バージョン11

スクリプトガイド

第2版

「真の発見の旅とは、新しい風景を探すことではなく、新たな視点を持つことである。」

マルセル・プルースト

JMP, A Business Unit of SAS
SAS Campus Drive
Cary, NC 27513

11.2

このマニュアルを引用する場合は、次の正式表記を使用してください: SAS Institute Inc. 2014. 『JMP® 11 スクリプトガイド　第２版』Cary, NC: SAS Institute Inc.

JMP® 11 スクリプトガイド　第２版

ISBN 978-1-61290-764-2

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

2014 年 7 月、第 1 刷

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at support.sas.com/publishing or call 1-800-727-3228.

技術ライセンスに関する通知

- Scintilla - Copyright © 1998-2012 by Neil Hodgson <neilh@scintilla.org>.

  All Rights Reserved.

  Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

  NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

- Telerik RadControls: Copyright © 2002-2012, Telerik. Usage of the included Telerik RadControls outside of JMP is not permitted.
- ZLIB Compression Library - Copyright © 1995-2005, Jean-Loup Gailly and Mark Adler.
- Made with Natural Earth. Free vector and raster map data @ naturalearthdata.com.
- Packages - Copyright © 2009-2010, Stéphane Sudre (s.sudre.free.fr). All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  Neither the name of the WhiteBox nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS

OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- iODBC software - Copyright © 1995-2006, OpenLink Software Inc and Ke Jin (www.iodbc.org). All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  – Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  – Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  – Neither the name of OpenLink Software Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL OPENLINK OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- bzip2, the associated library "libbzip2", and all documentation, are Copyright © 1996-2010, Julian R Seward. All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- R software is Copyright © 1999-2012, R Foundation for Statistical Computing.
- MATLAB software is Copyright © 1984-2012, The MathWorks, Inc. Protected by U.S. and international patents. See www.mathworks.com/patents. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

目次
スクリプトガイド

1 JMPの概要

マニュアルとその他のリソース 19
表記規則 21
JMPのマニュアル 21
JMPドキュメンテーションライブラリ 22
JMPヘルプ 26
JMPを習得するためのその他のリソース 26
チュートリアル 26
サンプルデータテーブル 27
統計用語とJSL用語の習得 27
JMPを使用するためのヒント 27
ツールヒント 27
JMP User Community 28
JMPer Cable 28
JMP関連書籍 28
「JMPスターター」ウィンドウ 28

2 概要

JMPスクリプト言語にようこそ 29
JSLでできること 31
JSL習得のサポート 31
JMPスクリプト言語（JSL）ガイド 31
スクリプトの索引 32
JMPからJSLを学ぶ 33
用語 34
基本的なJSL構文 37

3 始めましょう

JMPによるスクリプトの作成 39
分析レポートのスクリプトを取得する 41
データテーブルのスクリプトを取得する 42
ファイルを読み込むスクリプトを取得する 43
スクリプトを1つにまとめる 44

4 スクリプト作成のツール

スクリプトエディタ、ログウィンドウ、デバッガ、プロファイルの使用 49
スクリプトエディタの使用 51
スクリプトの実行 51
カラーコーディング 51
オートコンプリート機能 52
ツールヒント 52
ウィンドウの分割 53
括弧の自動マッチ 54
四角形のテキストブロックの選択 55
テキストのドラッグ&ドロップ 56
検索／置換 56
自動フォーマット 56
コード折りたたみマーカーの追加 57
詳細オプション 58
スクリプトエディタの環境設定 59
ログの使用 62
スクリプトウィンドウ内にログを表示 63
ログの保存 63
スクリプトのデバッグ／プロファイル 63
デバッガとプロファイルのウィンドウ 64
ブレークポイントの操作 67
変数の確認 70
ウォッチの操作 70
デバッガでの環境設定の変更 71
デバッガセッションの持続性 71
スクリプトのデバッグとプロファイルの例 72

5 JSLの構成要素

JSLの基礎を学ぶ 79
JSLの構文規則 81
値の区切り文字 81
数 84
名前 84
コメント 85
演算子 86
グローバル変数とローカル変数 89
ローカル名前空間 90
名前付き名前空間 90
Show Symbols、Clear Symbols、Delete Symbols 90
シンボルのロックおよびロック解除 91
名前解決のルール 92
適用範囲が指定されていない名前の解決 92
変数と列名のトラブルシューティング 96
変数とキーワードのトラブルシューティング 97
式を結合する他の方法 98
反復 99
For 99
While 100
Summation 101
Product 102
BreakおよびContinue 102
条件付き関数 104
条件 104
Match 106
Choose 107
Interpolate 108
Step 109
不完全または一致しないデータの比較 109
問い合わせ関数 112

6 データタイプ

数、文字列、日付、通貨、その他の使用 117
数および文字列 119
Unicode 文字 119
パス変数 120
パス変数の作成とカスタマイズ 122
相対パス 122
ファイルパスの区切り 123
日付時間の関数と形式 123
日付時間値 123
日付時間関数を使用したプログラム 124
データテーブル内の日付時間値 130
通貨 133
16 進数の関数と BLOB 関数 134
文字関数の使用 137
Concat 137
Munger 138
Repeat 139
パターンマッチと正規表現の使用 140

7 データ構造

さまざまなデータの処理 145
リスト 147
リストの評価 147
リストを使った割り当て 148
リスト内の処理の実行 148
リスト内の項目の数を求める 148
添え字 148
リスト内で項目を検索する 149
リスト演算子 151
リスト内での反復 153
リストの連結 154
行列 154
行列の作成 155
添え字 156

問い合わせ関数 159
比較演算子、範囲チェック演算子、論理演算子 160
数値演算子 160
結合 163
Transpose（転置） 164
行列とデータテーブル 164
行列とレポート 167
Loc関数 167
順位づけと並べ替え 169
特殊な行列 170
逆行列と連立一次方程式 174
分解と正規化 178
ユーザ定義の行列演算子の作成 182
統計処理の例 183
連想配列 187

8 プログラミング手法

複雑なスクリプト技術とその他の関数 201
リストと式 203
保存された式 203
マクロ 212
リストの操作 212
式の操作 215
高度な適用範囲指定と名前空間 218
Names Default To Here 219
適用範囲が指定された名前 221
名前空間 225
名前空間とスコープの参照 230
名前付き変数参照の解決 233
高度なスクリプトを作成する際のベストプラクティス 235
高度なプログラミングの概念 235
例外のスローとキャッチ 235
Function（関数） 237
Recurse（再帰） 238
Include 239
テキストファイルのロードと保存 239

BY グループを使ったスクリプト 240
ファイルをプロジェクトにまとめる 240
スクリプトの暗号化と暗号解読 243
その他の数値演算子 246
微分 246
代数的な処理 248
最大化と最小化 249
スクリプト実行のスケジューリング 250
メッセージを出力する関数 252
ログへの書き込みを行う 252
ユーザに情報を送る 253

9 データテーブル

データテーブルオブジェクトの操作 257
はじめに 259
データテーブルのスクリプトの基本 261
データテーブルを開く 261
新しいデータテーブルの作成 263
データの読み込み 264
現在のデータテーブルの設定 271
データテーブルに名前をつける 271
データテーブルの保存 272
データテーブルを非表示にする 273
データテーブルの高度なスクリプト 277
列 294
行 312
データ値へのアクセス 337
データテーブルへのメタデータの追加 339
計算 343

10 プラットフォームのスクリプト

分析の作成、反復、変更 347
概要 349
分析プラットフォームのスクリプト 350
インタラクティブなプラットフォームの起動とそのスクリプトの取得 351

プラットフォームの起動 351
スクリプトの保存 351
変更を加える 352
プラットフォームスクリプトの構文 353
By グループレポート 353
By グループスクリプトの保存 356
現在行われている分析にスクリプトコマンドを送る 357
コマンドと引数の規則 357
複数のメッセージを送る 358
オブジェクトが対応するメッセージ 359
Show Properties が戻すリストの読み方 359
プラットフォームの起動 361
列の指定 361
プラットフォームの Action コマンド 361
非表示のレポート 362
タイトル 362
プラットフォームウィンドウの一般メッセージ 363
追記 367
スプライン曲線 367
モデルのあてはめの効果 367
モデルのあてはめに送るコマンド 369
DOE（実験計画）のスクリプト 369
三次元散布図のスクリプト 371
管理図 371

11 表示ツリー

ウィンドウの作成と使用 377
表示の操作 379
ディスプレイボックスの紹介 379
ディスプレイボックスオブジェクトの参照 384
メッセージを送る 387
ビルトインウィンドウにアクセスするには 397
選択ウィンドウの使用 397
ディレクトリ内のファイル 398
表示ツリーの作成 399
基本事項 400

既存の表示の更新 401
インタラクティブな表示要素 404
モーダルウィンドウと非モーダルウィンドウ 409
作成した表示にメッセージを送る 425
独自の表示を始めから作成する 426
プラットフォームを含むディスプレイボックスの作成 427
カスタムプラットフォームの作成 429
シート 432
ジャーナル 434
Picture表示タイプ 435
モーダルウィンドウ 435
モーダルウィンドウの作成 436
汎用モーダルウィンドウ 436
廃止されたDialogをNew Windowに変換する 437
DialogとNew Windowの違い 442
ダイアログおよび列ダイアログの作成 448
スクリプトエディタのスクリプト 451
構文リファレンス 452

