second function  $Q_2^*$  takes into account the linking error on the new scale  $\theta_N$ . Thus, the estimates of  $A$  and  $B$  are obtained in such a way that the linking error on both scales is simultaneously taken into account.

To implement the Haebara method in practice requires a technique to perform the integration in Equation 15.27. One possibility is a form of numerical integration that approximates  $Q^*$ . A useful form of  $Q^*$  is

$$Q^*(A, B) \approx Q(A, B) = Q_1(A, B) + Q_2(A, B), \quad (15.28)$$

where

$$Q_1(A, B) = \frac{1}{S_1} \sum_{i=1}^{G_O} \left\{ \sum_{j=1}^n \sum_{k=1}^{m_j} \left[ \hat{P}_{jkO}(\theta_{iO}) - \hat{P}_{jkN}^*(\theta_{iO}) \right]^2 \right\} W_1(\theta_{iO})$$

and

$$Q_2(A, B) = \frac{1}{S_2} \sum_{i=1}^{G_N} \left\{ \sum_{j=1}^n \sum_{k=1}^{m_j} \left[ \hat{P}_{jkN}(\theta_{iN}) - \hat{P}_{jkO}^\#(\theta_{iN}) \right]^2 \right\} W_2(\theta_{iN}).$$

In Equation 15.28,  $\theta_{iO}$  ( $i = 1, 2, \dots, G_O$ ) and  $W_1(\theta_{iO})$  are ability points and weights intended to reflect the distribution of  $\theta_O$ ;  $\theta_{iN}$  ( $i = 1, 2, \dots, G_N$ ) and  $W_2(\theta_{iN})$  are ability points and weights for the  $\theta_N$  distribution;

$$S_1 = \sum_{i=1}^{G_O} W_1(\theta_{iO}) \cdot \sum_{j=1}^n m_j; \text{ and } S_2 = \sum_{i=1}^{G_N} W_2(\theta_{iN}) \cdot \sum_{j=1}^n m_j.$$

Notice that the standardization factor  $S$  in Equation 15.27 is subdivided into  $S_1$  and  $S_2$ , so that different standardization schemes may be used for  $Q_1$  and  $Q_2$ .

Because the criterion function  $Q$  is nonlinear with respect to  $A$  and  $B$ , a two-dimension multivariate minimization algorithm is needed. Let  $\mathbf{x} = (A, B)^T$  and  $\nabla Q(\mathbf{x})$  be the gradient (i.e., first partial derivatives) of  $Q$  at  $\mathbf{x}$ . Formally,

$$\nabla Q(\mathbf{x}) = \left[ \frac{\partial Q}{\partial A}(\mathbf{x}), \frac{\partial Q}{\partial B}(\mathbf{x}) \right]^T. \quad (15.29)$$

The essential components of the gradient are the first partial derivatives of the transformed category characteristic curves with respect to  $A$  and  $B$ . Kim and Kolen (2005) provide these partial derivatives under each of the four IRT models considered in this chapter. Then, the desired algorithm would be a simple application of the Newton method (also known as the Newton-Raphson method) to the system  $\nabla Q(\mathbf{x}) = 0$  of two nonlinear equations in two unknowns (Dennis & Schnabel, 1996). In this case, the Newton method leads to the following iteration formula:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{H}^{-1} \cdot \nabla Q(\mathbf{x}_i), \quad i = 0, 1, 2, \dots \quad (15.30)$$

In Equation 15.30,  $\mathbf{x}_i$  is the  $i$ th iterate, and  $\mathbf{H}^{-1}$  is the inverse of the Hessian matrix (i.e., the second partial derivative matrix) of the criterion function  $Q$  at  $\mathbf{x}_i$ . The iteration continues until a convergence between iterates is obtained. In practice, the Newton method is often replaced with the quasi-Newton method. The “quasi” in quasi-Newton indicates that the actual Hessian matrix  $\mathbf{H}$  is not used; rather, an approximation of it, denoted as  $\mathbf{A}$ , is used (Press, Teukolsky, Vetterling, & Flannery, 1992).  $\mathbf{A}$  is intended to be positive definite so as to make sure that the iterates always move in a downhill direction during the iterative solution-search process. A formal description of the justification for the use of the approximate Hessian matrix is found in Dennis and Schnabel (1996) and Press et al. (1992).

In practice, criterion functions for the Haebara method have been defined as various special cases of  $Q$  in Equation 15.28. The various cases can be characterized by two schemes for defining criterion functions (Kim & Kolen, 2007). One is the symmetry-related scheme that determines which ability scale, old or new, should be used to capture linking error occurring in the process of transforming scales. The other is the summation scheme over examinees (i.e., ability points) that is defined by the degree to which the ability distribution of the groups considered is reflected in weighting the squared differences between category characteristic curves.

Haebara (1980), for example, developed his method using both old and new scales and presented two summation schemes. One summation scheme is that all ability estimates from old and new groups are used with equal weight. The other is that, in each group, ability estimates are grouped into, for example,  $G_O$  or  $G_N$  small intervals, so that their middle points and relative frequencies are used for the ability values and weights. The latter was presented as an approximation to the former to reduce the computation time and computer storage space needed. However, criterion functions following the Haebara approach often have been defined in such forms as  $Q_1$  in Equation 15.28, for which only the old scale  $\theta_O$  is used, in most cases, with ability points independent of sampled groups (see, e.g., Baker, 1993; Kim & Hanson, 2002; Ogasawara, 2001). Theoretically, only  $Q_2$  could be also used as a criterion function to estimate  $A$  and  $B$  of the new-to-old transformation. In fact, Haebara (1980) first focused on  $Q_2$  to develop his

method and added  $Q_1$ . The use of the sub-function  $Q_2$  makes sense in terms of the test equating practice with nonequivalent groups, where a synthetic population is often defined as the new group only (see Kolen & Brennan, 2004, for the formation of the synthetic population).

*Stocking-Lord Method.* This method is described similarly to the Haebara method except that item-category characteristic curves are replaced with test characteristic curves. Stocking and Lord (1983) defined their criterion function using only the targeted scale (i.e., the old scale in the case of the new-to-old transformation), although they emphasized the symmetric property of scale linking in their paper. This chapter presents a general “symmetric” version of the Stocking-Lord method, which chooses  $A$  and  $B$  to minimize the following criterion function,  $F^*$ ,

$$F^*(A, B) = F_1^*(A, B) + F_2^*(A, B), \quad (15.31)$$

where

$$F_1^*(A, B) = \int_{-\infty}^{+\infty} [\hat{T}_O(\theta_O) - \hat{T}_N^*(\theta_O)]^2 \psi_1(\theta_O) d\theta_O$$

and

$$F_2^*(A, B) = \int_{-\infty}^{+\infty} [\hat{T}_N(\theta_N) - \hat{T}_O^\#(\theta_N)]^2 \psi_2(\theta_N) d\theta_N.$$

As in Equation 15.27 for the Haebara method,  $A$  and  $B$  are estimated in such a way that the linking error on both scales is taken into account simultaneously. Note that the criterion function  $F^*$  assumes that proper scoring functions are assigned to all of the  $n$  items involved. This implies that when the NR model is used as one of the item response models for a mixed-format test, the items analyzed by the NR model should be scored in ordered categories so that proper scoring functions may be used for the categories.

A practical approximation of  $F^*$  is

$$F^*(A, B) \approx F(A, B) = F_1(A, B) + F_2(A, B), \quad (15.32)$$

where

$$F_1(A, B) = \frac{1}{S'_1} [\hat{T}_O(\theta_{iO}) - \hat{T}_N^*(\theta_{iO})]^2 W_1(\theta_{iO})$$

and

$$F_2(A, B) = \frac{1}{S'_2} [\hat{T}_N(\theta_{iN}) - \hat{T}_O^\#(\theta_{iN})]^2 W_2(\theta_{iN}).$$

where  $S'_1 = \sum_{i=1}^{G_O} W_1(\theta_{iO})$  and  $S'_2 = \sum_{i=1}^{G_N} W_2(\theta_{iN})$ . Note that two separate standardization factors,  $S'_1$  and  $S'_2$ , are used, as in Equation 15.28. The criterion function  $F$  is non-linear with respect to  $A$  and  $B$ , and thus the multivariate search technique described for the Haebara method can be

used to solve for the  $A$  and  $B$  that minimize  $F$ . The technique involves the evaluation of partial derivatives of  $F$  with respect to  $A$  and  $B$ .

As with the Haebara method, the Stocking-Lord method can be implemented using various special cases of the general form in Equation 15.32. Stocking and Lord (1983), for example, defined the criterion function using only  $F_1$ , suggesting as a practical summation scheme that the values of  $\theta_{iO}$  should be chosen to be a spaced sample of about 200 ability estimates from the old group. With the non-symmetric scheme using only  $F_1$ , Baker and Al-Karni (1991) and Baker (1992) suggested that an arbitrary set of  $G_O$  ability points along the old scale, independently of estimated abilities from the old group, could be used to simplify the summation over examinees. The arbitrary set of ability points, for example, can be a set of equally spaced values of ability along the old scale.

As just described, the use of  $F_1$  alone has been the usual scale linking practice for the Stocking-Lord method. Doing so can be justified since, when the old scale is designated as the base scale, practical concerns focus on estimating an appropriate new-to-old transformation that is unidirectional. Because one desirable property of test equating is symmetry (Lord, 1980), however, it would be desirable for the symmetry property to be considered in linking scales through linear transformation.

## 15.3 Functions

The IRT scale transformation functions employ three new structures that are in `IRTst.h`:

- `ItemSpec` which contains item parameter estimates for the old or new form;
- `CommonItemSpec` which contains pairs of ID's designating the common items; and
- `IRTstControl` which contains general information (e.g., numbers of score points, minimum scores, maximum scores, etc.) as well as ability distributions.

`Wrapper_IRTst()` is the wrapper function for IRT scale-score transformation. It is discussed first. Then, the functions called (directly or indirectly) by `Wrapper_IRTst()` are discussed.

### 15.3.1 `Wrapper_IRTst()`

As distinct from other wrapper functions in *Equating Recipes*, `Wrapper_IRTst()` does its own printing. The function prototype for

`Wrapper_IRTst()` is:

```
void Wrapper_IRTst(FILE *outf, char tt[],
    char ItemNewFile[], char ItemOldFile[], char ItemCommonFile[],
    char DistNewFile[], char DistOldFile[],
    int HA, enum symmetry HAsym, enum OnOff HAfs, double HAs, double HAI,
    int SL, enum symmetry SLsym, enum OnOff SLfs, double SLs, double SLi,
    char ST[], int PrintFiles)
```

The input variables are:

- ▷ `outf` = pointer to output file;
- ▷ `tt` = user-supplied title;
- ▷ `ItemNewFile[]` = name of file containing item parameters for new form X (see comments for `ItemInfoRead()` function later in this chapter);
- ▷ `ItemOldFile[]` = name of file containing item parameters for old form Y (see comments for `ItemInfoRead()` function later in this chapter);
- ▷ `ItemCommonFile[]` = name of file containing pairs of ID's for common items (see comments for `CommonItemSpec()` function later in this chapter);
- ▷ `DistNewFile[]` = name of file containing the ability distribution for new form X (see comments for `ThetaInfoRead()` function later in this chapter);
- ▷ `DistOldFile[]` = name of file containing the ability distribution for old form Y (see comments for `ThetaInfoRead()` function later in this chapter);
- ▷ `HA` = Haebara results (1 means compute results; 0 means no results);
- ▷ `HAsym` = Haebara symmetric option (`old_scale`, `new_scale`, `symmetric`);
- ▷ `HAfs` = Haebara function standardization option (`on` or `off`);
- ▷ `HAs` = Haebara starting value for the slope;
- ▷ `HAI` = Haebara starting value of the intercept;
- ▷ `SL` = Stocking-Lord results (1 means compute results; 0 means no results);

- ▷ **SLsym** = Stocking-Lord symmetric option (**old\_scale**, **new\_scale**, **symmetric**);
- ▷ **SLfs** = Stocking-Lord function standardization option (**on** or **off**);
- ▷ **SLs** = Stocking-Lord starting value for the slope;
- ▷ **SLi** = Stocking-Lord starting value for the intercept; and
- ▷ **ST** = method to be used for scale transformation ("MS" signifies mean/sigma; "MM" signifies mean/mean; "HA" signifies Haebara; "SL" signifies Stocking-Lord; "NL" signifies no scale transformation); and
- ▷ **PrintFiles**: 0 mean don't print input item parameter and quadrature files; 1 means print input item parameter and quadrature files.

Note that: (a) if either **DistNewFile[]** or **DistOldFile[]** is set to **NULL**, then only mean/mean and mean/sigma results are computed; (b) **Wrapper\_IRTst()** has no built in bootstrapping capability; and (c) scale transformation results are not saved, except in the sense discussed next.

When **ST** is not set to "NL", then two files are saved. The first file has the name **ItemNewFile[]** with ".ST" (without the quotes) appended indicating that it contains the new item parameter estimates after scale transformation using the **ST** method. The second file has the name **DistNewFile[]** with ".ST" (without the quotes) appended indicating that it contains the estimated ability distribution for the new group after scale transformation using the **ST** method.

Within **Wrapper\_IRTst()** three types of structures are declared: **ItemSpec** (two instances), **CommonItemSpec** (one instance), and **IRTstControl** (one instance). Memory for these structures is deallocated before **Wrapper\_IRTst()** terminates, which means that the only results that are saved are the files generated by **ST**. See the discussion of the examples in Sections 15.4.1 and 15.4.2 for further clarification.