12 スクリプトによるグラフ作成
2次元プロットの作成と編集 461
グラフへのスクリプトの追加 463
JSLを使ったグラフィック要素の並べ替え 464
グラフへの凡例の追加 465
独自のグラフを始めから作成する 465
グラフの変更 466
グラフの要素 468
プロット用の関数 468
グラフフレームのプロパティの取得 473
凡例の追加 473
線、矢印、点、図形の描画 474
線 474
矢印 476
マーカー 477
扇形と円弧 479

一般的な図形：円、長方形、楕円 480
その他の図形：多角形と等高線 483
テキストの追加 486
色 487
透明度 488
塗りつぶしのパターン 489
線の種類 489
ピクセルを使った描画 490
インタラクティブなグラフ 491
Handle 492
MouseTrap 495
Drag 関数 496
トラブルシューティング 498

13 3D シーン
3D シーンのスクリプト 499
JSL 3D シーンについて 501
JSL 3D シーンボックス 501
表示領域の設定 504
透視投影シーンのセットアップ 505
平行投影シーンのセットアップ 506
ビューの変更 507
Translate コマンド 507
Rotate コマンド 507
Look At コマンド 509
天体球（アークボール） 510
グラフィックの基本要素 511
基本要素の例 513
基本要素の外観の制御 515
Begin および End のその他の用法 520
球、円柱、円盤の描画 520
テキストの描画 522
行列スタックの使用 523
照明と法線 526
光源の作成 526

照明モデル ... 528
法線ベクトル ... 528
シェーディングモデル ... 529
材質プロパティ ... 530
アルファブレンド ... 530
霧 ... 531
例 ... 531
ベジェ曲線 ... 532
マウスの使用 ... 535
引数 ... 537

14 JMP の拡張
外部データソース、分析ツール、オートメーション ... 539
リアルタイムのデータ取得 ... 541
データフィードオブジェクトの作成 ... 541
リアルタイムデータの読み込み ... 542
メッセージ付きデータフィードの制御 ... 543
ダイナミックリンクライブラリ（DLL） ... 547
JSL でのソケットの使用 ... 550
データベースアクセス ... 553
SAS の使用 ... 556
SAS DATA ステップの作成 ... 556
計算式列の SAS データステップコードの作成 ... 556
SAS 変数名 ... 557
SAS マクロ変数の値の取得 ... 557
SAS Metadata Server への接続 ... 558
環境設定 ... 561
サンプルスクリプト ... 561
MATLAB の使用 ... 562
MATLAB のインストール ... 563
R の操作 ... 565
R のインストール ... 565
JMP から R へのインターフェース ... 567
R の JSL スクリプト可能なオブジェクトインターフェース ... 567
JMP データタイプと R データタイプの相互変換 ... 567

トラブルシューティング 570
例 571
Excelの使用 572
OLEオートメーション 573
Visual Basicを使ったJMPのオートメーション 573
Visual C++を使ったJMPのオートメーション 581

15 アプリケーションの作成と共有
アプリケーションビルダーとアドインビルダー 585
アプリケーションビルダー 587
例 587
アプリケーションビルダーの用語 589
アプリケーションの設計 591
「アプリケーションビルダー」ウィンドウ 591
赤い三角ボタンのオプション 592
アプリケーションの作成 594
アプリケーションの編集または実行 605
アプリケーションの保存オプション 606
JMPアドイン 609
アドインビルダーを使ったアドインの作成 609
アドインの編集 613
［アドイン］メニューからのアドインの削除 614
アドインのアンインストール 614
アドインの共有 614
JSLを使ったアドインの登録 615
手動でのアドインの作成 615

16 プログラム例の紹介
サンプルによるプログラミングの学習 619
起動時のスクリプトの実行 621
文字の日付を数値の日付に変換 621
日付によるデータ抽出 623
計算式を含んだ列の作成 624
分析結果の一部を抜き出す 626
対話型プログラムの作成 628

A 互換性に関するメモ
JMP 11 における JMP 10 からの変更点 631
B 用語集
用語、概念、表記 639
索引
スクリプトガイド 643

第 1 章

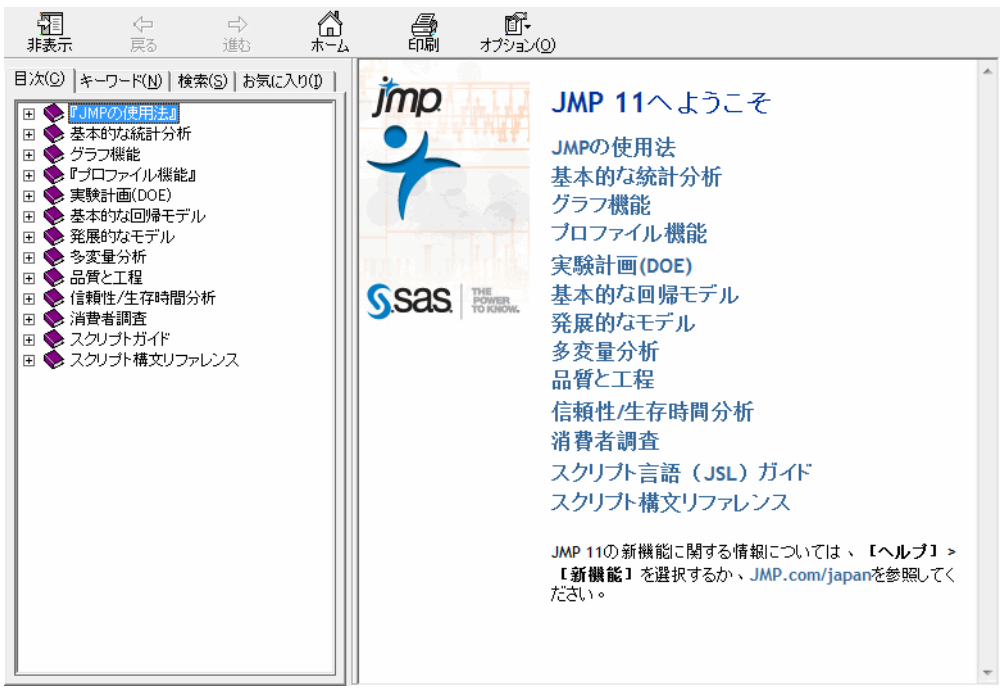
JMP の概要

マニュアルとその他のリソース

この章には以下の情報が記載されています。

- 本書の表記法
- JMP のマニュアル
- JMP ヘルプ
- その他のリソース
  - その他の JMP のドキュメンテーション
  - チュートリアル
  - 索引
  - Web リソース

図 1.1 JMP ヘルプのホームウィンドウ（Windows）



目次

表記規則 21
JMP のマニュアル 21
JMP ドキュメンテーションライブラリ 22
JMP ヘルプ 26
JMP を習得するためのその他のリソース 26
チュートリアル 26
サンプルデータテーブル 27
統計用語と JSL 用語の習得 27
JMP を使用するためのヒント 27
ツールヒント 27
JMP User Community 28
JMPer Cable 28
JMP 関連書籍 28
「JMP スターター」ウィンドウ 28

表記規則

マニュアルの内容と画面に表示される情報を対応付けるために、次の表記規則を使っています。

- サンプルデータ名、列名、パス名、ファイル名、ファイル拡張子、およびフォルダ名は「」で囲んで表記しています。
- スクリプトのコードは`Lucida Sans Typewriter`フォントで表記しています。
- スクリプトコードの結果（ログに表示されるもの）は*`Lucida Sans Typewriter`*（斜体）フォントで表記し、先に示すコードよりインデントされています。
- クリックまたは選択する項目は［］で囲んで太字で表記しています。これには以下の項目があります。
  - ボタン
  - チェックボックス
  - コマンド
  - 選択可能なリスト項目
  - メニュー
  - オプション
  - タブ名
  - テキストボックス
- 次の項目は太字で表記しています。
  - 重要な単語や句、JMPに固有の定義を持つ単語や句
  - マニュアルのタイトル
  - 変数名
- JMP Proのみの機能にはJMP Proアイコン JMP PRO がついています。JMP Proの機能の概要については http://www.jmp.com/software/pro/ をご覧ください。

注：特別な情報および制限事項には、この文のように「注:」という見出しがついています。

ヒント：役に立つ情報には「ヒント」という見出しがついています。

JMPのマニュアル

JMPには、印刷版、PDF版、電子本など、さまざまな形式のマニュアルが用意されています。

- PDF版は［**ヘルプ**］>［**ドキュメンテーション**］メニューまたはJMPオンラインヘルプのフッタから開くことができます。

- 検索しやすいようにすべてのドキュメンテーションが 1 つの PDF ファイルにまとめられた『JMP ドキュメンテーションライブラリ』と呼ばれるファイルがあります。『JMP ドキュメンテーションライブラリ』の PDF ファイルは［**ヘルプ**］>［**ドキュメンテーション**］メニューから開くことができます。
- 電子本は Amazon、Safari Books Online、および Apple iBookstore でお求めになれます。

JMP ドキュメンテーションライブラリ

以下の表は、JMP ライブラリに含まれている各ドキュメンテーションの目的および内容をまとめたものです。

マニュアル	目的	内容
『はじめての JMP』	JMP をあまりご存知ない方を対象とした入門ガイド	JMP の紹介と、データを作成および分析し始めるための情報
『JMP の使用法』	JMP のデータテーブルと、基本操作を理解する	一般的な JMP の概念と、データの読み込み、列プロパティの変更、データの並べ替え、SAS への接続など、JMP 全体にわたる機能の説明
『基本的な統計分析』	このマニュアルを見ながら、基本的な分析を行う	［分析］メニューからアクセスできる以下のプラットフォームの説明： • 一変量の分布 • 二変量の関係 • 対応のあるペア • 表の作成 ブートストラップを使用した標本分布の近似方法も含まれています。

マニュアル	目的	内容
『グラフ機能』	データに合った理想的なグラフを見つける	［グラフ］メニューからアクセスできる以下のプラットフォームの説明： • グラフビルダー • 重ね合わせプロット • 三次元散布図 • 等高線図 • バブルプロット • パラレルプロット • セルプロット • ツリーマップ • 散布図行列 • 三角図 • チャート 背景マップやカスタムマップの作成方法も記載されています。
『プロファイル機能』	対話式のプロファイルツールの使い方を学ぶ。任意の応答曲面の断面を表示できるようになります。	［グラフ］メニューに表示されるすべてのプロファイルについて。誤差因子の分析が、ランダム入力を使用したシミュレーションの実行とともに含まれています。
『実験計画 (DOE)』	実験の計画方法と適切な標本サイズの決定方法を学ぶ	**［実験計画（DOE）］**メニューのすべてのトピックについて。
『基本的な回帰モデル』	「モデルのあてはめ」プラットフォームとその多くの手法について学ぶ	［分析］メニューの「モデルのあてはめ」プラットフォームで使用できる、以下の手法の説明： • 標準最小2乗 • ステップワイズ • 正則化回帰 • 混合モデル • MANOVA • 対数線形-分散 • 名義ロジスティック • 順序ロジスティック • 一般化線形モデル

マニュアル	目的	内容
『発展的なモデル』	付加的なモデリング手法について学ぶ	［分析］>［モデリング］メニューで使用できる以下のプラットフォームの説明： • パーティション • ニューラル • モデルの比較 • 非線形回帰 • Gauss 過程 • 時系列分析 • 応答スクリーニング ［分析］>［モデリング］メニューの「スクリーニング」プラットフォームについては『実験計画 (DOE)』で説明しています。
『多変量分析』	複数の変数を同時に分析するための手法について理解を深める	［分析］>［多変量］メニューで使用できる以下のプラットフォームの説明： • 多変量の相関 • クラスター分析 • 主成分分析 • 判別分析 • PLS
『品質と工程』	工程を評価し、向上させるためのツールについて理解を深める	［分析］>［品質と工程］メニューで使用できる以下のプラットフォームの説明： • 管理図ビルダーと個々の管理図 • 測定システム分析 • 変動性図/計数値用ゲージチャート • 工程能力 • パレート図 • 特性要因図

マニュアル	目的	内容
『信頼性/生存時間分析』	製品やシステムにおける信頼性を評価し、向上させる方法、および人や製品の生存時間データを分析する方法について学ぶ	［分析］>［信頼性/生存時間分析］メニューで使用できる以下のプラットフォームの説明： • 寿命の一変量 • 寿命の二変量 • 再生モデルによる分析 • 劣化分析 • 信頼性予測 • 信頼性成長 • 信頼性ブロック図 • 生存時間分析 • 生存時間(パラメトリック)のあてはめ • 比例ハザードのあてはめ
『消費者調査』	消費者選好を調査し、その洞察を使用してより良い製品やサービスを作成するための方法を学ぶ	［分析］>［消費者調査］メニューで使用できる以下のプラットフォームの説明： • カテゴリカル • 因子分析 • 選択モデル • アップリフト • 項目分析
『スクリプトガイド』	パワフルなJMPスクリプト言語（JSL）の活用方法について学ぶ	スクリプトの作成やデバッグ、データテーブルの操作、ディスプレイボックスの構築、JMPアプリケーションの作成など。
『スクリプト構文リファレンス』	JSL関数、その引数、およびオブジェクトやディスプレイボックスに送信するメッセージについて理解を深める	JSLコマンドの構文、例、および注意書き。

注：［ドキュメンテーション］メニューでは、印刷可能な2つのリファレンスカードも用意されています。『メニューカード』はJMPのメニューをまとめた表で、『クイックリファレンス』はJMPのショートカットキーをまとめた表です。

JMPヘルプ

JMPヘルプは、一連のマニュアルの簡易版です。JMPのヘルプは、次のいくつかの方法で開くことができます。

- Windowsでは、F1キーを押すとヘルプシステムウィンドウが開きます。
- データテーブルまたはレポートウィンドウの特定の部分のヘルプを表示します。［**ツール**］メニューからヘルプツール を選択した後、データテーブルやレポートウィンドウの任意の位置でクリックすると、その部分に関するヘルプが表示されます。
- JMPウィンドウ内で［**ヘルプ**］ボタンをクリックします。
- Windowsの場合、［**ヘルプ**］メニューの［**ヘルプの目次**］、［**ヘルプの検索**］、［**ヘルプの索引**］の各オプションを使用して、JMPヘルプ内を検索し、目的の内容を表示します。Macの場合、［**ヘルプ**］>［**JMPヘルプ**］を選択します。
- http://jmp.com/support/help/ でヘルプを検索します（英語のみ）。

JMPを習得するためのその他のリソース

JMPのマニュアルとJMPヘルプの他、次のリソースもJMPの学習に役立ちます。

- チュートリアル（「チュートリアル」（26ページ）を参照）
- サンプルデータ（「サンプルデータテーブル」（27ページ）を参照）
- 索引（「統計用語とJSL用語の習得」（27ページ）を参照）
- 使い方ヒント（「JMPを使用するためのヒント」（27ページ）を参照）
- Webリソース（「JMP User Community」（28ページ）を参照）
- 専門誌『JMPer Cable』（「JMPer Cable」（28ページ）を参照）
- JMPに関する書籍（「JMP関連書籍」（28ページ）を参照）
- JMPスターター（「「JMPスターター」ウィンドウ」（28ページ）を参照）

チュートリアル

［**ヘルプ**］>［**チュートリアル**］を選択して、JMPのチュートリアルを表示できます。［**チュートリアル**］メニューの最初の項目は［**チュートリアルディレクトリ**］です。この項目を選択すると、すべてのチュートリアルをカテゴリ別に整理した新しいウィンドウが開きます。

JMPに慣れていない方は、まず［**初心者用チュートリアル**］を試してみてください。JMPのインターフェースおよび基本的な使用方法を学ぶことができます。

他のチュートリアルでは、円グラフの作成、グラフビルダーの使用など、JMPの具体的な活用法を学習できます。

サンプルデータテーブル

JMPのマニュアルで取り上げる例は、すべてサンプルデータを使用しています。次の操作はすべて［**ヘルプ**］>［**サンプルデータ**］で表示されるウィンドウで行えます。

- サンプルデータディレクトリを開く。
- すべてのサンプルデータテーブルを文字コード順に並べた一覧を表示する。
- カテゴリ別に整理されたリストからサンプルデータテーブルを見つける。

サンプルデータテーブルは次のディレクトリにインストールされています。

Windowsの場合: C:¥Program Files¥SAS¥JMP¥<バージョン番号>¥Samples¥Data

Macintoshの場合: \Library\Application Support\JMP\<バージョン番号>\Samples\Data

JMP Proでは、サンプルデータが（JMPではなく）JMPPROディレクトリにインストールされています。

統計用語とJSL用語の習得

［**ヘルプ**］メニューには、次の索引が用意されています。

統計の索引　統計用語が説明されています。

スクリプトの索引　JSL関数、オブジェクト、ディスプレイボックスに関する情報を検索できます。スクリプトの索引からサンプルスクリプトを編集して実行することもできます。

JMPを使用するためのヒント

JMPを最初に起動すると、「使い方ヒント」ウィンドウが表示されます。このウィンドウには、JMPを使う上でのヒントが表示されます。

「使い方ヒント」ウィンドウを表示しないようにするには、［**起動時にヒントを表示する**］のチェックを外します。再表示するには、［**ヘルプ**］>［**使い方ヒント**］を選択します。または、「環境設定」ウィンドウで非表示に設定することもできます。詳細については、『JMPの使用法』を参照してください。

ツールヒント

次のような項目の上にカーソルを置くと、その項目を説明するツールヒントが表示されます。

- メニューまたはツールバーのオプション
- グラフ内のラベル
- レポートウィンドウ内の結果（テキスト）（カーソルで円を描くと表示される）
- 「ホームウィンドウ」内のファイル名またはウィンドウ名
- スクリプトエディタ内のコード

ヒント：JMP 環境設定で、ツールヒントを表示しないよう設定できます。［**ファイル**］>［**環境設定**］>［**一般**］（Macintoshの場合は［**JMP**］>［**環境設定**］>［**一般**］）を選択し、［**メニューのヒントを表示**］のチェックを外します。

JMP User Community

JMP User Communityでは、さまざまな方法でJMPをさらに習得したり、他のSASユーザとのコミュニケーションを図ったりできます。ラーニングライブラリには1ページ構成のガイド、チュートリアル、デモなどが用意されており、JMPを使い始める上でとても便利です。また、JMPのさまざまなトレーニングコースに登録して、自己教育を進めることも可能です。

その他のリソースとして、ディスカッションフォーラム、サンプルデータやスクリプトファイルの交換、Webcastセミナー、ソーシャルネットワークグループなども利用できます。