### 15.3.2 Functions for Set-up and Clean-up

Before calling the functions that compute scaling constants for each of the four scale transformation methods, a preliminary process is required to set up the computing environment. The set-up process includes: (1) using dynamic memory allocation (DMA) to initialize pointers to variables of structure types (which are uniquely defined for scale linking), (2) reading the input for items for the new and old forms, (3) specifying the common items to be used, and (4) reading the input for ability distributions for the new and old examinee groups. For the set-up, three "stand-alone" functions are used: (1) **ItemInfoRead()**, (2) **ItemPairsRead()**, and (3) **ThetaInfoRead()**. The function **ItemInfoRead()** must be called before the

function `ItemPairsRead()` is executed, while the function `ThetaInfoRead()` may be used independently of the first two functions. After using the functions for the scale transformation methods, the set-up environment should be cleaned up so that the memory pointed to by the pointers to structures is returned to the heap (i.e., the region of free memory). The functions for such a clean-up task include (1) `StItemDeAlloc()`, (2) `StComItemDeAlloc()`, and (3) `StContDeAlloc()`. The three clean-up functions are stand-alone, and the `StContDeAlloc()` function must be used last. Provided below are detailed descriptions of the declaration and functionality for each of the set-up and clean-up functions.

```
struct ItemSpec *ItemInfoRead(FILE *inf, const char *oldOrnew,
    struct IRTstControl *Handle);
```

```
/*-----
```

**Functionality:**

Read pieces of information for items on the new or old form from an input file, depending on the `oldOrnew` string, which must be "new" or "old." Assume that the input file is opened but is not read in yet. To read the information, a pointer to the `ItemSpec` structure is first created using the function `malloc`, and then some of the structure members, if they are pointers, are initialized with designated memory addresses using the function `malloc`. In addition, some members of an `IRTstControl` structure referenced by the pointer `Handle` are initialized to their respective values that have been read.

**Input:**

<code>inf</code>	A file pointer; the file must be opened for input.
<code>oldOrnew</code>	A string indicating that the items to be read are on the old or new form. The string must be either "new" or "old."
<code>Handle</code>	A pointer to a variable of the <code>IRTstControl</code> structure, which is defined in the header file <code>IRTst.h</code> .

The input file must be prepared according to the following format:

Line 1	Number of items on either the new or old form.
Line 2+	A record for each item is typed. The record can range over multiple lines. So, the <code>fscanf</code> function is used, instead of the <code>fgets</code> function. The general format of the record is

```
ID Model CatNum ScoreFunc ScaleConst ItemParsList,
```

where

ID	Item ID
Model	L3 (three-parameter logistic model)
(uppercase)	GR (logistic graded response model)
	PC (generalized partial credit model)
	NR (nominal response model)
CatNum	Number of categories for the item
ScoreFunc	A list of scores given to CatNum categories
ScaleConst	Scaling Constant

In the case of the NR model, it should  
 equal 1; if other values are typed for the  
 NR model, they are ignored.  
 ItemParsList A list of item parameters for the item

Output:

Return a pointer to the ItemSpec structure.

-----\*/

```
struct CommonItemSpec *ItemPairsRead(FILE *inf, struct ItemSpec *NewItem,
struct ItemSpec *OldItem, struct IRTstControl *Handle);
```

/\*-----

Functionality:

Read pairs of new and old items, which are common items, from  
 an input file. Assume that the input file is opened but is not read in  
 yet. To read the information, a pointer to the ItemSpec structure is  
 first created using the function malloc. While reading the pairs  
 information, an array that is referenced by the pointer is set up with  
 the two ItemSpec "arrays," NewItem and OldItem. The common information  
 between the new and old items, such as ScaleConst, CatNum, and model,  
 is set with the new item information. In addition, the ComItemNum  
 member of an IRTstControl structure referenced by the pointer Handle is  
 initialized to the number of common items.

Input:

inf A file pointer; the file must be opened for input.  
 NewItem A pointer to the ItemSpec structure; the pointer points  
 to the memory address of an array of the ItemSpec  
 structure for items on the new form. The ItemSpec  
 structure is defined in the header file IRTst.h.  
 OldItem A pointer to the ItemSpec structure, which is used for  
 items on the old form.  
 Handle A pointer to a variable of the IRTstControl structure,  
 which is defined in IRTst.h.

The input file must be prepared according to the following format:

Line 1 Number of the common items, which are used to estimate  
 scaling constants, A and B.

Line 2+ Pairs of IDs for the new and old items  
 Each pair consists of two integers, a new item ID and  
 a matching old item ID.

Suppose, for example, that there are two common items  
 and the first and second items on the new form match  
 with the third and fourth items on the old form.

Pairs of items, then, must be designated as follows,  
 with the first line being for the number of common  
 items:

```
2
1 3
2 4
```

Output:



Return a pointer to the CommonItemSpec structure.

-----\*/

```
void ThetaInfoRead(FILE *inf, const char *oldOrnew,
    struct IRTstControl *Handle);
```

/\*-----

Functionality:

Read pieces of information for ability distributions of the new and old groups from an input file, depending on the oldOrnew string, which must be "new" or "old." Assume that the input file is opened but is not read in yet. While reading the pairs information, the member pointers of an IRTstControl structure referenced by the pointer Handle are set up by DMA. In addition, some members of the IRTstControl structure are initialized to their respective values that have been read.

Input:

inf	A file pointer; the file must be opened for input
oldOrnew	A string indicating that the ability distribution is for the old or new group. The string must be either "new" or "old."
Handle	A pointer to a variable of the IRTstControl structure, which is defined in the header file IRTst.h.

The input file must be prepared according to the following format:

Line 1	Number of the ability points for the new or old group.
Line 2+	Pairs of ability points and weights.

Each pair consists of an ability point and its weight. Suppose, for example, that there are five ability points to be used, either on the new or old scale. Pairs of points and weights for ability, then, must be typed as follows, with the first line being for the number of the pairs:

```
5
-2.0 0.05
-1.0 0.25
0.0 0.40
1.0 0.25
2.0 0.05
```

-----\*/

```
void StItemDeAlloc(struct ItemSpec *Item, const char *oldOrnew,
    struct IRTstControl *Handle);
```

/\*-----

Functionality:

Deallocate memory given to the pointer (Item) to an ItemSpec structure, depending on the form ("new" or "old"). The memory given to its members is also deallocated.

Input:

Item	A pointer to an array of the ItemSpec structure
oldOrnew	A string indicating that the array is about either the

```

                                old or new form.
                                Handle      A pointer to a variable of the IRTstControl structure
-----*/

void StComItemDeAlloc(struct CommonItemSpec *ComItem,
                     struct IRTstControl *Handle);
/*-----
   Functionality:
   Deallocate memory given to the pointer (ComItem) to a CommonItemSpec
   structure. The memory given to its members is also deallocated.

   Input:
   ComItem  A pointer to an array of the CommonItemSpec structure
   Handle   A pointer to a variable of the IRTstControl structure
-----*/

void StContDeAlloc(struct IRTstControl *Handle);
/*-----
   Functionality:
   Deallocate memory given to the members of the pointer Handle, which
   points to an IRTstControl structure.

   Input:
   Handle   A pointer to a variable of the IRTstControl structure
-----*/

```

### 15.3.3 Functions for the Moment Methods

The C code that computes scaling constants for the moment methods consists of the two “stand-alone” functions: `StMeanSigma()` and `StMeanMean()`. The function `StMeanSigma()` is used for the mean/sigma method and the function `StMeanMean()` is used for the mean/mean method. Detailed descriptions of the two functions are provided below.

```

void StMeanSigma(struct IRTstControl *Handle,
                 struct CommonItemSpec *ComItem, double *slope, double *intercept);
/*-----
   Functionality:
   Estimate the A (slope) and B (intercept) of the new-to-old
   transformation by using the mean/sigma method.

   Input:
   Handle      A pointer to a variable of the IRTstControl structure
   ComItem     A pointer to an array of the CommonItemSpec structure

   Output:
   *slope      The mean/sigma estimate of A
   *intercept  The mean/sigma estimate of B
-----*/

```

```

void StMeanMean(struct IRTstControl *Handle,
               struct CommonItemSpec *ComItem, double *slope, double *intercept);
/*-----
  Functionality:
  Estimate the A (slope) and B (intercept) of the new-to-old
  transformation by using the mean/mean method.

  Input:
      Handle      A pointer to a variable of the IRTstControl structure
      ComItem     A pointer to an array of the CommonItemSpec structure

  Output:
      *slope      The mean/mean estimate of A
      *intercept  The mean/mean estimate of B
-----*/

```

#### 15.3.4 Functions for the Characteristic Curve Methods

As opposed to the C code for the moment methods, the C code for the characteristic curve methods consists of two driver functions (**StHaebara()** and **StStockingLord()**) and many implementation functions for them. The calling structure for each driver function is as follows:

**StHaebara:** This function computes the Heabara estimates of  $A$  (slope) and  $B$  (intercept). For the two-dimension multivariate minimization algorithm, the function **StHaebara** calls the function **er\_dfpmin**. The function **StHaebara** also calls the two functions, **FuncHaebara** and **GradHaebara**, to calculate the value of the Haebara criterion function and the partial derivatives of the criterion function with respect to each of  $A$  and  $B$ . A more detailed function call structure is given below:

```

/*-----
StHaebara      calls      er_dfpmin
                               FuncHaebara
                               GradHaebara

er_dfpmin      calls      er_lnsrch

FuncHaebara    calls      ProbNew
                               ProbOld

GradHaebara    calls      ProbNew
                               ProbOld
                               PdNewOverS
                               PdOldOverS
                               PdNewOverI
                               PdOldOverI

ProbNew
ProbOld        calls      Prob3PL

```

```

                                ProblGR
                                ProbGPC
                                ProbnRM

PdNewOverS      calls  Pd3PLNewOverS
                                PdLGRNewOverS
                                PdGPCNewOverS
                                PdNRMNewOverS

PdOldOverS      calls  Pd3PLOldOverS
                                PdLGROldOverS
                                PdGPCOldOverS
                                PdNRMOldOverS

PdNewOverI      calls  Pd3PLNewOverI
                                PdLGRNewOverI
                                PdGPCNewOverI
                                PdNRMNewOverI

PdOldOverI      calls  Pd3PLOldOverI
                                PdLGROldOverI
                                PdGPCOldOverI
                                PdNRMOldOverI
-----*/

```

The declaration and descriptions of the function StHaebara are listed below:

```

void StHaebara(struct IRTstControl *Handle, struct CommonItemSpec *ComItem,
    enum symmetry SYM, enum OnOff FuncStd, double S0, double IO,
    double *slope, double *intercept);
/*-----
    Functionality:
    Estimate the A (slope) and B (intercept) of the new-to-old
    transformation by using the Haebara method.

    Input:
        Handle      A pointer to a variable of the IRTstControl structure
        ComItem     A pointer to an array of the CommonItemSpec structure
        SYM         Symmetric option (old_scale, new_scale, or symmetric)
        FuncStd     Function standardization option (on or off)
        S0          A starting value for the slope A
        IO          A starting value for the slope B

    Output:
        *slope      The Haebara estimate of A
        *intercept  The Haebara estimate of B
-----*/

```

StStockingLord: This function computes the Stocking-Lord estimates of  $A$  (slope) and  $B$  (intercept). As with the function

StHaebara, the function `er_dfpmin` is called for the minimization solution. The function `StStockingLord` also calls the two functions, `FuncStockingLord` and `GradStockingLord`, to calculate the value of the Stocking-Lord criterion function and the partial derivatives of the criterion function with respect to each of  $A$  and  $B$ . A more detailed function call structure is given below:

```

/*-----
StStockingLord      calls      er_dfpmin
                                FuncStockingLord
                                GradStockingLord

er_dfpmin           calls      er_lnsrch

FuncStockingLord    calls      ProbNew
                                ProbOld

GradStockingLord    calls      ProbNew
                                ProbOld
                                PdNewOverS
                                PdOldOverS
                                PdNewOverI
                                PdOldOverI

Note: The above six fuctions that are called
      by the function GradStockingLord call
      the same functions as in the function
      call structure for the Heabara method.
-----*/

```

The declaration and descriptions of the function `StStockingLord` are listed below:

```

void StStockingLord(struct IRTstControl *Handle,
    struct CommonItemSpec *ComItem, enum symmetry SYM, enum OnOff FuncStd,
    double S0, double I0, double *slope, double *intercept);
/*-----
    Functionality:
    Estimate the A (slope) and B (intercept) of the new-to-old
    transformation by using the Stocking-Lord method.

    Input:
        Handle      A pointer to a variable of the IRTstControl structure
        ComItem     A pointer to an array of the CommonItemSpec structure
        SYM         Symmetric option (old_scale, new_scale, or symmetric)
        FuncStd     Function standardization option (on or off)
        S0          A starting value for the slope A
        I0          A starting value for the slope B

    Output:
        *slope      The Stocking-Lord estimate of A

```

```
*intercept The Stocking-Lord estimate of B
```

```
-----*/
```

### 15.3.5 A Utility Function: ScaleTransform

The final linking task is to use the scaling constants (of the new-to-old transformation) to linearly transform both the new form item parameter estimates and the ability estimates for the new group's distribution onto the old scale. To do so, Equation 15.9 is used for the ability estimates, and Equations 15.11–15.14 are used for the item parameter estimates. Performing the transformations is straightforward, but can be tedious. The function `ScaleTransform()` can be used for this purpose. This function allows users to save the two sets of transformed output (one for the new item parameters and the other for the new group's distribution) into separate files. A detailed description of the function `ScaleTransform()` is provided below:

```
void ScaleTransform(const char *ItemOutF, const char *DistOutF,
    double slope, double intercept, struct ItemSpec *NewItem,
    struct IRTstControl *Handle);
```

/\*-----\*/

Functionality:  
Use the values of the slope and intercept of the new-to-old transformation to convert both the parameter estimates of the new form items and the ability points for the new group's distribution.

Input:

ItemOutF	The name of a file in which the transformed output for items on the new form is saved; The format of output is, in essence, the same as that of input, so the output file can be read by the function <code>ItemInfoRead</code> without any syntax error.
DistOutF	The name of a file in which the transformed ability points for the new group distribution are saved along with the original weights; As with <code>ItemOutF</code> , the output is saved using the format of input for the ability distribution, so the file <code>DistOutF</code> can be read by the function <code>ThetaInfoRead</code> without problems.
slope	The value of A (slope) for a chosen scale transformation method.
intercept	The value of B (intercept) for a chosen scale transformation method.
NewItem	A pointer to an array of the <code>ItemSpec</code> structure, which is for the new form.
Handle	A pointer to a variable of the <code>IRTstControl</code> structure.

Output:

ItemOutF	The transformed output file for the new form
DistOutF	The transformed output file for the new group's distribution

\* Note: The original input remains intact for both the items and  
distribution.

-----\*/

## 15.4 Examples

Provided next are two examples. The first involves mixed-format test forms; the second involves multiple-choice test forms. The mixed-format example is more complicated and described in more detail.