WebサイトのJMPリソースにアクセスするには［**ヘルプ**］>［**JMP User Community**］を選択します。

JMPer Cable

JMPer Cableは、JMPユーザを対象とした年刊の専門誌です。JMPer Cableは次のJMP Webサイトで閲覧可能です。

http://www.jmp.com/about/newsletters/jmpercable/（英語）

JMP関連書籍

JMP関連書籍は、次のJMP Webページで紹介されています。

http://www.jmp.com/japan/academic/books.shtml

「JMPスターター」ウィンドウ

JMPまたはデータ分析にあまり慣れていないユーザは、「JMPスターター」ウィンドウから開始するとよいでしょう。カテゴリ分けされた項目には説明がついており、ボタンをクリックするだけで該当の機能を起動できます。「JMPスターター」ウィンドウには、［**分析**］、［**グラフ**］、［**テーブル**］、および［**ファイル**］メニュー内の多くのオプションがあります。

- 「JMPスターター」ウィンドウを開くには、［**表示**］（Macintoshでは［**ウィンドウ**］）>［**JMPスターター**］を選択します。
- WindowsでJMPの起動時に自動的に「JMPスターター」を表示するには、［**ファイル**］>［**環境設定**］>［**一般**］を選び、「開始時のJMPウィンドウ」リストから［**JMPスターター**］を選択します。Macintoshでは、［**JMP**］>［**環境設定**］>［**起動時にJMPスターターウィンドウを表示する**］を選択します。

第 2 章

概要
JMP スクリプト言語にようこそ

JMP スクリプト言語（JSL）でスクリプトを記述すると、JMP の分析結果を再現することができます。パワーユーザの多くは、JMP の機能を拡張するスクリプトを作成し、製造に関する設定など定期的に行う分析を自動化しています。JSL の習得が面倒な人でも、スクリプトを自動生成する機能を利用できます。

JSL によっていろいろなことを行えます。

- 列の計算式を実行する
- プラットフォームを起動する
- プラットフォームをインタラクティブに変更する
- グラフを作成する

目次

JSL でできること 31
JSL 習得のサポート 31
JMP スクリプト言語（JSL）ガイド 31
スクリプトの索引 32
JMP から JSL を学ぶ 33
用語 34
基本的な JSL 構文 37

JSL でできること

JMP では、データテーブルや分析の現在の状態を再現するスクリプトを自動的に生成し、保存できます。分析の途中でスクリプトをスクリプトウィンドウ（または**スクリプトエディタ**）、データテーブル、または分析レポートに保存しておけば、後で修正して別のプロジェクトに利用できます。分析が終わった時点でスクリプトを保存すれば、最終結果を再現することができます。

以下は、JSL スクリプトが役に立つ例です。

- 分析プロセスを最初から最後まで詳細に記述しなければならない場合。たとえば、管理当局や、学術論文をレビューする人のために、追跡調査に役立つ記録を作成できます。
- 研究所の技術者が、一連の分析を定期的に実施する場合。
- 毎日、新しいデータに同じ手順で同じモデルをあてはめる場合。

JMP を通常どおりインタラクティブに使用して、作業を再現するスクリプトを保存しておけば、スクリプトを実行するだけで結果が再現できるようになります。

JSL は、設計上、次のような目的には使用できません。

- JMP での操作そのものをスクリプトとして記録することはできません。スクリプト言語の中にはスクリプトレコーダーによる記録が便利なものもありますが、JMP のように結果が大切なソフトウェアではあまり重要ではありません。スクリプトレコーダーを使って、インタラクティブな操作が実行される様子を観察することはできません。
- JSL は、JMP を操作するための代替手段として用意されたコマンドラインインターフェースではありません。

JSL 習得のサポート

JMP には、JSL スクリプトを書いたり理解したりする上で役立つツールがいくつか用意されています。

JMP スクリプト言語（JSL）ガイド

『スクリプトガイド』では、スクリプト言語にあまり慣れていない JMP ユーザ向けの基本的な情報（用語や構文など）から、より高度な情報まで説明しています。

第 2 章～第 4 章	JSL の習得、基本的なスクリプトを生成させる方法、また JSL スクリプトを作成する環境についての情報を含む。
第 5 章～第 8 章	JSL の基本要素、数値や文字列などの基本的なデータタイプ、リストや行列、連想配列の記述、名前空間、JSL でのプログラミングの基礎について紹介する。

第9章～第13章	データテーブル、プラットフォーム、ウィンドウ、グラフィックなどのJMPオブジェクトに対するJSLの使用方法を取り上げる。
第14章	SAS、R、Excelなどの外部プログラムと連携するためのスクリプトの記述方法について説明する。
第15章	ボタン、リスト、グラフ、その他のオブジェクトを含むウィンドウをドラッグ&ドロップで作成できる、アプリケーションビルダーでのJMPアプリケーションの作成について紹介する。また、アドインビルダーを使ってスクリプトをコンパイルし、簡単に共有できるファイルを作成する方法についても説明する。
第16章	スクリプトのいろいろなサンプルや例の紹介。コピーして、必要なところだけ変更を加えて利用できる。
付録Aおよび付録B	前バージョンのJMPとの互換性に関する情報を提供。また、JSLの概念と用語について説明する。

スクリプトの索引

［**ヘルプ**］メニューにあるスクリプトの索引には、JSL関数、オブジェクト、およびディスプレイボックスの簡単な説明と構文が記載されています。各エントリに用意されている例を実行したり、修正を加えてコードのテストに利用することができます。例を実行すると、下に埋め込まれたログウィンドウにメッセージが表示されます。

「スクリプトの索引」ウィンドウには、次のボタンがあります。


検索を開始するには、［**検索**］ボタンをクリックします。


［**クリア**］ボタンをクリックすると、検索テキストボックスがクリアされ、新たに検索を開始することができます。


検索方法とパラメータを設定するには、［**設定**］ボタンをクリックします。

［**設定**］ボタンには複数の検索方法が用意されています。

部分一致　少なくとも一部が検索文字列と一致するすべての項目を戻します。たとえば、「leas」で検索すると、「Release Zoom」や「Partial Least Squares」といったメッセージが戻されます。デフォルトでは、この方法により検索されます。

フレーズ検索　検索文字列と完全に一致するテキストを含む項目を戻します。たとえば、「text」で検索すると、「text」という文字列を含むすべての要素が戻されます。

すべての項　いずれかの文字列またはすべての文字列を含む項目を戻します。たとえば、「t test」で検索すると、「Pat Test」、「Shortest Edit Script」、「Paired t test」が戻されます。

いずれかの項　検索文字列のいずれかを含む項目を戻します。たとえば、「text string」で検索すると、「Context Box」、「Drag Text」、「Is String」などが戻されます。

正規表現 検索ボックスにワイルドカード（*）とピリオド（.）を使用できます。たとえば、「get *name」で検索すると、「Get Name Info」、「Get Namespace」などのメッセージが戻されます。「get.*name」で検索すると、「Get Color Theme Names」、「Get Name Info」、「Get Effect Names」などの項目が戻されます。

［**設定**］ボタンにはいくつかの検索パラメータも用意されています。

すべてのフィールド 索引内のすべてのフィールドを対象に検索文字列を検索するよう指定します。

タイトルのみ 索引のタイトルのみを対象に検索文字列を検索するよう指定します。

例のみ 索引の例のみを対象に検索文字列を検索するよう指定します。

例を除く 例以外を対象に検索文字列を検索するよう指定します。

JMP のオンラインヘルプで項目の詳細を確認するには、その項目の［**トピックのヘルプ**］ボタンをクリックします。

JMP から JSL を学ぶ

JMP 自身が、実は最高の JSL プログラマーです。JMP でインタラクティブに作業し、その結果をスクリプトで保存して、後から再使用できます。スクリプトに簡単な編集を加え、テンプレートとして使えば、毎日の作業がスピードアップします。

JSL は非常に柔軟性のある言語で、同じ目的をさまざまな方法で達成できます。以下はその例です。JMP で分析を行うと、大部分の処理をデフォルトで自動的に行った場合でも、分析のあらゆる詳細が保存されます。ということは、ユーザが書くスクリプトも同様に完全で詳細でなければならないのでしょうか。そんなことはありません。通常は、グラフィカルユーザインターフェース（GUI）で選択する項目だけで問題ありません。たとえば、サンプルデータのフォルダにある「Big Class.jmp」を開き、「**身長 (インチ)**」、「**体重 (ポンド)**」、「**性別**」の一変量の分布を起動したいときは、次のようなスクリプトだけで大丈夫です。

```
Distribution( Y (:Name("身長 ( インチ )"), :Name("体重 ( ポンド )"), :性別 ) );
```

GUI で「一変量の分布」プラットフォームを実行し、レポートの赤い三角ボタンのメニューから［**スクリプト**］>［**スクリプトをスクリプトウィンドウに保存**］を選択すると、次のようなスクリプトが表示されます。

```
Distribution(
   Nominal Distribution( Column( :sex ) ),
   Continuous Distribution( Column( :height ) ),
   Continuous Distribution( Column( :weight ) )
);
```

どちらのスクリプトも結果は同じです。

JSL をいろいろと試してみてください。こんなこともできるのでは、と思ったなら、おそらくできるはずです。どうなるか試してみましょう。

用語

スクリプトの作成を始める前に、このマニュアル全体で使用されている基本的なJSL用語を知っておく必要があります。

演算子と関数

演算子は、一般的な演算に使用される1文字または2文字の記号（+や=など）です。

関数はコマンドで、追加情報として引数を指定する場合もあります。

一部のJSL関数は演算子と同様に機能しますが、より複雑なアクションを実行できます。たとえば、次の2つのスクリプトは同じ処理を行います。

```
2 + 3;
Add( 2, 3 );
```