### 15.4.1 *Mixed Format*

The `main()` function in Table 15.1 illustrates using `Wrapper_IRTst()` for hypothetical test data for two mixed-format test forms that were administered independently to two examinee groups.<sup>3</sup> Both forms consist of ten multiple-choice (MC) items and five constructed-response (CR) items. The MC item responses are modeled using the 3PL model, while the CR items are modeled using the GR model. Three examples are provided. To link the new and old scales, the first example uses the ten MC items, the second example uses the five CR items, and the third example uses all 15 items. For each example, the four scale transformation methods are run. The third example also shows how to call the `ScaleTransform()` function using the Stocking-Lord solutions.

---

<sup>3</sup>The comments following each of the parameters in `Wrapper_IRTst()` are not required.

TABLE 15.1. Main() Code to Illustrate IRT Scale Transformation for Three Mixed-Format Dummy Data Examples

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    FILE *outf;

    outf = fopen("Chap 15 out (dummy)", "w");

    /* IRT scale transformation
       Three Mixed Format Dummy Data Examples */

    Wrapper_IRTst(                                     /*** Example 1 ***/
        outf,
        "Mixed Format Dummy Data Example using dummyV1items", /* heading */
        "dummyXitems", /* name of file containing item parameters for X */
        "dummyYitems", /* name of file containing item parameters for V */
        "dummyV1items", /*name of file containing id's for for V */
        "dummyXdlist", /* name of file containing ability dist for X */
        "dummyYdlist", /* name of file containing ability dist for y */
        1, symmetric, on, 1, 0, /* Haebara parameters */
        1, symmetric, on, 1, 0, /* Stocking-Lord parameters */
        "NL", /* scale trans method ["MS", "MM", "HA", "SL", "NL" (none)] */
        0 /* do not print input files */
    );
    Wrapper_IRTst(                                     /*** Example 2 ***/
        outf,
        "Mixed Format Dummy Data Example using dummyV2items", /* heading */
        "dummyXitems", /* name of file containing item parameters for X */
        "dummyYitems", /* name of file containing item parameters for V */
        "dummyV2items", /*name of file containing id's for for V */
        "dummyXdlist", /* name of file containing ability dist for X */
        "dummyYdlist", /* name of file containing ability dist for y */
        1, symmetric, on, 1, 0, /* Haebara parameters */
        1, symmetric, on, 1, 0, /* Stocking-Lord parameters */
        "NL", /* scale trans method ["MS", "MM", "HA", "SL", "NL" (none)] */
        0 /* do not print input files */
    );
    Wrapper_IRTst(                                     /*** Example 3 ***/
        outf,
        "Mixed Format Dummy Data Example using dummyV3items", /* heading */
        "dummyXitems", /* name of file containing item parameters for X */
        "dummyYitems", /* name of file containing item parameters for V */
        "dummyV3items", /*name of file containing id's for for V */
        "dummyXdlist", /* name of file containing ability dist for X */
        "dummyYdlist", /* name of file containing ability dist for y */
        1, symmetric, on, 1, 0, /* Haebara parameters */
        1, symmetric, on, 1, 0, /* Stocking-Lord parameters */
        "SL", /* scale trans method ["MS", "MM", "HA", "SL", "NL" (none)] */
        0 /* do not print input files */
    );

    fclose(outf);
    return 0;
}
*****/

```



The input files need to be prepared in the manner indicated in the comments for the set-up functions. For the examples presented here, the input files for items on the new and old forms (`dummyXitems` and `dummyYitems`, respectively) are as follows:

```
[dummyXitems]
15
1  L3 2 0 1      1.7  1.0755 -1.8758  0.1240
2  L3 2 0 1      1.7  0.6428 -0.9211  0.1361
3  L3 2 0 1      1.7  0.6198 -1.3362  0.1276
4  L3 2 0 1      1.7  0.6835 -1.8967  0.1619
5  L3 2 0 1      1.7  0.9892 -0.6427  0.2050
6  L3 2 0 1      1.7  0.5784 -0.8181  0.1168
7  L3 2 0 1      1.7  0.9822 -0.9897  0.1053
8  L3 2 0 1      1.7  1.6026 -1.2382  0.1202
9  L3 2 0 1      1.7  0.8988 -0.5180  0.1320
10 L3 2 0 1      1.7  1.2525 -0.7164  0.1493
11 GR 4 0 1 2 3  1.7  1.1196 -2.1415  0.0382  0.6551
12 GR 4 0 1 2 3  1.7  1.2290 -1.7523 -1.0660  0.3533
13 GR 4 0 1 2 3  1.7  0.6405 -2.3126 -1.8816  0.7757
14 GR 4 0 1 2 3  1.7  1.1622 -1.9728 -0.2810  1.1387
15 GR 4 0 1 2 3  1.7  1.2249 -2.2207 -0.8252  0.9702

[dummyYitems]
15
1  L3 2 0 1      1.7  0.7444 -1.5617  0.1609
2  L3 2 0 1      1.7  0.5562 -0.1031  0.1753
3  L3 2 0 1      1.7  0.5262 -1.0676  0.1602
4  L3 2 0 1      1.7  0.6388 -1.3880  0.1676
5  L3 2 0 1      1.7  0.8793 -0.2051  0.1422
6  L3 2 0 1      1.7  0.4105  0.0555  0.2120
7  L3 2 0 1      1.7  0.7686 -0.3800  0.2090
8  L3 2 0 1      1.7  1.0539 -0.7570  0.1270
9  L3 2 0 1      1.7  0.7400  0.0667  0.1543
10 L3 2 0 1      1.7  0.7479  0.0281  0.1489
11 GR 4 0 1 2 3  1.7  0.9171 -1.7786  0.7177  1.45011
12 GR 4 0 1 2 3  1.7  0.9751 -1.4115 -0.4946  1.15969
13 GR 4 0 1 2 3  1.7  0.5890 -1.8478 -1.4078  1.51339
14 GR 4 0 1 2 3  1.7  0.9804 -1.6151  0.3002  2.04728
15 GR 4 0 1 2 3  1.7  1.0117 -1.9355 -0.2267  1.88991
```

The three input files for common-item sets (`dummyV1items`, `dummyV2items`, and `dummyV3items`) are as follows:

[dummyV1items]	[dummyV2items]	[dummyV3items]
10	5	15
1 1	11 11	1 1
2 2	12 12	2 2
3 3	13 13	3 3
4 4	14 14	4 4
5 5	15 15	5 5
6 6		6 6
7 7		7 7
8 8		8 8
9 9		9 9
10 10		10 10
		11 11
		12 12
		13 13
		14 14
		15 15

Finally, the two input files for ability distributions for the new and old groups (`dummyXdlist` and `dummyYdlist`, respectively) are as follows:

[dummyXdlist]	[dummyYdlist]
10	10
-0.3970E+01 0.2627E-03	-0.4093E+01 0.1323E-03
-0.3095E+01 0.4983E-02	-0.3185E+01 0.3191E-02
-0.2220E+01 0.3490E-01	-0.2277E+01 0.3153E-01
-0.1346E+01 0.1430E+00	-0.1370E+01 0.1463E+00
-0.4708E+00 0.2939E+00	-0.4619E+00 0.3138E+00
0.4040E+00 0.3215E+00	0.4459E+00 0.3182E+00
0.1279E+01 0.1630E+00	0.1354E+01 0.1534E+00
0.2154E+01 0.3470E-01	0.2261E+01 0.3096E-01
0.3028E+01 0.3656E-02	0.3169E+01 0.2480E-02
0.3903E+01 0.1511E-03	0.4077E+01 0.9835E-04

Table 15.2 presents the scaling constants ( $A$  and  $B$ ) that were obtained from applying the four scale transformation methods to each of the three examples of scale linking. The results indicate that the linear relation between the old and new scales can vary depending on which common-item sets are used for scale linking. It is noteworthy that across three examples, the performance of the Haebara method is very similar to that of the Stocking-Lord method. The content of the files produced by the `ScaleTransform()` function is provided in Table 15.3.

TABLE 15.2. Output Illustrating IRT Scale Transformation for Three Mixed-Format Dummy Data Examples

```

/*****/
Mixed Format Dummy Data Example using dummyV1items

IRT Scale Transformation

Input files:

    dummyXitems (item parameters for new form X)
    dummyYitems (item parameters for new form Y)
    dummyV1items (id's for common items)
    dummyXdlist (ability distribution for new form X)
    dummyYdlist (ability distribution for old form Y)

Haebara symmetric option = symmetric
Haebara function standardization option = on
Haebara starting value for slope (A) = 1.00000
Haebara starting value for intercept (B) = 0.00000

Stocking-Lord symmetric option = symmetric
Stocking-Lord function standardization option = on
Stocking-Lord starting value for slope (A) = 1.00000
Stocking-Lord starting value for intercept (B) = 0.00000

-----
                        A=slope      B=intercept
Mean/Sigma Method:      1.27487      0.86514
Mean/Mean Method:       1.31978      0.91432
Haebara Method:         1.31780      0.82780
Stocking-Lord Method:    1.31823      0.82329
-----

Mixed Format Dummy Data Example using dummyV2items

...

-----
                        A=slope      B=intercept
Mean/Sigma Method:      1.15641      0.70194
Mean/Mean Method:       1.20184      0.73380
Haebara Method:         1.17229      0.70426
Stocking-Lord Method:    1.17060      0.70458
-----

Mixed Format Dummy Data Example using dummyV3items

...

-----
                        A=slope      B=intercept
Mean/Sigma Method:      1.16357      0.72146
Mean/Mean Method:       1.27406      0.81638
Haebara Method:         1.20397      0.72588
Stocking-Lord Method:    1.21393      0.73509
-----

/*****/

```

TABLE 15.3. Contents of Files Created for Mixed-Format Dummy Data Example

```

/*****
Transformed Item Paramter Estimates for New Form
Based on Stocking-Lord Method
Results stored in file named: dummyXitems.ST

  1  L3 2 0 1 1.7      0.88597 -1.54199  0.12400
  2  L3 2 0 1 1.7      0.52952 -0.38306  0.13610
  3  L3 2 0 1 1.7      0.51058 -0.88696  0.12760
  4  L3 2 0 1 1.7      0.56305 -1.56736  0.16190
  5  L3 2 0 1 1.7      0.81488 -0.04510  0.20500
  6  L3 2 0 1 1.7      0.47647 -0.25802  0.11680
  7  L3 2 0 1 1.7      0.80911 -0.46633  0.10530
  8  L3 2 0 1 1.7      1.32018 -0.76799  0.12020
  9  L3 2 0 1 1.7      0.74041  0.10628  0.13200
 10  L3 2 0 1 1.7      1.03178 -0.13457  0.14930
 11  GR 4 0 1 2 3 1.7    0.92230 -1.86453  0.78146  1.53033
 12  GR 4 0 1 2 3 1.7    1.01242 -1.39207 -0.55895  1.16397
 13  GR 4 0 1 2 3 1.7    0.52763 -2.07223 -1.54903  1.67673
 14  GR 4 0 1 2 3 1.7    0.95739 -1.65974  0.39398  2.11739
 15  GR 4 0 1 2 3 1.7    1.00904 -1.96067 -0.26664  1.91284
-----

Transformed Ability distribution for New Group
Based on Stocking-Lord Method
Results stored in file named: dummyXdistrib.ST

-4.084193E+000  2.627000E-004
-3.022009E+000  4.983000E-003
-1.959824E+000  3.490000E-002
-8.988534E-001  1.430000E-001
 1.635739E-001  2.939000E-001
 1.225516E+000  3.215000E-001
 2.287700E+000  1.630000E-001
 3.349885E+000  3.470000E-002
 4.410855E+000  3.656000E-003
 5.473040E+000  1.511000E-004
*****/

```

### 15.4.2 3PL

Kolen and Brennan (2004, sect. 6.8) provide a 3PL example of IRT scale transformation. The Form X and Form Y item parameter estimates are in Kolen and Brennan (2004, Table 6.5). The common items are 3, 6, 9, 12, 15, 18, 21, 24, 30, 33, and 36 in both forms. The ability distributions are in Kolen and Brennan (2004, Table 6.10). The new form item parameter estimates are in a file named `KB6Xitmes`, the old form item parameter estimates are in `KB6Yitmes`, and the pairs of common-item ID's are in `KB6Vitmes`. In `KB6Xitmes`, the record for item 1 is:

```
1   L3   2   0   1   1.7       0.5496  -1.7960  0.1751
```

The records for all other items (both new and old forms) follow the same conventions.

Table 15.4 provides a `main()` function to estimate the slope and intercept for all four transformation methods.<sup>4</sup> Note that the Haebara method uses `symmetric` as the symmetric option, which is consistent with Haebara's (1980) suggestion, while the Stocking-Lord method uses `old_scale` as the symmetric option, which is consistent with the approach of Stocking and Lord (1983). The transformation-constants output is provided in Table 15.5. Based on the mean/sigma method, Table 15.6 provides the transformed item parameter estimates for the new form and the transformed ability distribution for the new group.

---

<sup>4</sup>The comments following each of the parameters in `Wrapper_IRTst()` are not required.

TABLE 15.4. Main() Code to Illustrate IRT Scale Transformation for Kolen and Brennan (2004, chap. 6) Example

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    FILE *outf;

    outf = fopen("Chap 15 out (K&B Chap 6 Example)", "w");

    /* IRT scale transformation
       for Kolen and Brennan (2004, Chap 6) Example */

    Wrapper_IRTst(
        outf,
        "Kolen and Brennan (2004, Chapter 6) Example", /* heading */
        "KB6Xitems", /* name of file containing item parameters for X */
        "KB6Yitems", /* name of file containing item parameters for Y */
        "KB6Vitems", /*name of file containing id's for for V */
        "KB6Xdists", /* name of file containing ability dist for X */
        "KB6Ydist", /* name of file containing ability dist for y */
        1, /* Haebara results: 1 --> yes; 0--> no */
        symmetric, /* Haebara sym option (old_scale, new_scale, symmetric) */
        on, /* Haebara function standardization option (on or off) */
        1, /* Haebara starting value for the slope */
        0, /* Haebara starting value of the intercept */
        1, /* Stocking-Lord results: 1 --> yes; 0--> no */
        old_scale, /* Stocking-Lord sym (old_scale, new_scale, symmetric) */
        on, /* Stocking-Lord function standardization option (on or off) */
        1, /* Stocking-Lord starting value for the slope */
        0, /* Stocking-Lord starting value of the intercept */
        "MS", /* scale trans method ["MS", "MM", "HA", "SL", "NL" (none)] */
        0 /* do not print input files */
    );

    fclose(outf);
    return 0;
}
*****/

```

TABLE 15.5. Output Illustrating IRT Scale Transformation for Kolen and Brennan (2004, chap. 6) Example

```

/*****/
Kolen and Brennan (2004, Chapter 6) Example

IRT Scale Transformation

Input files:

    KB6Xitems (item parameters for new form X)
    KB6Yitems (item parameters for new form Y)
    KB6Vitems (id's for common items)
    KB6Xdist (ability distribution for new form X)
    KB6Ydist (ability distribution for old form Y)

Haebara symmetric option = symmetric
Haebara function standardization option = on
Haebara starting value for slope (A) = 1.00000
Haebara starting value for intercept (B) = 0.00000

Stocking-Lord symmetric option = old_scale
Stocking-Lord function standardization option = on
Stocking-Lord starting value for slope (A) = 1.00000
Stocking-Lord starting value for intercept (B) = 0.00000

-----
Mean/Sigma Method:      A=slope      B=intercept
                        1.17607      -0.50419
Mean/Mean Method:       1.14496      -0.47897
Haebara Method:         1.06383      -0.45399
Stocking-Lord Method:    1.08616      -0.47336
-----

/*****/

```

TABLE 15.6. Contents of Files Created for Kolen and Brennan (2004, chap. 6) Example

```

/*****/
Transformed Item Parameter Estimates for New Form
Based on Mean/Sigma Method
Results stored in file named: KB6Xitems.ST

  1  L3 2 0 1 1.7      0.46732 -2.61642  0.17510
  2  L3 2 0 1 1.7      0.67096 -1.06823  0.11650
  3  L3 2 0 1 1.7      0.38697 -1.33932  0.20870
  4  L3 2 0 1 1.7      1.22807  0.06421  0.28260
  5  L3 2 0 1 1.7      0.82818 -0.70177  0.26250
  6  L3 2 0 1 1.7      0.49648 -1.51173  0.20380
  7  L3 2 0 1 1.7      0.73159  0.03046  0.32240
  8  L3 2 0 1 1.7      0.97315 -0.65720  0.22090
  9  L3 2 0 1 1.7      0.64146 -0.47926  0.16000
 10  L3 2 0 1 1.7      0.77971  0.68823  0.36480
 11  L3 2 0 1 1.7      0.81559  0.34470  0.23990
 12  L3 2 0 1 1.7      0.56399 -0.44468  0.12400
 13  L3 2 0 1 1.7      1.04789 -0.01412  0.25350
 14  L3 2 0 1 1.7      0.89212  0.42279  0.15690
 15  L3 2 0 1 1.7      0.90896  0.62602  0.29860
 16  L3 2 0 1 1.7      0.78167  0.21310  0.25210
 17  L3 2 0 1 1.7      0.75973  0.09890  0.22730
 18  L3 2 0 1 1.7      0.82240 -0.27485  0.05350
 19  L3 2 0 1 1.7      0.55796 -0.03929  0.12010
 20  L3 2 0 1 1.7      0.89756  0.61085  0.20360
 21  L3 2 0 1 1.7      0.29581  2.17350  0.14890
 22  L3 2 0 1 1.7      0.71696  0.74257  0.23320
 23  L3 2 0 1 1.7      0.94739  0.18099  0.06440
 24  L3 2 0 1 1.7      1.23963  0.70023  0.24530
 25  L3 2 0 1 1.7      0.43679  1.11762  0.14270
 26  L3 2 0 1 1.7      0.78175  0.76386  0.08790
 27  L3 2 0 1 1.7      1.59947  1.14961  0.19920
 28  L3 2 0 1 1.7      1.27926  1.27086  0.16420
 29  L3 2 0 1 1.7      0.82172  1.31202  0.14310
 30  L3 2 0 1 1.7      0.59690  2.13034  0.08530
 31  L3 2 0 1 1.7      1.07570  1.70201  0.24430
 32  L3 2 0 1 1.7      0.72844  1.51160  0.08650
 33  L3 2 0 1 1.7      1.19720  1.32531  0.07890
 34  L3 2 0 1 1.7      0.49385  3.58008  0.13990
 35  L3 2 0 1 1.7      0.78711  3.16540  0.10900
 36  L3 2 0 1 1.7      1.10478  2.03484  0.10750
-----

Transformed Ability distribution for New Group
Based on Mean/Sigma Method
Results stored in file named: KB6Xdists.ST

-5.208486E+000  1.010000E-004
-4.162956E+000  2.760000E-003
-3.117426E+000  3.021000E-002
-2.071895E+000  1.420000E-001
-1.026365E+000  3.149000E-001
 1.798905E-002  3.158000E-001
 1.063519E+000  1.542000E-001
 2.109050E+000  3.596000E-002
 3.154580E+000  3.925000E-003
 4.200110E+000  1.860000E-004
/*****/

```



# 16

## Test Equating Using Item Response Theory

Once the IRT item and ability parameters on two test forms are placed on the same scale, it is possible to obtain equivalents on one form relative to the other form. IRT provides two approaches to finding such equivalents: one is IRT true score equating, and the other is IRT observed score equating. This chapter describes these two IRT equating methods more generally than in Kolen and Brennan (2004), so that these methods can be implemented for mixed-format tests as well as single-format tests. Nearly the same notational conventions are used in this chapter as in Chapter 15.

### 16.1 Basic Concepts

Assume that there are two forms of a mixed-format test, Form X and Form Y, that are associated with the new and old ability scales,  $\theta_N$  and  $\theta_O$ , respectively. The old scale associated with Form Y is designated as the common scale, and Form X is a new form to be equated to Form Y. A random variable  $X$  represents test scores on Form X and has the probability and cumulative distributions,  $f(x)$  and  $F(x)$ , respectively. Similarly, a random variable  $Y$  for test scores on Form Y has the probability and cumulative distributions,  $g(y)$  and  $G(y)$ , respectively.

IRT true score equating is the process of finding the Form-Y true score equivalent of a given true score on Form X. Likewise, IRT observed score equating refers to the process of finding the Form-Y observed score equivalent of a given observed score on Form X. The true and observed scores

are on the raw score scale that is developed by summing item scores, as in classical test theory (CTT). However, IRT true score equating uses the true score dictated by the IRT models, while IRT observed score equating uses the distribution of observed scores expected under the IRT models.

### 16.1.1 Scale Transformation as Equating

Linking ability scales from two test forms is often regarded as “equating” in the IRT literature. Suppose that the linking function for the two scales is  $\theta_O = A\theta_N + B$ . From this perspective, the score scale of primary interest is the IRT ability scale, and the ability scores  $(\theta_N, \theta_O)$  that satisfy this linear function consist of the pairs of “equivalent” scores. That is, equating is none other than scale transformation. However, another perspective on equating in IRT treats such scale linking as a preliminary step for equating and, based on a common ability scale, further attempts to find equivalent “test scores” between two test forms. The test scores are true scores in IRT true score equating and observed scores in IRT observed score equating.<sup>1</sup>

### 16.1.2 Definition of True Scores in IRT

Let  $X_j$  and  $Y_j$  be the scoring variables for a response for item  $j$  on Form X and Form Y, respectively, of an  $n$ -item mixed-format test. According to CTT, the true score  $T_X$  on Form X is the expectation of the observed score  $X = \sum_{j=1}^n X_j$ . Similarly, the true and observed scores for Form Y are designated as  $T_Y$  and  $Y = \sum_{j=1}^n Y_j$ , respectively. In IRT, the expectation of the observed score  $X$  given ability  $\theta$  is equivalent to the test characteristic curve (Hambleton & Swaminathan, 1985; Kolen & Brennan, 2004; Lord, 1980). Thus, the true score  $T_X$  given ability  $\theta$  is defined by the test characteristic curve  $T_X(\theta)$ , as follows:

$$T_X(\theta) = E(X|\theta) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jkX} P_{jkX}(\theta), \quad (16.1)$$

where  $U_{jkX}$  and  $P_{jkX}$  are the score and probability associated with category  $k$  of item  $j$  with  $m_j$  categories on Form X. The true score for Form Y is similarly defined as

$$T_Y(\theta) = E(Y|\theta) = \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jkY} P_{jkY}(\theta). \quad (16.2)$$

Because the test characteristic curve is a monotonic increasing function of  $\theta$ , the lower and upper limits of any particular true score  $t_X$  on Form X

---

<sup>1</sup>It should not be assumed that IRT true scores are always necessarily the same as true scores in classical test theory or universe scores in generalizability theory.

are obtained as  $\theta$  approaches  $-\infty$  and  $+\infty$ , respectively. As  $\theta$  approaches  $-\infty$ , the first category has a probability value of 1 and the other categories have a value of 0 or  $c_{jX}$  (the lower asymptote for a 3PL item  $j$ ), whereas as  $\theta$  approaches  $+\infty$ , the last category has a probability value of 1 and the other categories have a value of 0. Thus,  $t_X$  has the following range:

$$\sum_{j=1}^n (U_{j1X} + \delta_j c_{jX}) < t_X < \sum_{j=1}^n U_{jm_j X}, \quad (16.3)$$

where if item  $j$  is a 3PL item,  $\delta_j = 1$ ; otherwise  $\delta_j = 0$ . Note that when all items are 3PL items with the two category scores  $U_{j1X} = 0$  and  $U_{j2X} = 1$ , the lower limit is simply  $\sum_{j=1}^n c_{jX}$ . Using the the same logic, the true score  $t_Y$  on Form Y has the following range:

$$\sum_{j=1}^n (U_{j1Y} + \delta_j c_{jY}) < t_Y < \sum_{j=1}^n U_{jm_j Y}. \quad (16.4)$$

### 16.1.3 Observed Score Distributions Generated by IRT

IRT models are expressions of the conditional probabilities for category scores given item and ability parameters. The observed test score is a function of category scores. Thus, IRT has a framework for generating the conditional distributions,  $f(x|\theta)$  or  $g(y|\theta)$ , of observed scores, which results in the marginal distributions,  $f(x)$  or  $g(y)$ , over the range of  $\theta$ . IRT observed score equating employs such theoretically derived marginal distributions of observed scores.

For mixed-format tests containing polytomous items, one can use a compound multinomial distribution to model the probability distribution of observed test scores at a given ability level  $\theta$  (Kolen & Brennan, 2004). Roughly speaking, “compound” refers to varying category characteristic curves among items on a test, whereas “multinomial” refers to multiple categories for item scores. The compound multinomial distribution reduces to the compound binomial distribution when the test contains only dichotomous items (Lord, 1980; Lord & Wingersky, 1984). Unfortunately, the probability generating function for a compound multinomial distribution has not been expressed in a tractable form so that the conditional probability distribution of test scores may be functionally derived. However, the conditional distribution can be calculated iteratively using the recursive algorithm described by Kolen and Brennan (2004, p. 219), as follows.

Define  $f_r(x|\theta)$  as the probability distribution of test score  $x$  (on Form X) over the first  $r$  items for examinees of ability  $\theta$ . Further, for item 1 having  $m_1$  categories, denote the category characteristic probabilities  $P_{11}(\theta)$ ,  $P_{12}(\theta)$ ,  $\dots$ ,  $P_{1m_1}(\theta)$ , respectively, as  $f_1(x = U_{11}|\theta)$ ,  $f_1(x = U_{12}|\theta)$ ,  $\dots$ ,  $f_1(x = U_{1m_1}|\theta)$ . Then for  $r > 1$ , the recursion formula for finding the probability of examinees with ability  $\theta$  earning score  $x$  after the  $r$ th added item

is

$$f_r(x|\theta) = \sum_{k=1}^{m_r} f_{r-1}(x - U_{rk}|\theta)P_{rk}(\theta), \quad \text{for } x \text{ between } \min_r \text{ and } \max_r, \quad (16.5)$$

where  $\min_r$  and  $\max_r$  are the minimum and maximum scores after adding the  $r$ th item. Note that, by definition,  $f_{r-1}(x - U_{rk}|\theta) = 0$  when  $x - U_{rk} < \min_{r-1}$  or  $x - U_{rk} > \max_{r-1}$ . This recursion algorithm basically does, at a given level  $\theta$ , the following: (1) finds across items all possible combinations of item categories that lead to a specific score  $x$ , (2) calculates the product of values of category characteristic curves for each combination, and then (3) sums all the products across combinations.

Given the conditional probability distributions, the marginal probability distribution can be calculated by integrating the conditional distribution over the continuous ability distribution. A practical approach to this integration is to sum the conditional distributions with relative weights over a set of discrete  $\theta$  values, such that

$$f(x) = \sum_{i=1}^{N_X} f(x|\theta_{iX})W(\theta_{iX}), \quad (16.6)$$

where  $\theta_{iX}$  ( $i = 1, 2, \dots, N_X$ ) and  $W(\theta_{iX})$  are the  $N_X$  quadrature points and weights, respectively, for the ability distribution. Usually, it is the case that the quadrature weights are standardized such that  $\sum_i W(\theta_{iX}) = 1$ .

Using the algorithm expressed by Equation 16.5, one can calculate the conditional distribution  $g(y|\theta)$  for Form Y. With the quadrature points and weights for the Form-Y ability distribution,  $\theta_{iY}$  ( $i = 1, 2, \dots, N_Y$ ) and  $W(\theta_{iY})$ , the marginal distribution can be obtained using

$$g(y) = \sum_{i=1}^{N_Y} g(y|\theta_{iY})W(\theta_{iY}). \quad (16.7)$$

## 16.2 Equating Methods

The IRT equating methods use unique definitions for equivalent scores between two forms. IRT true score equating states that the true score on Form X associated with a given value of  $\theta$  on the common ability scale is equivalent to the true score on Form Y associated with that  $\theta$ .

By contrast, based on the marginal distributions of observed test scores, IRT observed score equating defines the Form-Y equivalent of a given score on Form X as the Form-Y score that has the same percentile rank as the Form-X score. The following sections describe how each definition leads to its equating function.