最初の行は+演算子を使用しています。2行目は等価な関数である`Add()`を使用しています。

JSLの演算子のすべてに、等価な関数がありますが、関数の中には等価な演算子がないものもあります。たとえば、`Sqrt(a)`は`Sqrt()`関数でしか表現できません。

注：以前のバージョンのJMPおよびそれらのマニュアルでは、**演算子**と**関数**という2つの用語があまり区別されていませんでした。今バージョンでは、より厳密に区別して記載しています。

オブジェクトとメッセージ

オブジェクトとは、JMPの動的なエンティティで、たとえばデータテーブル、データ列、プラットフォーム結果ウィンドウ、グラフなどのこと。ほとんどのオブジェクトは、何らかのアクションの実行するメッセージを受け取ることができます。

メッセージとは、オブジェクトに送られるJSL式を指します。オブジェクトは、送られたメッセージを評価し、何らかのアクションを実行します。次の例で、`dt`はデータテーブルオブジェクトです。`<<`はメッセージが続くことを意味します。次のメッセージは、指定の変数を使って要約テーブルを作成するよう指示しています。

```
dt << Summary( Group( :年齢 ), Mean( :Name("身長（インチ）") ) )
```

この式で、`dt`はデータテーブルへの参照を含む変数の名前です。この変数には任意の名前を使用できます。このマニュアルでは、データテーブルの参照を`dt`と表記します。以下に、このマニュアルでよく使用されているオブジェクトの表記方法を示します。

略語	オブジェクト
`dt`	データテーブル
`col`	データテーブルの列
`colname`	データテーブルの列の名前

略語	オブジェクト
obj	オブジェクト
db	ディスプレイボックス

これらの変数には参照が事前に割り当てられているわけではありません。どの変数も、使用する前に割り当てておく必要があります。次の例では、A という名前のグローバル変数に“世界のみなさん、こんにちは”という値が割り当てられています。Show(A) コマンドが処理されると、A の値が表示されます。

```
A = " 世界のみなさん、こんにちは ";
Show( A );
   A = " 世界のみなさん、こんにちは ";
```

引数とパラメータ

引数は、関数またはメッセージに送ることのできる追加の情報です。たとえば、Root(25) の場合、25 は Root() 関数に対する引数です。Root() は指定された引数に対して実行され、結果（5）を戻します。

プログラミングやスクリプティングのマニュアルを読むと、パラメータも登場するのが普通です。**パラメータ**は、関数が受け入れる引数についての説明です。たとえば、Root() は一般的に Root(number) のように指定され、*number* がパラメータです。

パラメータと**引数**は、関数が必要とする情報をそれぞれ異なる視点から表現したものです。

このマニュアルでは、簡単にするために両方のケースで**引数**という用語を使用します。

名前付き引数は、事前に決められた引数のセットの中から、引数の名前を記述して明示的に指定するものです。たとえば、Graph Box() 関数内の title(" 折れ線グラフ ") は、タイトルを明示的に定義する名前付き引数です。

```
Graph Box( title(" 折れ線グラフ "),
      Frame Size( 300, 500 ),
      Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
      Pen Color( "Blue" );
      Line( [10 30 70], [88 22 44] ));
```

Frame Size() の引数の 300 と 500 は名前付き引数ではありません。これらの引数は位置に重要な意味があります。最初の引数は幅、2 番目の引数は高さを表します。

オプションの引数

関数とメッセージには、必須の引数とオプションの引数があります。オプションの引数は、指定しなくてもかまいません。表記上、オプションの引数は鍵括弧で囲んであります。たとえば、次のような設定が行えます。

```
Root( x, <n> )
```

で、引数 *x* は必須です。引数 *n* はオプションです。

オプションの引数の多くに、デフォルトの値があります。たとえば Root() の場合、*n* のデフォルト値は 2 です。

コード	出力	説明
Root(25)	5	25 の平方根を戻す。
Root(25, 2)	5	25 の平方根を戻す。
Root(25, 3)	2.92401773821287	25 の三乗根を戻す。

式

式は、タスクを実行する JSL コードの一部分です。JSL 式は、データの保持、データの処理、およびオブジェクトへのコマンドの送信を行います。たとえば、次の式は「Big Class.jmp」サンプルデータテーブルを開き、二変量のグラフを作成します。

```
Open( "$SAMPLE_DATA¥Big Class.JMP" );
   Bivariate( Y( :Name(" 体重 ( ポンド )") ), X( :Name(" 身長 ( インチ )") ) );
```

Or および縦棒の記号

1 本の縦棒（|）は論理 OR を表します。引数を表すとき、**or** を単に | で表記します。

たとえば、パス名を指定する引数として absolute|relative のような表記があった場合、引数には次の 2 つのオプションの**どちらか**を指定できることを示しています。

- absolute は絶対パス名を示します。
- relative は相対パス名を示します。

同様に、縦棒を使って 3 つ以上のオプションを示す場合もあります。

スクリプトのフォーマット

空白（スペース、タブ、改行など）と大文字、小文字の違いは JSL では無視されます。つまり、次の 2 つの式は等価です。

```
// 式 1
sum=0; for(i=1,i<=10,i++,sum+=i;show(i,sum))

// 式 2
Sum = 0;
For( i = 1, i <= 10, i++,
   Sum += i;
   Show( i, Sum );
);
```

スクリプトはお好みのフォーマットで記述できますが、スクリプトエディタを使えば、自動的にフォーマットすることも可能です。このマニュアルでは、大文字／小文字、スペース、改行、タブなどの使い方に、スクリプトエディタのデフォルトのフォーマットを使用します。スクリプトエディタの使用方法については、「スクリプト作成のツール」の章の「スクリプトエディタの使用」（51 ページ）を参照してください。

注：空白に関連する唯一の例外は、2 文字の演算子（<= または ++）です。これらの演算子はスペースで分離することができません。

基本的な JSL 構文

JSL スクリプトは一連の式です。各式は、タスクを実行する JSL コードの一部分です。JSL 式は、データの保持、データの処理、およびオブジェクトへのコマンドの送信を行います。

多くの式は、メッセージ名の後ろに内容が括弧で囲まれた構造になっています。

```
Message Name ( argument 1, argument 2, ... )
```

JSL 名の意味はコンテキストによって異なります。同じ名前でも、データテーブルのコンテキストと、関数のコンテキストでは全く別の意味を持つことがあります。詳細については、「JSL の構成要素」の章の「名前解決のルール」（92 ページ）を参照してください。

括弧の一致など特定の記述ルールに従うものはほとんどが JSL 式として有効です。たとえば、次のような設定が行えます。

```
New Window( " ウィンドウ ",
   << modal,
   Text box( " 世界のみなさん、こんにちは " ),
   Text Box( "-----" ),
   ButtonBox( "OK" )
);
```

上の例で、次の点に注意してください。

- 名前にはスペースを埋め込むことができます。詳細については、「JSL の構成要素」の章の「名前」（84 ページ）を参照してください。
- メッセージの内容は、必ず一致した括弧で囲みます。「JSL の構成要素」の章の「括弧」（81 ページ）を参照してください。
- 項目はカンマで区切ります。「JSL の構成要素」の章の「カンマ」（81 ページ）を参照してください。
- JSL には大文字と小文字の区別がありません。“`text box();`” と “`Text Box()`” は等価です。
- メッセージが別のメッセージの入れ子になるケースも多くあります。

第 3 章

始めましょう
JMPによるスクリプトの作成

同じデータを使い、同じレポートを定期的に作成するケースはよくあります。この章では、テキストデータを読み込む、Excelファイルを開く、レポートを作成するといった一般的なタスクを実行するスクリプトを、JMPで自動生成させる方法を説明します。最後のチュートリアルでは、それらをまとめ、1つのスクリプトでExcelファイルを開き、3つのレポートを自動的に作成できるようにします。

このマニュアルは、JMPの使用には慣れているものの、JSLにはあまり慣れていないユーザを対象としています。一般的なタスクの実行については、『JMPの使用法』を参照してください。また、『はじめてのJMP』も、基本概念やJMPでの操作の流れを理解するのに役立ちます。

目次

分析レポートのスクリプトを取得する 41
データテーブルのスクリプトを取得する 42
ファイルを読み込むスクリプトを取得する 43
スクリプトを1つにまとめる 44

分析レポートのスクリプトを取得する

分析を再現するためのスクリプトは、次のような手順で取得できます。

1. 一変量の分布など、プラットフォームを起動します。
2. 必要に応じて変更を加えます。たとえば、検定や他のグラフを追加します。
3. 結果を再現するスクリプトを生成させます。

スクリプトはデータテーブルに保存できるので、他のユーザにデータテーブルを送るだけで、そのユーザはスクリプトを実行し、レポートを再現することができます。

例

次の手順に従って、一変量の分布のレポートを作成し、レポートを再現するスクリプトを生成させ、スクリプトをデータテーブルに保存してみましょう。

注：例で使用するデータテーブルはJMPの「Samples¥Data」フォルダにあります。

1. 「Companies.jmp」サンプルデータテーブルを開きます。
2. ［**分析**］>［**一変量の分布**］を選択して、「一変量の分布」起動ウィンドウを開きます。
3. 「列の選択」ボックスで**「利益 ($M)」**を選択し、［**Y, 列**］ボタンをクリックします。
4. ［**OK**］をクリックします。