### 16.2.1 IRT True Score Equating

Denote as  $e_Y[T_X(\theta)]$  the Form-Y true score equivalent of a Form-X true score associated with a given  $\theta$ . The definition of IRT true score equating is

$$e_Y[T_X(\theta)] = T_Y(\theta). \quad (16.8)$$

Further, let  $T_X^{-1}$  be the inverse function of  $T_X$ , which is a function of  $\theta$ . Then, the IRT true score equating function is

$$e_Y(t_X) = T_Y(t_X^{-1}), \quad (16.9)$$

where  $t_X^{-1} = \theta$ . Equation 16.9 implies that true score equating is a three-step process (Kolen & Brennan, 2004):

1. Specify a true score  $t_X$  on Form X.
2. Find the value of  $\theta$  that corresponds to that true score.
3. Use  $T_Y(\theta)$  to find the true score  $t_Y$  on Form Y that corresponds to that value of  $\theta$ .

The following points should be noted concerning this process. First, the specified true score must be within the range of  $t_X$  (see Equation 16.3) and typically is an integer. Because of the restricted range of  $t_X$ , Form-Y equivalents are not defined at the Form-X integer scores that are equal to the minimum and maximum test scores. Second, Step 2 requires the use of an iterative search technique, because of the non-linear relation between  $t_X$  and  $\theta$ . The Newton-Raphson method, which is described by Equation 15.30 in Chapter 15, can be used. To apply the Newton-Raphson method to Step 2, the function of  $\theta$  under consideration is

$$h(\theta) = t_X - \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jkX} P_{jkX}(\theta). \quad (16.10)$$

The first derivative of  $h(\theta)$  with respect to  $\theta$  is

$$h'(\theta) = - \sum_{j=1}^n \sum_{k=1}^{m_j} U_{jkX} \frac{\partial}{\partial \theta} P_{jkX}(\theta), \quad (16.11)$$

where  $\partial P_{jkX}(\theta)/\partial \theta$  is model specific. Third, item parameters are usually unknown, so, in practice, the three-step process uses estimates of them.

The true score equating relationship in Equation 16.9 often is used to convert observed test scores on Form X to observed test scores on Form Y, although no theoretical reason exists for treating observed scores as true scores (Kolen & Brennan, 2004). When this is done, a procedure is needed for converting Form-X test scores outside of the range of possible true scores on Form X. The following procedure, originally suggested by Kolen (1981), can be used:

1. Set the minimum test score ( $\min_X = \sum_{j=1}^n U_{j1X}$ ) on Form X equal to the minimum test score ( $\min_Y = \sum_{j=1}^n U_{j1Y}$ ) on Form Y.
2. Set the lower bound of Form-X true scores,  $\text{lb}_X = \sum_{j=1}^n (U_{j1X} + \delta_j c_{jX})$ , equal to the lower bound of Form-Y true scores,  $\text{lb}_Y = \sum_{j=1}^n (U_{j1Y} + \delta_j c_{jY})$ .
3. Use linear interpolation to find equivalents between the minimum scores and lower bounds.
4. Set the Form-X maximum test score ( $\max_X = \sum_{j=1}^n U_{jm_jX}$ ) equal to the Form-Y maximum test score ( $\max_Y = \sum_{j=1}^n U_{jm_jY}$ ).

In the third step, the linear interpolation is formalized such that

$$e_Y(t_X^*) = \frac{\text{lb}_Y - \min_Y}{\text{lb}_X - \min_X} (t_X^* - \min_X) + \min_Y, \quad (16.12)$$

where  $t_X^*$  indicates a score outside the range of possible true scores, but within the range of possible observed test scores. If  $\text{lb}_X = \min_X$  (i.e., 3PL items are not involved), then the second and third steps are ignored.

### 16.2.2 IRT Observed Score Equating

The IRT observed score equating method uses equipercentile procedures with the marginal observed score distributions,  $f(x)$  and  $g(y)$ , for Forms X and Y, respectively, to find Form-Y equivalents of Form-X observed scores. The equipercentile equating function identifies test scores on Form X that have the same percentile ranks as test scores on Form Y. Specifically, the equipercentile equating function is

$$e_Y(x) = Q^{-1}[P(x)], \quad (16.13)$$

where  $Q^{-1}$  is the inverse of the percentile rank function  $Q(y)$  based on  $g(y)$ , and  $P(x)$  is the percentile rank function based on  $f(x)$ .

Unlike IRT true score equating, IRT observed score equating requires an explicit specification of the distribution of ability in the population of examinees (Kolen & Brennan, 2004). This requirement is related to the fact that IRT equating methods do not necessarily assume random equivalence between examinee groups that took Forms X and Y. Thus, as in the common-item nonequivalent groups design, the marginal distributions,  $f(x)$  and  $g(y)$ , should be defined for a synthetic population.

Suppose that Forms X and Y are associated with Populations 1 and 2, respectively. Based on Equations 16.6 and 16.7, the four marginal distributions are (Kolen & Brennan, 2004):

1.  $f_1(x) = \sum_i f(x|\theta_{iX})W(\theta_{iX})$  is the Form-X distribution for Population 1.

2.  $f_2(x) = \sum_i f(x|\theta_{iY})W(\theta_{iY})$  is the Form-X distribution for Population 2.
3.  $g_1(y) = \sum_i g(y|\theta_{iX})W(\theta_{iX})$  is the Form-Y distribution for Population 1.
4.  $g_2(y) = \sum_i g(y|\theta_{iY})W(\theta_{iY})$  is the Form-Y distribution for Population 2.

Then, the Form-X and Form-Y marginal distributions for the synthetic population are

$$f_s(x) = w_1 f_1(x) + w_2 f_2(x) \quad (16.14)$$

and

$$g_s(y) = w_1 g_1(y) + w_2 g_2(y), \quad (16.15)$$

where the subscript  $s$  refers to the synthetic population with weights  $w_1$  and  $w_2$  for Populations 1 and 2, respectively. These weights must satisfy the conditions that  $w_1 + w_2 = 1$  and  $w_1, w_2 \geq 0$ .

In practice, the observed score distributions are calculated using the estimates of item parameters and ability distributions. Thus, the observed score equating results are subject to sampling error. Moreover, to simplify the formation of observed score distributions for a synthetic population, an investigator can set  $w_1 = 1$  and  $w_2 = 0$ , in which case the synthetic population is simply the Form-X population.<sup>2</sup>

## 16.3 Functions

The IRT equating functions in *Equating Recipes* employ three new structures that are in `IRTeq.h`:

- `RawFitDist` which contains a fitted distribution (new form or old form);
- `RawTruObsEquiv` which contains IRT true-score and observed-score equivalents; and
- `IRT_INPUT` which contains all IRT input and some output.

The structures `ItemSpec` and `IRTstControl` in `IRTst.h` are also employed, but `CommonItemSpec` is not used. (See page 236 for a description of these structures.) In *Equating Recipes* it is assumed that any required scale transformation has been performed before IRT equating is conducted; hence, there is no need for the `CommonItemSpec` structure.

---

<sup>2</sup>This statement should not be interpreted as a recommendation to set  $w_1 = 1$ .

`Wrapper_IRTeq()` is the wrapper function for IRT true-score and observed-score equating in the raw-score metric.<sup>3</sup> Note that `Wrapper_IRTeq()` does not compute equated scale scores. That is the purpose of `Wrapper_ESS()` (see page 23).

For IRT equating there are two approaches to obtaining moments. They can be obtained using the actual frequency distribution for group 1 that took the new form; that is the general approach taken in *Equating Recipes*. Alternatively, for IRT equating, only, moments can be based on ability distributions. `Wrapper_IRTeq()` computes both types of moments for equated raw scores (assuming the necessary distributions are provided), and `Print_IRTeq()` prints them. Moments for equated scale scores are computed and printed using a separate function, `Print_ESS_QD()`.

### 16.3.1 Wrapper\_IRTeq()

The function prototype for `Wrapper_IRTeq()` is:

```
void Wrapper_IRTeq(char design, char method, double w1,
    char ItemNewFile[], char ItemOldFile[],
    char DistNewFile[], char DistOldFile[],
    struct ItemSpec *NewItems, struct ItemSpec *OldItems,
    struct RawFitDist *NewForm, struct RawFitDist *OldForm,
    struct RawTruObsEquiv *RawEq, struct IRTstControl *StInfo, int *NewFD,
    struct IRT_INPUT *irtall, struct PDATA *pinall, struct ERAW_RESULTS *r)
```

The input variables are:

- ▷ `design` = 'R' (random-groups design), or 'S' (single groups design, or 'C' (common-item nonequivalent groups design) ;
- ▷ `method` = 'T' (IRT true score equating), or 'O' (IRT observed score equating), or 'A' (both IRT true-score and observed-score equating);
- ▷ `w1` = weight for new group ( $0 \leq w1 \leq 1$ );
- ▷ `ItemNewFile[]` = name of file containing item parameters for new form X on the scale of Y (see comments for `ItemInfoRead()` function in Section 15.3.2);
- ▷ `ItemOldFile[]` = name of file containing item parameters for old form Y (see comments for `ItemInfoRead()` function in Section 15.3.2);

---

<sup>3</sup>The raw-score metric results are those obtained prior to using a scale score conversion. Thus, the phrase “raw-score metric” applies to both IRT observed score equating and IRT true score equating.



- ▷ `DistNewFile[]` = name of file containing the ability distribution for new form X on the scale of Y (see comments for `ThetaInfoRead()` function in Section 15.3.2);
- ▷ `DistOldFile[]` = name of file containing the ability distribution for old form Y (see comments for `ThetaInfoRead()` function in Section 15.3.2);
- ▷ `NewItems` = `ItemSpec` structure containing new-form item parameter estimates on the scale of Y;
- ▷ `OldItems` = `ItemSpec` structure containing old-form item parameter estimates;
- ▷ `NewForm` = `RawFitDist` structure containing new-form IRT fitted distribution;
- ▷ `OldForm` = `RawFitDist` structure containing old-form IRT fitted distribution;
- ▷ `RawEq` = `RawTruObsEquiv` structure containing IRT true-score and observed-score equivalents;
- ▷ `StInfo` = `IRTstControl` structure containing general information (e.g., numbers of score points, minimum scores, maximum scores, etc.) as well as the ability distributions read from the files `DistNewFile[]` and `DistOldFile[]`; and
- ▷ `NewFD` = new-form group 1 actual relative frequency distribution.

The output variables are:

- ▷ `irtall` = `IRT_INPUT` structure that stores all IRT input data (as well as some output);
- ▷ `pinall` = `PDATA` structure that stores all input and includes the `IRT_INPUT` structure `irtall` as a member; and
- ▷ `r` = `ERAW_RESULTS` structure that includes IRT equivalents and moments.

Strictly speaking `design` is not ever used by `Wrapper_IRTeq()`. It is included as one of the parameters for consistency with other wrapper functions. In *Equating Recipes*, design issues are addressed through scale transformation.

In general, `Wrapper_IRTeq()` computes moments based on ability distributions as well as the new-form actual frequency distribution. There are exceptions, however. If `NewFD` = `NULL`, then no moments are computed based on the new form actual frequency distribution. Also, if *only* IRT true-score

equating is requested (i.e., `method = 'T'`), then the ability distributions in `DistNewFile[]` and `DistOldFile[]` are not read and, obviously, moments based on ability distributions cannot be computed. (When `method = 'T'`, both `DistNewFile[]` and `DistOldFile[]` can be set to `NULL`.)

### 16.3.2 Print\_IRTeq()

`Print_IRTeq()` is the print function associated with `Wrapper_IRTeq()`. The function prototype for `Print_IRTeq()` is:

```
void Print_IRTeq(FILE *fp, char tt[], struct PDATA *pinall,
                 struct ERAW_RESULTS *r, int PrintFiles)
```

This function prints the IRT raw-score-metric results computed by `Wrapper_IRTeq()`. The input variables are:

- ▷ `fp` = file pointer for output;
- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ `pinall` = a `PDATA` structure used to pass variables from one function to another; and
- ▷ `r` = an `ERAW_RESULTS` structure that contains equated raw-score results; and
- ▷ `PrintFiles`: 0 mean don't print input item parameter and quadrature files (if provided); 1 means print input item parameter and quadrature files (if provided).

### 16.3.3 Print\_ESS\_QD()

The function prototype for `Print_ESS_QD()` is:

```
void Print_ESS_QD(FILE *fp, char tt[], struct PDATA *pinall,
                  struct ESS_RESULTS *s, struct RawFitDist *NewForm,
                  struct RawFitDist *OldForm)
```

`Print_ESS_QD()` computes and prints equated scale score moments based on ability distributions. `Wrapper_ESS()` must be called before `Print_ESS_QD()`, because `Wrapper_ESS()` computes unrounded and round-

ed scale-score conversions that are used by `Print_ESS_QD()`. The input variables are:

- ▷ `fp` = file pointer for output;

- ▷ `tt[]` = user supplied text identifier that is printed at the top of the file;
- ▷ `pinall` = a `PDATA` structure used to pass variables from one function to another;
- ▷ `s` = an `ESS_RESULTS` structure that contains equated scale-score conversions (both unrounded and rounded);
- ▷ `NewForm` = a `RawFitDist` structure for the new form that contains the fitted relative frequency distribution for group 1 that took the new form; and
- ▷ `OldForm` = a `RawFitDist` structure for the old form that contains the fitted relative frequency distribution for group 2 that took the old form.

`Print_ESS_QD()` should be called only if `method = 'O'` or `method = 'A'` because ability distributions are read (by `Wrapper_IRTeq()`) only if IRT observed score equating is requested. Note that moments computed and printed by `Print_ESS_QD()` are not stored in any structure.

#### 16.3.4 Set-up and Clean-up

As in Chapter 15, before calling the functions that conduct IRT equating, a preliminary process is required to set up the computing environment. The set-up process includes: (1) using DMA to initialize pointers to variables of the `ItemSpec` and `IRTstControl` structures which are defined for the input of items and ability distributions, (2) reading the input for items for the new and old forms, (3) reading the input for ability distributions for the new and old examinee groups, and (4) using DMA to prepare the memory for saving equating results. For the set-up, four “stand-alone” functions are used: (1) `ItemInfoRead()`, (2) `ThetaInfoRead()`, (3) `RawFitMem()`, and (4) `RawEqResultsMem()`. `ItemInfoRead()` and `ThetaInfoRead()` must be called before the “Mem” functions are executed. As described in Chapter 15, the `ItemInfoRead()` and `ThetaInfoRead()` functions work independently and thus either can be called first. Likewise, the “Mem” functions work independently.