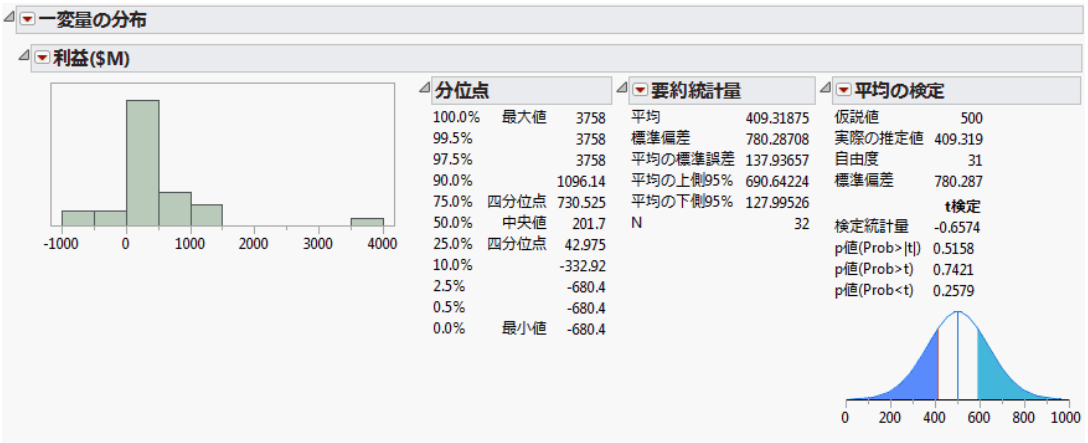
   一変量の分布のレポートウィンドウが表示されます。
5. 「一変量の分布」の赤い三角ボタンのメニューから［**積み重ねて表示**］を選択し、レポートを横に表示します。
6. 「利益 ($M)」の赤い三角ボタンのメニューで［**外れ値の箱ひげ図**］の選択を解除し、このオプションをオフにします。
7. 「利益 ($M)」の赤い三角ボタンのメニューから［**平均の検定**］を選択します。

   「平均の検定」ウィンドウが表示されます。
8. ［**仮説平均を指定**］ボックスに 500 と入力します。
9. ［**OK**］をクリックします。

   レポートウィンドウに平均の検定が追加されます。

   これで、カスタマイズしたレポートが作成できました。

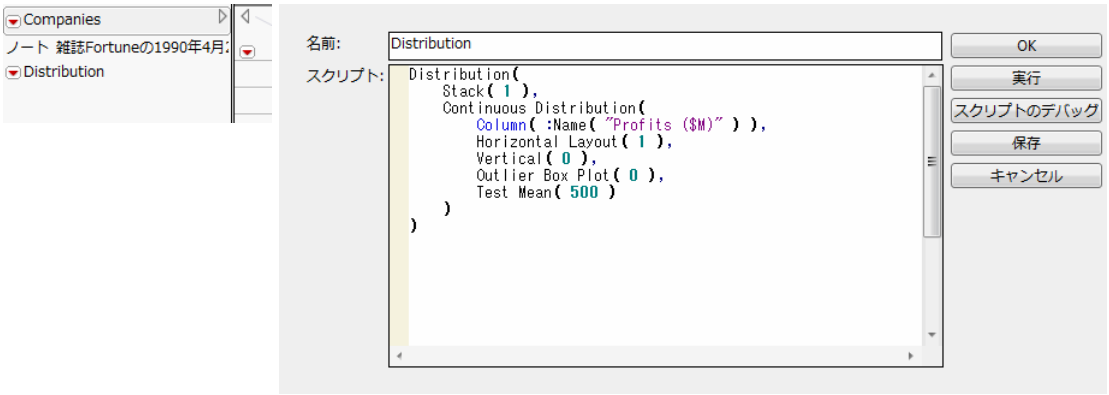
図 3.1 カスタマイズした一変量の分布レポート



10. 「一変量の分布」の赤い三角ボタンのメニューから［**スクリプト**］>［**スクリプトをデータテーブルに保存**］を選択します。

    これで、データテーブルに「Distribution」（一変量の分布）という名前のスクリプトが保存されました。そのスクリプトの赤い三角ボタンのメニューから［**編集**］を選択すると、スクリプトが表示されます。

図 3.2 データテーブルに保存された一変量の分布のスクリプト



11. このスクリプトを実行して、最終的なレポートをそのまま再現するには、スクリプトの赤い三角ボタンのメニューから［**スクリプトの実行**］を選択します。

データテーブルのスクリプトを取得する

データテーブルを再現するためのスクリプトは、次のような手順で取得します。

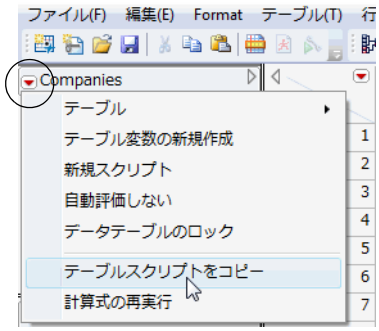
1. データテーブルを開きます。
2. 必要に応じて変更を加えます。たとえば、スクリプトを追加したり、値を修正したり、新しい列を追加したりします。
3. データテーブルを再現するスクリプトを取得します。

例

先ほどの例で取得した、スクリプトを保存したデータテーブルを使用します。

1. データテーブル名の隣にある赤い三角ボタンをクリックします。
2. ［**テーブルスクリプトをコピー**］を選択します。

図3.3 テーブルスクリプトをコピー



3. ［**ファイル**］>［**新規作成**］>［**スクリプト**］を選択してスクリプトウィンドウを開きます。
4. ［**編集**］>［**貼り付け**］を選択します。

これで、データテーブルを再現するスクリプトができました。このスクリプトを保存しておけば、後でいつでもデータテーブルを再現することができます。データテーブルにはスクリプトもそのまま含まれます。

ファイルを読み込むスクリプトを取得する

ファイルを読み込むスクリプトを取得するには、JMP で該当するファイルを開きます。JMP は、ファイルを開いた際の手順を自動的に記録します。

テキストファイルの読み込み

1. ［**ファイル**］>［**開く**］を選びます。

   「データファイルを開く」ウィンドウが表示されます。
2. **「ファイル名」**の横のリストから［**テキストファイル**］を選択します。
3. 「開く方法」セクションで、［**データ、形式を識別する**］を選択します。

4. ファイルを選択し、[**開く**]を選択します。

   ファイルはデータテーブルとして開きます。データテーブルには「ソース」という名前のスクリプトが含まれます。これは、ユーザが使用したテキスト読み込みルールを使ってテキストファイルを読み込むJSLスクリプトです。

5. 「ソース」の赤い三角ボタンのメニューから[**編集**]を選択します。

   スクリプトをコピーして新しいスクリプトウィンドウに貼り付け、保存します。こうして保存したスクリプトを後で実行すれば、同じテキストファイルを再度読み込むことができます。

ヒント：このスクリプトは、読み込むテキストファイルの指定と、それをJMPに正しく読み込むためのオプションを含む`Open()`式で構成されています。式の最初の部分は、読み込むファイルのパス名です。このスクリプトを保存し、別のPCなどで実行する際は、テキストファイルを指すパス名の編集が必要となる場合があります。パス名については、「データタイプ」の章の「パス変数」（120ページ）で詳しく説明しています。

スクリプトを1つにまとめる

週に1回、新しいデータがExcelファイルに保存され、それを基に同じ内容のレポートを作成する必要があるとしましょう。ファイルを開いて毎週同じ手順を実行することもできますが、新しいExcelファイルをJMPに読み込み、分析するところまでをすべて自動で行う方がずっと効率的です。次の例では、スクリプトを準備し、毎週実行する方法を示しています。

Microsoft Excelファイルの読み込み

1. 新しいスクリプトウィンドウを開きます（[**ファイル**] > [**新規作成**] > [**スクリプト**]）。
2. スクリプトウィンドウに、「Solubil.xls」サンプルデータファイルを開く`Open()`式を入力します。ファイルはJMPの「Samples¥Import Data」フォルダにあります。

   ```
   Open( "$SAMPLE_IMPORT_DATA¥Solubil.xls" );
   ```

   後で式を追加するので、式の最後に必ずセミコロンを付けてください。セミコロンには式をつなぐ役目があります。

3. [**編集**] > [**スクリプトの実行**]を選択し、Excelファイルを読み込むスクリプトを実行します。

   Excelファイルがデータテーブルとして開きます。

注：パス変数を使わず、ファイルの絶対パスまたは相対パスを指定することもできます。相対パスを指定した場合は、スクリプトを実行するときに、スクリプトとファイルが相対的に同じ場所になければなりません。絶対パスの場合は、そのスクリプトを使用する他のユーザが指定のパスでファイルにアクセスできることを確認してください。パス名の使用について詳しくは、「データタイプ」の章の「パス変数」（120ページ）を参照してください。

レポートを実行してスクリプトを取得する

一変量の分布、三次元散布図、多変量の3つのレポートを作成するとしましょう。それぞれをGUIを使って作成し、スクリプトをスクリプトウィンドウに追加します。

1. 新しいデータテーブルを開いたまま、［**分析**］>［**一変量の分布**］を選択します。
2. 「**Labels**」以外の列をすべて選択し、［**Y, 列**］を選択します。
3. ［**OK**］をクリックします。
4. 「**eth**」の赤い三角ボタンのメニューから、Ctrl キーを押しながら［**ヒストグラムオプション**］>［**度数の表示**］を選択します。