The two “Mem” functions are associated with their respective structure types, which are defined in the header file `IRTeq.h`. The function `RawFitMem()` allocates memory to the pointer members of the `RawFitDist` structure that is defined for saving the distributions estimated by IRT. The function `RawEqResultsMem()` initializes memory for the pointer members of the `RawTruObsEquiv` structure that is defined for saving raw-to-raw score conversions from IRT true and observed score equating.

Four memory deallocation functions can be used to clean up the memory prepared by the set-up process: (1) `RawFitDeAlloc()`,

(2) `RawEqResultsDeAlloc()`, (3) `StItemDeAlloc()`, and (4) `StContDeAlloc()`. The first two “DeAlloc” functions correspond to the two “Mem” functions, in order. `StItemDeAlloc()` and `StContDeAlloc()` are described in Section 15.3.2. The first two “DeAlloc” functions must be called before `StItemDeAlloc()` and `StContDeAlloc()`. `StContDeAlloc()` must be called last.

Provided below are the declarations and descriptions of functionality for each of the set-up and clean-up functions. The functions `ItemInfoRead()`, `ThetaInfoRead()`, `StItemDeAlloc()`, and `StContDeAlloc()`, are not discussed further here, since they are described in Chapter 15.

```
void RawFitMem(struct ItemSpec *Items, const char *oldOrnew,
               struct IRTstControl *Handle, struct RawFitDist *Form)
/*-----
  Functionality:
    Allocate memory to the pointer members of the RawFitDist structure.

  Input:
    Items      A pointer to an array of the ItemSpec structure
    oldOrnew   A string indicating that the array is about either
               the old or new form (either "old" or "new" must be used
               as characters)
    Handle     A pointer to a variable of the IRTstControl structure
    Form       A pointer to an object of the RawFitDist structure
-----*/

void RawEqResultsMem(struct ItemSpec *NewItems, struct IRTstControl *Handle,
                     struct RawTruObsEquiv *RawEq)
/*-----
  Functionality:
    Allocate memory to the pointer members of the RawTruObsEquiv structure.

  Input:
    NewItems   A pointer to an array of the ItemSpec structure for
               the new form
    Handle     A pointer to a variable of the IRTstControl structure
    RawEq      A pointer to an object of the RawTruObsEquiv structure
-----*/

void RawFitDeAlloc(struct RawFitDist *Form)
/*-----
  Functionality:
    Deallocate memory given to an object of the RawFitDist structure.

  Input:
    Form       A pointer to an object of the RawFitDist structure
-----*/
```

```
void RawEqResultsDeAlloc(struct RawTruObsEquiv *RawEq)
/*-----
  Functionality:
    Deallocate memory given to an object of the RawTruObsEquiv structure.

  Input:
    RawEq      A pointer to an object of the RawTruObsEquiv structure
-----*/
```

16.3.5 trueScoreEq()

trueScoreEq() performs IRT true score equating with mixed-format tests. To find the  $\theta$ -equivalent of a Form X true score, the function trueScoreEq() calls the function er\_rtsafe(). A more detailed function call structure is given below:

```
/*-----
trueScoreEq      calls      er_rtsafe
                                trueScore

er_rtsafe        calls      funcd

trueScore        calls      ProbCCC

funcd            calls      f_mix
                                f_mixDer

f_mix            calls      ProbCCC

f_mixDer         calls      PdCCCoverTheta

ProbCCC          calls      Prob3PL
                                ProbLGR
                                ProbGPC
                                ProbNRM

PdCCCoverTheta   calls      Pd3PLoverTheta
                                PdLGRoverTheta
                                PdGPCoverTheta
                                PdNRMoverTheta
-----*/
```

The declaration and a description of the functionality of `trueScoreEq()` are provided next.

```
short trueScoreEq(struct IRTstControl *Handle, struct ItemSpec *NewItems,
                 struct ItemSpec *OldItems, int nScores, const double *newScores,
                 double *eqvOld, double *theta, double *newMin, double *OldMin)
/*-----
  Functionality:
    Performs IRT true score equating with mixed-format tests.

  Input:
    Handle      A pointer to control the computing environment for equating
                The same type of structure as used for IRT scale
                transformation.

    NewItems    A pointer to designate items on the new form (0-offset)
    OldItems    A pointer to designate items on the old form (0-offset)
    nScores     Number of new form scores for which old form equivalents
                are to be computed
    newScores   New form scores for which old form equivalents are to be
                computed. These scores are assumed to be equally spaced;
                the difference between consecutive elements is assumed to be
                constant (0-offset).

  Output:
    eqvOld      vector of old form equivalents of new form scores; 0-offset
    theta       vector of theta values which produce the vector of integer
                new form true scores.

    *NewMin     the lowest possible score on the new form.
    *OldMin     the lowest possible score on the old form.
  -----*/
```

### 16.3.6 IRTmixObsEq()

`IRTmixObsEq()` performs IRT observed score equating with mixed-format tests. To find the Form Y equivalents of Form X observed scores, the `IRTmixObsEq()` function uses the `IRTmixObsDist()` and `EquiEquate()` functions, the latter of which is described on page 21. The function `IRTmixObsDist()` computes the marginal (observed scores) distribution for a mixed-format test form that is estimated by IRT models. The marginal distribution is obtained by calling `ObsDistGivenTheta()` which uses the function `recurs()` to compute the conditional distribution of observed scores given an ability value. The function `EquiEquate()` is *Equating Recipe*'s conventional equipercentile equating function (see page 21). A more detailed function call structure is given next.

```
/*-----
  IRTmixObsEq           calls  IRTmixObsDist
                        EquiEquate

  IRTmixObsDist         calls  ObsDistGivenTheta
  -----*/
```

```
ObsDistGivenTheta      calls   ProbCCC
                           recurs
```

```
ProbCCC                calls   Prob3PL
                           ProgLGR
                           ProbGPC
                           ProbNRM
```

```
-----*/
```

The declaration and a description of the functionality of IRTmixObsEq() are provided next.

```
short IRTmixObsEq(struct IRTstControl *Handle, struct ItemSpec *NewItems,
                  struct ItemSpec *OldItems, double wNew, double wOld,
                  struct RawFitDist *newForm, struct RawFitDist *oldForm,
                  struct RawTruObsEquiv *RawEq)
/*-----*/
Functionality:
    Calculation of IRT observed score equivalents of new form scores.

Input:
    Handle      A pointer to control the computing environment for equating;
                 the same type of structure as used for IRT scale
                 transformation.
    NewItems    A pointer to designate items on the new form (0-offset)
    OldItems    A pointer to designate items on the old form (0-offset)
    wNew        New group weight for a synthetic group
    wOld        Old group weight for a synthetic group; wNew + wOld = 1.0
    newForm     A pointer to an object of the RawFitDist structure for the
                 new form
    oldForm     A pointer to an object of the RawFitDist structure for the
                 old form
    RawEq       A pointer to an object of the RawTruObsEquiv structure to
                 save the equating results for raw scores.

Output:
    newForm     Fitted distributions for the new form for the new, old, and
                 synthetic groups
    oldForm     Fitted distributions for the old form for the new, old, and
                 synthetic groups
    RawEq       Old form raw-score equivalents of new form scores;
                 The results are saved into RawEq->unroundedEqObs.
/*-----*/
```

## 16.4 Examples

Provided next are two examples that are essentially a continuation of the examples in Section 15.4. The first example involves mixed-format test forms; the second involves multiple-choice test forms. The mixed-format example is more complicated and described in more detail.

### 16.4.1 Mixed Format

The example discussed here uses the test data of Example 3 in Chapter 15 (see Section 15.4.1). Recall that each of the new and old test forms includes ten MC items and five CR items, with the minimum and maximum raw scores being 0 and 25, respectively. It is assumed that:

- the `dummyYitems` and `dummyYdist` files contain the item parameters and ability distribution, respectively, for the old form (Form Y); and
- the item parameters and ability distribution for the new form (Form X) have been converted to the old scale (Form-Y scale) and are in the files `dummyXitems.ST` and `dummyXdist.ST`, respectively.

Using `Wrapper_IRTeq()`, Table 16.1 provides the `main()` function for this example.<sup>4</sup> The synthetic population is defined in `Wrapper_IRTeq()` to be the population for the new Form X; i.e.,  $w_1 = 1$ .

Five types of structures are required in the `main()` function for the raw-score IRT equating:<sup>5</sup>

- `ItemSpec` (defined in `IRTst.h`) is used to store the item parameter estimates for the old form and the transformed item parameter estimates for the new form;
- `IRTstControl` (defined in `IRTst.h`) is used to store the ability distribution for the old form and the transformed ability distribution for the new form, as well as some other variables;
- `RawFitDist` (defined in `IRTeq.h`) is used to store new form and old form distributions of test scores fitted by IRT models;
- `RawTruObsEquiv` (defined in `IRTeq.h`) is used to save equating results for both the IRT true and observed score equating methods; and
- `IRT_INPUT` (defined in `IRTeq.h`) is used to store IRT input and results including those stored in the above four structures.

The raw-score equating results are saved in the file named `Chap 16 out (dummy)`, which is listed in Tables 16.2 and 16.3. Table 16.2 shows that the two IRT raw-score equating methods yielded very similar results. The names of the files input to *Equating Recipes* are provided at the top of the output. It is especially important to verify that these are the correct file names. In particular, it is the transformed results that must be used for the new form and new group.

---

<sup>4</sup>The comments following each of the parameters in `Wrapper_IRTeq()` are not required.

<sup>5</sup>Other structures are required as well for `Wrapper_IRTeq()`, as indicated at the top of Table 16.1. These structures have been described in other chapters.



The first set of moments at the bottom of Table 16.2 are the usual moments based on using the actual distribution of raw scores for the group that took the new form. We will call these moments “standard moments.” The computation of IRT raw-score equating results does not actually require the actual distribution of raw scores for the group that took the new form. Consequently, if the user wants the standard moments, then this distribution must be provided to *Equating Recipes*. That is the purpose of the following code in Table 16.1:

```
struct USTATS          x;

ReadFdGet_USTATS("fdexample.dat",1,2,0,25,1,'X',&x);
```

The second set of moments at the bottom of Table 16.2 does not require the actual distribution of raw scores for the group that took the new form. Rather these moments use the IRT ability distribution for group 1 that took the new Form X. We will use the term “IRT fitted moments” for moments based on an ability distribution. IRT fitted moments should be interpreted with caution. In effect, they are the moments of expected distributions given the IRT models used. They are not necessarily the actual moments for any particular subset of examinees.

In *Equating Recipes*, IRT fitted moments are provided only if IRT observed score equating is requested (i.e., `Method` is 0 or A), the rationale being that IRT true score equating does not require ability (or quadrature) distributions. However, when both IRT true-score and observed-score equating are requested, IRT fitted moments are provided for both methods.

Table 16.3 provides old-form and new-form IRT fitted distributions for group 1, group 2, and the synthetic group. (Due to space limitations only partial distributions are provided.) The associated IRT fitted moments are also provided. IRT fitted moments for different groups should be compared cautiously.

In the code in Table 16.1, `Wrapper_ESS()` (see page 23) reads the conversion table (hypothetical, in this example) and computes both rounded and unrounded scale scores. Then, `Print_ESS()` (see page 25) prints these results, which are provided in Tables 16.4, 16.5, and 16.6, respectively. Tables 16.5 and 16.6 also provide the standard moments for unrounded and rounded scale scores, respectively. (In general, these moments are provided only if the actual new-form group 1 frequency distribution is available, as it is for this example.)

Scale-score IRT fitted moments are computed and printed by the function `Print_ESS_QD()`, provided IRT observed score equating is requested (i.e., `Method` is 0 or A in `Wrapper_IRTeq()`). For this dummy data example, scale-score IRT fitted moments are provided in Table 16.7.

TABLE 16.1. Main() Code to Illustrate IRT True-Score and Observed-Score Equating for a Mixed-Format Dummy Data Example) Example

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct ItemSpec      NewItems, OldItems;
    struct RawFitDist    NewForm, OldForm;
    struct RawTruObsEquiv RawEq;
    struct IRTstControl  StInfo;
    struct IRT_INPUT     irtall;
    struct PDATA         pdirt;
    struct ERAW_RESULTS  er;
    struct USTATS        x;
    struct ESS_RESULTS   es;

    FILE *outf;

    outf = fopen("Chap 16 out (dummy)", "w");

    /* Equating of Form X (new form) to Form Y (old form) using
       the item parameters from Example 3 of Chapter 15 that are
       placed on the Form Y scale */

    ReadFdGet_USTATS("fdexample.dat", 1, 2, 0, 25, 1, 'X', &x);

    /* IRT raw score equating */
    Wrapper_IRTeq(
        'C', /* Design (R/S/C) */
        'A', /* Method (T=true/O=observed/A=both) */
        1.0, /* weight for new group that took X */
        "dummyXitems.ST", /* name of file containing parameters for X */
        "dummyYitems", /* name of file containing parameters for Y */
        "dummyXdists.ST", /* name of file containing theta dist for X */
        "dummyYdist", /* name of file containing theta dist for Y */
        &NewItems, /* struct for parameters for X items */
        &OldItems, /* struct for parameters for Y items */
        &NewForm, /* struct for fitted dist for X */
        &OldForm, /* struct for fitted dist for Y */
        &RawEq, /* struct for IRT true and observed equating */
        &StInfo, /* struct containing quadrature dist's */
        x.fd, /* actual freq dist for group that took X */
        &irtall, /* struct that includes all IRT info */
        &pdirt, /* struct PDATA that contains "all" input */
        &er /* equated raw-score results */
    );
    Print_IRTeq(outf, "Mixed Format Dummy Data Example", &pdirt, &er, 0);

    /* IRT scale score results */
    Wrapper_ESS(&pdirt, &er, 0, 25, 1, "convRawSS2.in", 2, -5, 255, &es);
    Print_ESS(outf, "Mixed Format Dummy Data Example", &pdirt, &es);
    Print_ESS_QD(outf, "Mixed Format Dummy Data Example", &pdirt, &es,
        &NewForm, &OldForm);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 16.2. Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example

```

/*****
Mixed Format Dummy Data Example

```

#### IRT True Score Equating and Observed Score Equating

```

Input Design = C (ignored)
Name of file containing item parameter estimates for X: dummyXitems.ST
Name of file containing item parameter estimates for Y: dummyYitems
w1 = weight for new group that took X = 1.00000
Name of file containing theta distribution for X: dummyXdistribution.ST
Name of file containing theta distribution for Y: dummyYdistribution
Number of score categories for the new form = 26
Number of score categories for the old form = 26