   6つすべてのヒストグラムに度数が表示されます。
5. 「一変量の分布」ウィンドウで、一変量の分布の赤い三角ボタンのメニューから［**スクリプト**］>［**スクリプトのコピー**］を選択します。
6. スクリプトウィンドウの`Open()`式の1、2行あとにカーソルを置き、［**編集**］>［**貼り付け**］を選択します。
7. 最後の閉じ括弧のあとにセミコロンを入力します。
8. ［**グラフ**］>［**三次元散布図**］を選択します。
9. 「**Labels**」以外の列をすべて選択し、［**Y, 列**］を選択します。
10. ［**OK**］をクリックします。
11. 一変量の分布レポートで行ったように、三次元散布図のスクリプトをコピーしてスクリプトウィンドウに貼り付けます。最後に必ずセミコロンを加えます。
12. ［**分析**］>［**多変量**］>［**多変量の相関**］を選択します。
13. 「**Labels**」以外の列をすべて選択し、［**Y, 列**］を選択します。
14. ［**OK**］をクリックします。
15. 一変量の分布および三次元散布図で行ったように、多変量の相関のスクリプトをコピーしてスクリプトウィンドウに貼り付けます。

図3.4 完成したスクリプト

```
Open( "$SAMPLE_IMPORT_DATA¥Solubil.xls" );

Distribution(
    Continuous Distribution( Column( :eth ), Show Counts( 1 ) ),
    Continuous Distribution( Column( :oct ), Show Counts( 1 ) ),
    Continuous Distribution( Column( :ccl4 ), Show Counts( 1 ) ),
    Continuous Distribution( Column( :c6c6 ), Show Counts( 1 ) ),
    Continuous Distribution( Column( :hex ), Show Counts( 1 ) ),
    Continuous Distribution( Column( :chcl3 ), Show Counts( 1 ) )
);

Scatterplot 3D(
    Y( :eth, :oct, :ccl4, :c6c6, :hex, :chcl3 ),
    Frame3D( Set Grab Handles( 0 ), Set Rotation( -54, 0, 38 ) )
);

Multivariate(
    Y( :eth, :oct, :ccl4, :c6c6, :hex, :chcl3 ),
    Estimation Method( "Row-wise" ),
    Scatterplot Matrix(
        Density Ellipses( 1 ),
        Shaded Ellipses( 0 ),
        Ellipse Color( 3 )
    )
);
```

スクリプトの保存

これで、手動で行ったすべての手順を再現するスクリプトを取得できました。スクリプトを保存し、データテーブルおよびすべてのレポートウィンドウを閉じます。

1. スクリプトが表示されたスクリプトウィンドウで、［**ファイル**］>［**上書き保存**］または［**ファイル**］>［**名前を付けて保存**］を選択します。
2. ファイル名を指定します（たとえば、「週間レポート」）。
3. ［**保存**］をクリックします。

スクリプトの実行

更新されるExcelファイルが毎週同じ場所に保存され、かつ同じ列を含んでいる限り、スクリプトを実行するだけでレポートを自動的に作成できます。

1. 保存したスクリプトを開きます。
2. ［**編集**］>［**スクリプトの実行**］を選択します。

   JMPでExcelファイルが開き、3つのレポートが表示されます。

このスクリプトを他の人に送ることができます。同じ場所の同じExcelファイルにアクセスできる人であれば、JMPでスクリプトを実行し、レポートを見ることができます。

上級者用メモ: Auto-Submit

特定のスクリプトを、スクリプトウィンドウで開くのではなく、直接実行したい場合は、スクリプトの最初の行に次のコマンドを入力します。

```
//!
```

このコマンドの入力が最初の行でなかったり、同じ行に別の文字も入力されている場合、このコマンドは効力を持ちません。

このコマンドは、ファイルを開く際に、無効にすることができます。

1. ［**ファイル**］**>**［**開く**］を選びます。
2. 呼び出されたダイアログにて、JSL ファイルを選択し、Ctrl キーを押しながら［**開く**］をクリックします。

   これにより、スクリプトは実行されずに、スクリプトウィンドウに表示されます。

また、ホームウィンドウでファイルを右クリックし、［**スクリプトの編集**］を選択することにより、スクリプトを実行せずスクリプトウィンドウに表示させることができます。

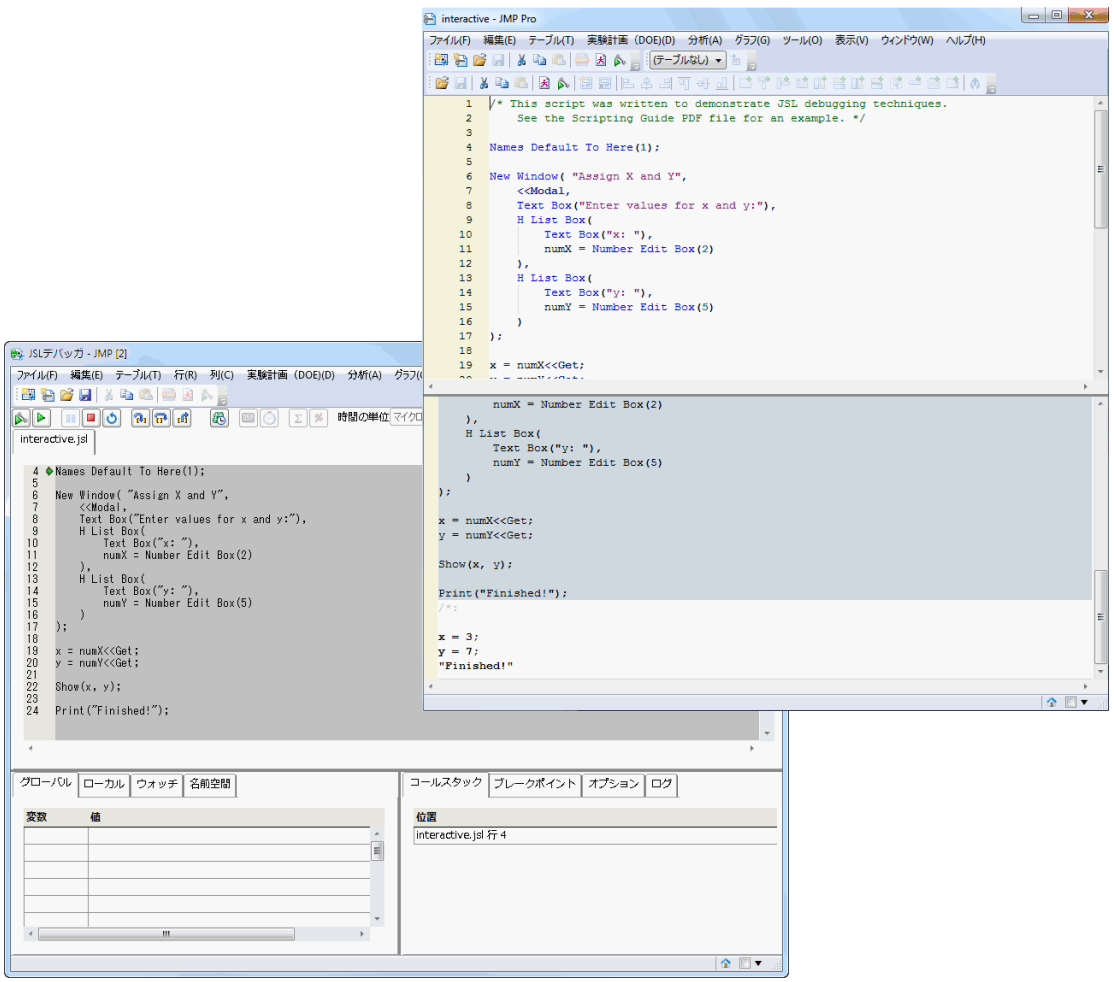
第 4 章

スクリプト作成のツール

スクリプトエディタ、ログウィンドウ、デバッガ、プロファイルの使用

JMP にはスクリプト作成のためのプログラミングツールが用意されています。スクリプトエディタには、構文による色分け、入力のオートコンプリート、対となる括弧の強調表示、コードの折りたたみをはじめ、スクリプトをすばやく作成するための機能が各種用意されています。エラーメッセージやコマンドの戻り値などが出力されるログウィンドウは、スクリプトエディタの中に表示することもできます。JMP スクリプト言語（JSL）デバッガおよびプロファイルは、スクリプトのトラブルシューティングに役立ちます。

図 4.1 ログを含むスクリプトエディタとデバッガ



目次

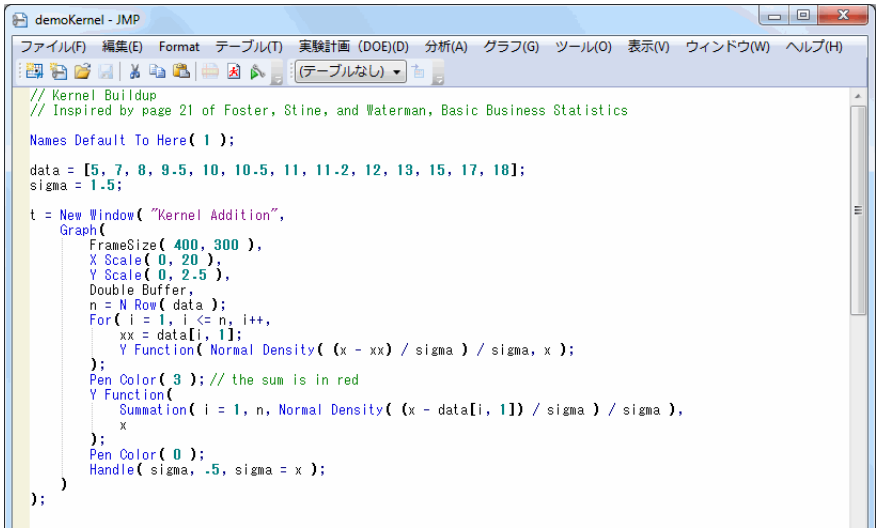
スクリプトエディタの使用 51
スクリプトの実行 51
カラーコーディング 51
オートコンプリート機能 52
ツールヒント 52
ウィンドウの分割 53
括弧の自動マッチ 54
四角形のテキストブロックの選択 55
テキストのドラッグ&ドロップ 56
検索／置換 56
自動フォーマット 56
コード折りたたみマーカーの追加 57
詳細オプション 58
スクリプトエディタの環境設定 59
ログの使用 62
スクリプトウィンドウ内にログを表示 63
ログの保存 63
スクリプトのデバッグ／プロファイル 63
デバッガとプロファイルのウィンドウ 64
ブレークポイントの操作 67
変数の確認 70
ウォッチの操作 70
デバッガでの環境設定の変更 71
デバッガセッションの持続性 71
スクリプトのデバッグとプロファイルの例 72