```

Equated Raw Scores			
Raw Score (X)	Method 0: IRT Tr Scr	Method 1: IRT Ob Scr	theta for IRT Tr Scr
0.00000	0.00000	0.11673	-99.00000
1.00000	1.20258	1.11392	-99.00000
2.00000	2.17866	2.10105	-3.96135
3.00000	3.12596	3.08675	-2.99898
4.00000	4.10392	4.07555	-2.50968
5.00000	5.09336	5.06551	-2.16075
6.00000	6.08758	6.05239	-1.87578
7.00000	7.08182	7.03633	-1.62575
8.00000	8.07219	8.02307	-1.39701
9.00000	9.05609	9.01550	-1.18219
10.00000	10.03338	10.01000	-0.97667
11.00000	11.00679	10.99953	-0.77682
12.00000	11.98069	11.97886	-0.57943
13.00000	12.95982	12.95479	-0.38141
14.00000	13.94772	13.94250	-0.17983
15.00000	14.94572	14.94580	0.02801
16.00000	15.95332	15.95938	0.24470
17.00000	16.96839	16.97872	0.47263
18.00000	17.98807	18.00032	0.71418
19.00000	19.00933	19.02026	0.97184
20.00000	20.03004	20.03690	1.24923
21.00000	21.04871	21.05122	1.55300
22.00000	22.06348	22.06333	1.89783
23.00000	23.06981	23.06807	2.32149
24.00000	24.05812	24.06021	2.95823
25.00000	25.00000	25.03341	99.00000

#### Moments based on actual frequency dist for new form

Mean	15.99019	15.98717
S.D.	5.22588	5.23981
Skew	-0.03290	-0.03656
Kurt	2.07164	2.07447

#### Moments based on IRT fitted dist for new form

Mean	17.38785	17.38766
S.D.	5.13337	5.14513
Skew	-0.64543	-0.64944
Kurt	2.82652	2.83400

```

/*****

```

TABLE 16.3. Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: IRT Fitted Distributions and Moments

/\*\*\*\*\*/

Relative Frequency Distributions for New Form X

Using IRT Fitted Distributions

Raw Score (X)	grp 1	grp 2	grp s
0.00000	0.00033	0.00043	0.00033
...	...	...	...
10.00000	0.02746	0.04654	0.02746
11.00000	0.02995	0.05510	0.02995
12.00000	0.03228	0.06340	0.03228
13.00000	0.03761	0.06845	0.03761
14.00000	0.04728	0.07014	0.04728
15.00000	0.05813	0.07176	0.05813
16.00000	0.06480	0.07516	0.06480
17.00000	0.06641	0.07679	0.06641
18.00000	0.06881	0.07223	0.06881
19.00000	0.07607	0.06300	0.07607
...	...	...	...
25.00000	0.03995	0.00476	0.03995

Relative Frequency Distributions for Old Form Y

Using IRT Fitted Distributions

Raw Score (Y)	grp 1	grp 2	grp s
0.00000	0.00027	0.00035	0.00027
...	...	...	...
10.00000	0.02759	0.04767	0.02759
11.00000	0.03040	0.05616	0.03040
12.00000	0.03320	0.06403	0.03320
13.00000	0.03879	0.06870	0.03879
14.00000	0.04813	0.07030	0.04813
15.00000	0.05804	0.07184	0.05804
16.00000	0.06378	0.07469	0.06378
17.00000	0.06502	0.07556	0.06502
18.00000	0.06739	0.07075	0.06739
19.00000	0.07451	0.06188	0.07451
...	...	...	...
25.00000	0.04281	0.00521	0.04281

-----  
Raw Score Moments (based on IRT fitted distributions)

Group	Form	mean	sd	skew	kurt
1	X	17.37318	5.12703	-0.67046	2.88708
2	X	14.56084	4.93020	-0.27127	2.49635
s	X	17.37318	5.12703	-0.67046	2.88708
1	Y	17.38764	5.14537	-0.64954	2.83409
2	Y	14.56169	4.92995	-0.24656	2.48012
s	Y	17.38764	5.14537	-0.64954	2.83409

/\*\*\*\*\*/

TABLE 16.4. Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Scale Score Conversion Table

```

/*****
Mixed Format Dummy Data Example

Name of file containing yct[]: convRawSS2.in
Minimum raw score in base form conversion, Y =      0.00000
Maximum raw score in base form conversion, Y =     25.00000
Lowest possible rounded scale score =      -5
Highest possible rounded scale score =     255

CONVERSION TABLE FOR Y (BASE FORM)

```

Raw Score	Y   Scale   Score
-0.50000	-5.00000
0.00000	0.00000
1.00000	10.00000
2.00000	20.00000
3.00000	30.00000
4.00000	40.00000
5.00000	50.00000
6.00000	60.00000
7.00000	70.00000
8.00000	80.00000
9.00000	90.00000
10.00000	100.00000
11.00000	110.00000
12.00000	120.00000
13.00000	130.00000
14.00000	140.00000
15.00000	150.00000
16.00000	160.00000
17.00000	170.00000
18.00000	180.00000
19.00000	190.00000
20.00000	200.00000
21.00000	210.00000
22.00000	220.00000
23.00000	230.00000
24.00000	240.00000
25.00000	250.00000
25.50000	255.00000

```

/*****

```

TABLE 16.5. Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Unrounded Scale Scores

```

/*****/
Mixed Format Dummy Data Example

```

```

Name of file containing yct[]: convRawSS2.in

```

Equated Scale Scores (unrounded)		
Raw Score (x)	Method 0: IRT Tr Scr	Method 1: IRT Ob Scr
0.00000	0.00000	1.16734
1.00000	12.02583	11.13919
2.00000	21.78659	21.01047
3.00000	31.25956	30.86746
4.00000	41.03924	40.75554
5.00000	50.93362	50.65508
6.00000	60.87576	60.52388
7.00000	70.81824	70.36332
8.00000	80.72190	80.23073
9.00000	90.56088	90.15499
10.00000	100.33383	100.10002
11.00000	110.06795	109.99525
12.00000	119.80691	119.78862
13.00000	129.59817	129.54791
14.00000	139.47720	139.42502
15.00000	149.45715	149.45799
16.00000	159.53317	159.59382
17.00000	169.68386	169.78716
18.00000	179.88069	180.00320
19.00000	190.09333	190.20260
20.00000	200.30037	200.36904
21.00000	210.48713	210.51217
22.00000	220.63476	220.63327
23.00000	230.69810	230.68066
24.00000	240.58117	240.60208
25.00000	250.00000	250.33410
Mean	159.90194	159.87171
S.D.	52.25884	52.39812
Skew	-0.03290	-0.03656
Kurt	2.07164	2.07447

```

/*****/

```

TABLE 16.6. Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Rounded Scale Scores

```

/*****/
Mixed Format Dummy Data Example

```

```

Name of file containing yct[]: convRawSS2.in

```

Equated Scale Scores (rounded)		
Raw Score (x)	Method 0:	Method 1:
	IRT Tr Scr	IRT Ob Scr
0.00000	0	0
1.00000	10	10
2.00000	20	20
3.00000	30	30
4.00000	40	40
5.00000	50	50
6.00000	60	60
7.00000	70	70
8.00000	80	80
9.00000	90	90
10.00000	100	100
11.00000	110	110
12.00000	120	120
13.00000	130	130
14.00000	140	140
15.00000	150	150
16.00000	160	160
17.00000	170	170
18.00000	180	180
19.00000	190	190
20.00000	200	200
21.00000	210	210
22.00000	220	220
23.00000	230	230
24.00000	240	240
25.00000	250	250
Mean	159.81677	159.81677
S.D.	52.23452	52.23452
Skew	-0.05389	-0.05389
Kurt	2.09191	2.09191

```

/*****/

```

TABLE 16.7. Output Illustrating IRT True-score and Observed-score Equating for a Mixed-Format Dummy Data Example: Scale Score Moments Based on IRT Fitted Distributions

```

/*****/
Scale Score Moments (based on IRT fitted distributions)
  TX means based on Form X IRT true score equating
  OX means based on Form X IRT observed score equating

Group Form      mean      sd      skew      kurt

Unrounded:

  2      Y      145.61692  49.29952  -0.24656  2.48012
  1      TX      173.87852  51.33365  -0.64543  2.82652
  1      OX      173.87664  51.45128  -0.64944  2.83400

Rounded:

  2      Y      145.61692  49.29952  -0.24656  2.48012
  1      TX      173.73182  51.27026  -0.67046  2.88708
  1      OX      173.73182  51.27026  -0.67046  2.88708

/*****/

```



### 16.4.2 3PL

The example discussed here is a continuation of the 3PL example in Section 15.4.2. Recall that each of the new and old test forms includes 36 dichotomously-scored items, with responses modelled using the 3PL model. It is assumed that:

- the `KB6Yitems` and `KB6Ydist` files contain the item parameters and ability distribution, respectively, for the old form (Form Y); and
- the item parameters and ability distribution for the new form (Form X) have been converted to the old scale (Form-Y scale) and are in the files `KB6Xitems.ST` and `KB6Xdist.ST`, respectively.

Using `Wrapper_IRTeq()`, Table 16.8 provides the `main()` function for this example.<sup>6</sup> The synthetic population is defined in `Wrapper_IRTeq()` to be the population for the new Form X; i.e.,  $w_1 = 1$ . The output is provided in Tables 16.9 and 16.10.

For this example, the data sets used to estimate the item parameters are the same as those used for the examples in Sections 4.4, 6.6, and 10.6.<sup>7</sup> For the new form data set, the actual distribution of raw scores is obtained using the following code in Table 16.8:

```
struct BSTATS          xv;
int fieldsACT[2][3] = {{0,37,38},{0,39,40}};

convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'X','V',&xv);
```

<sup>6</sup>The comments following each of the parameters in `Wrapper_IRTeq()` are not required.

<sup>7</sup>The IRT true-score and observed-score equivalents in Table 16.9 are not quite the same as those in Kolen and Brennan (2004, Tables 6.9 and 6.11). One reason for this is that the item parameter estimates in files `KB6Yitems` and `KB6Xitems.ST` used as input to *Equating Recipes* are identical to those in Kolen and Brennan (2004, Tables 6.8), which are rounded to four decimal places, whereas more precise estimates were used to obtain the equivalents reported in Kolen and Brennan (2004). A similar comment applies to the ability distributions.

TABLE 16.8. Main() Code to Illustrate IRT True-Score and Observed-Score Equating for Kolen and Brennan (2004, chap. 6) Example

```

/*****
#include "ERutilities.h"
#include "ER.h"

int main(void)
{
    struct ItemSpec      NewItems, OldItems;
    struct RawFitDist    NewForm, OldForm;
    struct RawTruObsEquiv RawEq;
    struct IRTstControl  StInfo;
    struct IRT_INPUT     irtall;
    struct PDATA         pdirt;
    struct ERAW_RESULTS  er;
    struct BSTATS        xv;

    FILE *outf;
    int fieldsACT[2][3] = {{0,37,38},{0,39,40}};

    outf = fopen("Chap 16 out (K&B Chap 6 Example)", "w");

    /* IRT true-score and observed score equating for
       Kolen and Brennan (2004, Chap 6) example. Wrapper_IRTst()
       for Kolen and Brennan (2004, Chap 6) example could be placed
       here, with no changes or additions of other code */

    /* Actual frequency distribution for new group on X: see
       Chapter 6 example in Equating Recipes (2008, p. 65-69)
       Chapter 5 example in Kolen and Brennan (2004, p. 151) */

    convertFtoW("mondax.dat",2,fieldsACT,"mondax-temp");
    ReadRawGet_BSTATS("mondax-temp",1,2,0,36,1,0,12,1,'X','V',&xv);

    Wrapper_IRTeq(
        'C',                                /* Design (R/S/C) */
        'A',                                /* Method (T=true/O=observed/A=both) */
        1.0,                                /* weight for new group that took X */
        "KB6Xitems.ST", /* name of file containing parameters for X */
        "KB6Yitems",    /* name of file containing parameters for Y */
        "KB6Xdists.ST", /* name of file containing theta dist for X */
        "KB6Ydist",     /* name of file containing theta dist for Y */
        &NewItems,       /* struct for parameters for X items */
        &OldItems,       /* struct for parameters for Y items */
        &NewForm,        /* struct for fitted dist for X */
        &OldForm,        /* struct for fitted dist for Y */
        &RawEq,          /* struct for IRT true and observed equating */
        &StInfo,         /* struct containing quadrature dist's */
        xv.fdl,          /* actual freq dist for group that took X */
        &irtall,         /* struct that includes all IRT info */
        &pdirt,          /* struct PDATA that contains "all" input */
        &er              /* equated raw-score results */
    );
    Print_IRTeq(outf, "K&B Chapter 6 Example: w1 = 1", &pdirt, &er, 0);

    fclose(outf);
    return 0;
}
*****/

```

TABLE 16.9. Output Illustrating IRT True-Score and Observed-Score Equating for Kolen and Brennan (2004, chap. 6) Example

```

/*****
K&B Chapter 6 Example: w1 = 1
IRT True Score Equating and Observed Score Equating
Input Design = C (ignored)
Name of file containing item parameter estimates for X: KB6Xitems.ST
Name of file containing item parameter estimates for Y: KB6Yitems
w1 = weight for new group that took X = 1.00000
Name of file containing theta distribution for X: KB6Xdistribution.ST
Name of file containing theta distribution for Y: KB6Ydistribution
Number of score categories for the new form = 37
Number of score categories for the old form = 37
-----

```

Equated Raw Scores			
Raw Score (X)	Method 0:	Method 1:	theta for
	IRT Tr Scr	IRT Ob Scr	IRT Tr Scr
0.00000	0.00000	-0.34282	-99.00000
1.00000	0.88802	0.61797	-99.00000
2.00000	1.77604	1.58023	-99.00000
3.00000	2.66406	2.54607	-99.00000
4.00000	3.55208	3.51863	-99.00000
5.00000	4.44010	4.50272	-99.00000
6.00000	5.32812	5.50491	-99.00000
7.00000	6.13415	6.53186	-4.33553
8.00000	7.18658	7.58611	-2.76939
9.00000	8.39639	8.66206	-2.06260
10.00000	9.62363	9.74833	-1.60654
11.00000	10.82783	10.83677	-1.26757
12.00000	12.00268	11.93076	-0.99450
13.00000	13.15215	13.04597	-0.76275
14.00000	14.28300	14.19741	-0.55876
15.00000	15.40232	15.37004	-0.37427
16.00000	16.51639	16.51341	-0.20382
17.00000	17.63001	17.59764	-0.04359
18.00000	18.74585	18.64402	0.10918
19.00000	19.86405	19.67901	0.25658
20.00000	20.98213	20.73909	0.40018
21.00000	22.09532	21.87860	0.54124
22.00000	23.19760	23.10523	0.68084
23.00000	24.28309	24.29262	0.82004
24.00000	25.34755	25.36488	0.96006
25.00000	26.38954	26.36725	1.10245
26.00000	27.41076	27.34587	1.24930
27.00000	28.41560	28.32431	1.40338
28.00000	29.40991	29.32187	1.56840
29.00000	30.39912	30.35358	1.74941
30.00000	31.38562	31.38002	1.95358
31.00000	32.36469	32.34828	2.19186
32.00000	33.31866	33.28258	2.48268
33.00000	34.21002	34.20073	2.86060
34.00000	34.98014	35.07631	3.39938
35.00000	35.57570	35.85308	4.32158
36.00000	36.00000	36.39047	99.00000

```

-----
Moments based on actual frequency dist for new form
      Mean      16.18296      16.18289
      S.D.       7.20030      7.11341
      Skew       0.49745      0.53788
      Kurt       2.51835      2.57169
-----
Moments based on IRT fitted dist for new form
      Mean      16.17533      16.17846
      S.D.       7.19742      7.11139
      Skew       0.50259      0.54289
      Kurt       2.52548      2.57726
/*****

```

TABLE 16.10. Output Illustrating IRT True-score and Observed-score Equating for Kolen and Brennan (2004, chap. 6)  
Example: IRT Fitted Distributions and Moments

/\*\*\*\*\*

Relative Frequency Distributions for New Form X

Using IRT Fitted Distributions

Raw Score (X)	grp 1	grp 2	grp s
0.00000	0.00001	0.00000	0.00001
...	...	...	...
14.00000	0.05796	0.05913	0.05796
15.00000	0.05607	0.05831	0.05607
16.00000	0.05549	0.05578	0.05549
17.00000	0.05407	0.05363	0.05407
18.00000	0.04976	0.05328	0.04976
19.00000	0.04233	0.05411	0.04233
20.00000	0.03377	0.05383	0.03377
21.00000	0.02701	0.05039	0.02701
22.00000	0.02400	0.04389	0.02400
23.00000	0.02446	0.03671	0.02446
24.00000	0.02613	0.03173	0.02613
...	...	...	...
36.00000	0.00031	0.00021	0.00031

Relative Frequency Distributions for Old Form Y

Using IRT Fitted Distributions

Raw Score (Y)	grp 1	grp 2	grp s
0.00000	0.00002	0.00000	0.00002
...	...	...	...
14.00000	0.05015	0.05444	0.05015
15.00000	0.04809	0.05446	0.04809
16.00000	0.04885	0.05163	0.04885
17.00000	0.05056	0.04790	0.05056
18.00000	0.05021	0.04586	0.05021
19.00000	0.04586	0.04677	0.04586
20.00000	0.03792	0.04946	0.03792
21.00000	0.02894	0.05094	0.02894
22.00000	0.02210	0.04860	0.02210
23.00000	0.01945	0.04230	0.01945
24.00000	0.02088	0.03457	0.02088
...	...	...	...
36.00000	0.00141	0.00121	0.00141

-----  
Raw Score Moments (based on IRT fitted distributions)

Group	Form	mean	sd	skew	kurt
1	X	15.81471	6.52395	0.58506	2.72512
2	X	18.02832	6.35835	0.28482	2.40416
s	X	15.81471	6.52395	0.58506	2.72512
1	Y	16.17437	7.12303	0.53807	2.57614
2	Y	18.66520	6.87875	0.22723	2.30574
s	Y	16.17437	7.12303	0.53807	2.57614

/\*\*\*\*\*

## References

- Baker, F.B. (1992). Equating tests under the graded response model. *Applied Psychological Measurement*, 16, 87–96.
- Baker, F.B. (1993). Equating tests under the nominal response model. *Applied Psychological Measurement*, 17, 239–251.
- Baker, F.B., & Al-Karni, A. (1991). A comparison of two procedures for computing IRT equating coefficients. *Journal of Educational Measurement*, 28, 147–162.
- Birnbaum, A. (1968). Some latent trait models and their use in inferring an examinee's ability. In F.M. Lord and M.R. Novick (Eds.), *Statistical theories of mental test scores* (pp. 397–479). Reading, MA: Addison-Wesley.
- Bock, R.D. (1972). Estimating item parameters and latent ability when responses are scored in two or more nominal categories. *Psychometrika*, 37, 29–51.
- Bock, R.D., & Aitkin, M. (1981). Marginal maximum likelihood estimation of item parameters: Application of an EM algorithm. *Psychometrika*, 46, 443–459.
- de Boor, C. (1978). *A practical guide to splines* (Applied Mathematical Sciences, Volume 27). New York: Springer-Verlag.

- Braun, H. I., & Holland, P. W. (1982). Observed-score test equating: A mathematical analysis of some ETS equating procedures. In P. W. Holland & D. B. Rubin (Eds.), *Test equating* (pp. 9–49). New York: Academic.
- Brennan, R. L. (2006, June). *Chained linear equating*. (CASMA Technical Note No. 3). Iowa City, IA: Center for Advanced Studies in Measurement and Assessment, The University of Iowa. (Available on [www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma))
- von Davier, A. A., Holland, P. W., & Thayer, D. T. (2004). *The kernel method of test equating*. New York: Springer-Verlag.
- Dennis, J.E., & Schnabel, R.B. (1996). *Numerical methods for unconstrained optimization and nonlinear equations*. Philadelphia, PA: Society for Industrial and Applied Mathematics.
- Haebara, T. (1980). Equating logistic ability scales by a weighted least squares method. *Japanese Psychological Research*, 22, 144–149.
- Hambleton, R.K., & Swaminathan, H. (1985). *Item response theory: Principles and applications*. Boston: Kluwer.
- Hanson, B. A. (1991a). A note on Levine's formula for equating unequally reliable tests using data from the common-item nonequivalent groups design. *Journal of Educational Statistics*, 16, 93–100.
- Hanson, B. A. (1991b). *Method of moments estimates for the four-parameter beta compound binomial model and the calculation of classification consistency indexes* (ACT Research Report 91-5). Iowa City, IA: ACT Inc.
- Hanson, B. A., & Brennan, R. L. (1990) An investigation of classification consistency indexes estimated under alternative strong true score models. *Journal of Educational Measurement*, 27 345–359.
- Holland, P. W., King, B. F., & Thayer, D. T. (1989). *The standard error of equating for the kernel method of equating score distributions* (Technical Report 89-83). Princeton, NJ: Educational Testing Service.
- Holland, P. W., & Thayer, D. T. (1987). *Notes on the use of log-linear models for fitting discrete probability distributions* (Technical Report 87-79). Princeton, NJ: Educational Testing Service.

- Holland, P. W., & Thayer, D. T. (1989). *The kernel method of equating score distributions* (Technical Report 89-84). Princeton, NJ: Educational Testing Service.
- Holland, P. W., & Thayer, D. T. (2000). Univariate and bivariate log-linear models for discrete test score distributions. *Journal of Educational and Behavioral Statistics*, 25, 133–183.
- Huynh, H. (1976). On the reliability of decisions in domain-referenced testing. *Journal of Educational Measurement*, 13, 253–264.
- Keats, J. A., & Lord, F. M. (1962). A theoretical distribution for mental test scores. *Psychometrika*, 27, 59–72.
- Kendall, M., & Stuart, A. (1977). *The advanced theory of statistics* (4th ed., Vol. 1). New York: Macmillan.
- Kernighan, B. W., & Ritchie, D. (1988). *The C programming language* (2nd. ed.). Englewood Cliffs, NJ: Prentice Hall.
- Kim, J., & Hanson, B.A. (2002). Test equating under the multiple-choice model. *Applied Psychological Measurement*, 26, 255–270.
- Kim, S., & Kolen, M.J. (2005). *Methods for obtaining a common scale under unidimensional IRT models: A technical review and further extension* (Iowa Testing Programs Occasional Paper No. 52). Iowa City, IA: Iowa Testing Programs, University of Iowa.
- Kim, S., & Kolen, M.J. (2007). Effects on scale linking of different definitions of criterion functions for the IRT characteristic curve methods. *Journal of Educational and Behavioral Statistics*, 32, 371–397.
- Kim, S., & Lee, W. (2006). An extension of four IRT linking methods for mixed-format tests. *Journal of Educational Measurement*, 43, 53–76.
- Lord, F. M. (1965). A strong true-score theory, with applications. *Psychometrika*, 30, 239–270.
- Lord, F.M. (1980). *Applications of item response theory to practical testing problems*. Hillsdale, NJ: Erlbaum.
- Lord, F. M., & Wingersky, M. S. (1984). Comparison of IRT true-score and equipercentile observed-score "equatings." *Applied Measurement in Education*, 8, 452–461.

- Loyd, B.H., & Hoover, H.D. (1980). Vertical equating using the Rasch model. *Journal of Educational Measurement*, 17, 179–193.
- Kolen, M. J., & Brennan, R. L. (2004). *Test equating, scaling, and linking: Methods and practices* (2nd ed.). New York: Springer-Verlag.
- Marco, G.L. (1977). Item characteristic curve solutions to three intractable testing problems. *Journal of Educational Measurement*, 14, 139–160.
- Masters, G.N. (1982). A Rasch model for partial credit scoring. *Psychometrika*, 47, 149–174.
- Muraki, E. (1992). A generalized partial credit model: Application of an EM algorithm. *Applied Psychological Measurement*, 16, 159–176.
- Muraki, E. (1997). A generalized partial credit model. In W.J. van der Linden and R.K. Hambleton (Eds.), *Handbook of modern item response theory* (pp. 153–164). New York: Springer-Verlag.
- Ogasawara, H. (2001). Standard errors of item response theory equating/linking by response function methods. *Applied Psychological Measurement*, 25, 53–67.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). *Numerical recipes in C: The art of scientific computing* (2nd ed.). New York: Cambridge University Press.
- Reinsch, C. (1967) Smoothing by spline functions. *Numerische Mathematik*, 10, 177–183.
- Rosenbaum, P. R., & Thayer, D. T. (1987). Smoothing the joint and marginal distributions of scored two-way contingency tables in test equating. *British Journal of Mathematical and Statistical Psychology*, 40, 43–49.
- Samejima, F. (1969). Estimation of a latent ability using a response pattern of graded scores. *Psychometrika Monograph Supplement*, 17.
- Samejima, F. (1997). Graded response model. In W.J. van der Linden and R.K. Hambleton (Eds.), *Handbook of modern item response theory* (pp. 85–100). New York: Springer- Verlag.
- Schönberg, I. J. (1964). Spline functions and the problem of graduation. *Proceedings of the National Academy of Sciences of the United States*, 52, 947–950.



- Stocking, M. L., & Lord, R. M. (1983). Developing a common metric in item response theory. *Applied Psychological Measurement*, 7, 201–210.
- Wang, T. (2007, July). *An alternative continuization method to the kernel method in von Davier, Holland, and Thayer's (2004) test equating framework*. (CASMA Research Report No. 11). Iowa City, IA: Center for Advanced Studies in Measurement and Assessment, The University of Iowa. (Available on [www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma))
- Wang, T. (2008). The continuized log-linear method: An alternative to the kernel method of continuization in test equating. *Applied Psychological Measurement*, 32, 527–542.
- Wang, T., & Brennan, R. L. (2006, August). *A modified frequency estimation equating method for the common-item non-equivalent groups design*. (CASMA Research Report No. 19). Iowa City, IA: Center for Advanced Studies in Measurement and Assessment, The University of Iowa. (Available on [www.education.uiowa.edu/casma](http://www.education.uiowa.edu/casma))
- Wang, T., & Brennan, R. L. (2009). A modified frequency estimation equating method for the common-item non-equivalent groups design. *Applied Psychological Measurement*, 33, 118–132.



# Index of Wrapper and Print Functions

<code>Print_BB()</code> , 96	<code>Wrapper_Bootstrap()</code> , 82
<code>Print_BLL()</code> , 120	<code>Wrapper_CC()</code> , 211
<code>Print_Boot_se_eraw()</code> , 85	<code>Wrapper_CK()</code> , 178
<code>Print_Boot_se_ess()</code> , 85	<code>Wrapper_CL()</code> , 123
<code>Print_CC()</code> , 212	<code>Wrapper_CN()</code> , 50
<code>Print_CK()</code> , 179	<code>Wrapper_ESS()</code> , 23
<code>Print_CL()</code> , 124	<code>Wrapper_IRTeq()</code> , 264
<code>Print_CN()</code> , 53, 68	<code>Wrapper_IRTst()</code> , 237
<code>Print_CubSpl()</code> , 159	<code>Wrapper_RB()</code> , 97
<code>Print_ESS()</code> , 25	<code>Wrapper_RC()</code> , 207
<code>Print_ESS_QD()</code> , 266	<code>Wrapper_RK()</code> , 174
<code>Print_IRTeq()</code> , 266	<code>Wrapper_RL()</code> , 121
<code>Print_RB()</code> , 98	<code>Wrapper_RN()</code> , 38
<code>Print_RC()</code> , 208	<code>Wrapper_SC()</code> , 209
<code>Print_RK()</code> , 175	<code>Wrapper_SK()</code> , 176
<code>Print_RL()</code> , 121	<code>Wrapper_SL()</code> , 122
<code>Print_RN()</code> , 38, 58	<code>Wrapper_Smooth_BB()</code> , 96
<code>Print_SC()</code> , 210	<code>Wrapper_Smooth_BLL()</code> , 119
<code>Print_SK()</code> , 177	<code>Wrapper_Smooth_CubSpl()</code> , 157
<code>Print_SL()</code> , 122	<code>Wrapper_Smooth_ULL()</code> , 118
<code>Print_SN()</code> , 39, 58	<code>Wrapper_SN()</code> , 39
<code>Print_SynDens()</code> , 69	
<code>Print_ULL()</code> , 119	

See Table 1.1 on page 8 for a logical grouping of `Wrapper` and `Print` functions along with their page numbers.