スクリプトエディタの使用

スクリプトエディタを使うと、JSL スクリプトを簡単に書いたり読んだりできます。図4.2 に、構文による色分け、インラインのコメント、自動フォーマットなどの基本的な機能を示します。その他の一般的なプログラミングオプションについては、この節で後述します。

スクリプトエディタの機能は、ログウィンドウをはじめ、スクリプトの編集や記述ができる場所ならどこででも使用できます（スクリプトの索引やアプリケーションビルダーなど）。

図4.2 スクリプトエディタ



スクリプトの実行

スクリプト全体を実行するには、［**編集**］>［**スクリプトの実行**］を選択します。

特定の行のスクリプトのみを実行するには、それらの行を選択し、［**編集**］>［**スクリプトの実行**］を選択します。

Windows では、1 行または複数の行を選択してから、数字キーパッドの Enter を押して実行することもできます。

カラーコーディング

スクリプトウィンドウでは次のような色が使用されます。

- テキスト、識別子（JSL 関数）、括弧、ユーザマクロは黒
- スクリプトエディタの背景は白
- デバッガの背景、無効になっている背景、ガイドはグレー

- コメントは緑
- 文字列は紫
- 数値は緑がかった青
- 演算子記号、最初のキーワード、JSL オブジェクトは濃い青
- 演算子名、2 番目と 3 番目のキーワード、マクロは青
- 不明なオブジェクトは赤

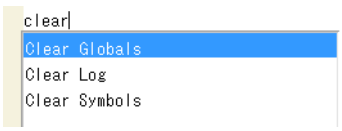
色は環境設定でカスタマイズできます。「スクリプトエディタの環境設定」（59 ページ）を参照してください。

オートコンプリート機能

関数の名前を正確に覚えていないときは、自動入力機能を使えば、途中までタイプした内容に一致する関数のリストが表示されます。Windows の場合は Ctrl キーを押しながらスペースキーを押します（Macintosh では Option キーを押しながら Esc キー）。

JSL 変数をクリアしたいものの、そのコマンドを覚えていないとしましょう。その場合、`clear` とタイプし、Ctrl キーを押しながらスペースキーを押すと、クリアコマンドの候補が表示されます。挿入したいコマンドを選択します。

図 4.3 オートコンプリートの例

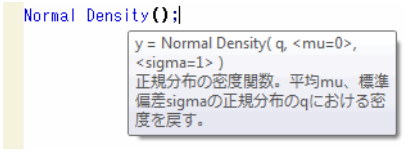


ツールヒント

関数を使用する際に構文を覚えていない場合や、詳しい情報を知りたい場合は、その関数の上にカーソルを置くと短い説明が表示されます。ツールヒントは JSL 関数のみで使用でき、プラットフォームのコマンドやメッセージ、ユーザが作成した関数では使用できません。JSL 関数名は、スクリプトエディタに青色で表示されます。

ツールヒントには、関数の構文や引数、簡単な説明が表示されます（図 4.4）。ヒントは、スクリプトエディタウィンドウのステータスバーにも表示されます。

図 4.4 JSL 関数のツールヒント



スクリプトを実行した後、変数名の上にカーソルを置くと、変数の現在の値がわかります。変数のツールヒントをオフにするには、**[環境設定] > [スクリプトエディタ] > [変数値のヒントを表示する]** の選択を解除します。

関数のツールヒントをオフにするには、［**環境設定**］>［**スクリプトエディタ**］>［**演算子または関数のヒントを表示する**］の選択を解除します。

JSL 変数のツールヒントの例

1. スクリプトウィンドウに次の内容を入力し、実行します。

   ```
   my_variable = 8;
   ```

2. 実行後、変数名の上にカーソルを置きます。

   ツールヒントに変数の名前と値（8）が表示されます。

3. 次の内容を入力し、実行します。

   ```
   my_variable = "eight";
   ```

4. 実行後、変数名の上にカーソルを置きます。

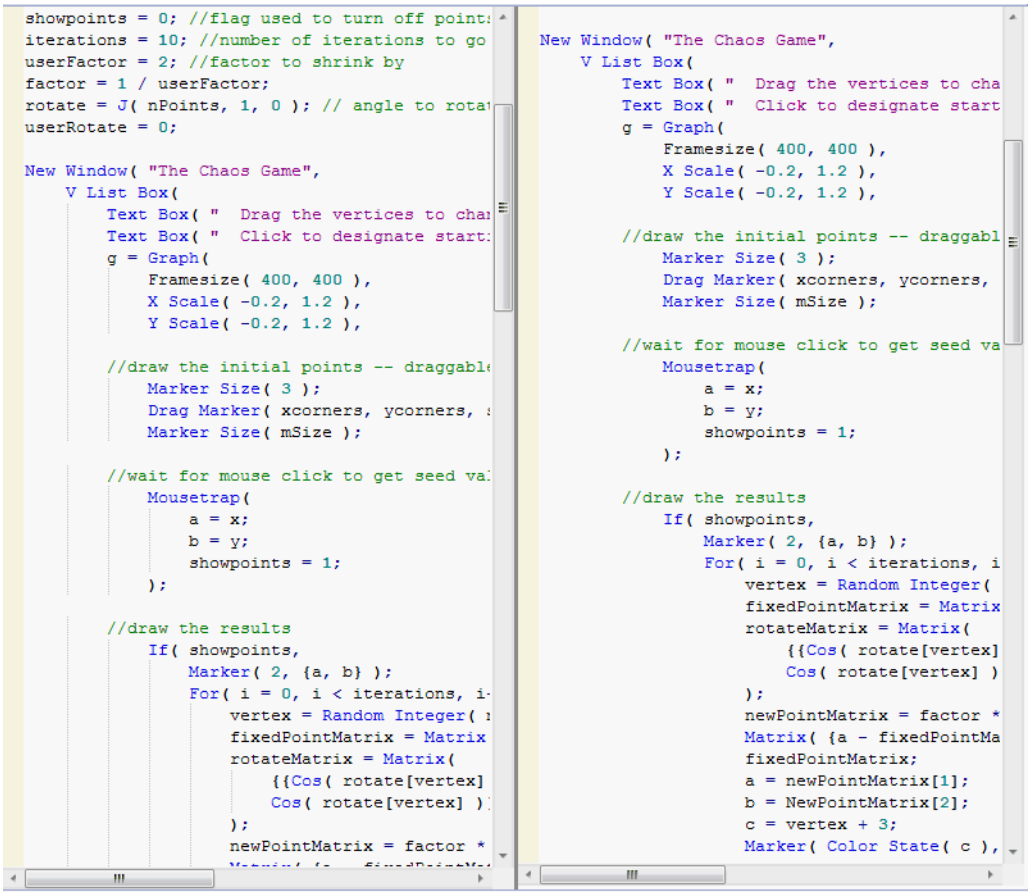
   ツールヒントに変数の名前と値（"eight"）が表示されます。

ウィンドウの分割

スクリプトエディタウィンドウは、縦または横に 2 つに分割できます。ウィンドウを分割すると、コード内の 2 つの箇所を別々にスクロールし、編集することができます。一方のウィンドウでコードに変更を加えた場合、他方のウィンドウでもただちにその変更が反映されます。

- 開いているスクリプトエディタウィンドウを分割するには、ウィンドウ内で右クリックし、［**分割**］>［**横**］または［**縦**］を選択します。
- 分割したウィンドウを元に戻すには、右クリックして［**分割の削除**］を選択します。

図 4.5 ウィンドウを横に分割した例



```
showpoints = 0; //flag used to turn off point
iterations = 10; //number of iterations to go
userFactor = 2; //factor to shrink by
factor = 1 / userFactor;
rotate = J( nPoints, 1, 0 ); // angle to rota
userRotate = 0;

New Window( "The Chaos Game",
    V List Box(
        Text Box( "  Drag the vertices to cha
        Text Box( "  Click to designate start
        g = Graph(
            Framesize( 400, 400 ),
            X Scale( -0.2, 1.2 ),
            Y Scale( -0.2, 1.2 ),

        //draw the initial points -- draggabl
            Marker Size( 3 );
            Drag Marker( xcorners, ycorners,
            Marker Size( mSize );

        //wait for mouse click to get seed va
            Mousetrap(
                a = x;
                b = y;
                showpoints = 1;
            );

        //draw the results
            If( showpoints,
                Marker( 2, {a, b} );
                For( i = 0, i < iterations, i
                    vertex = Random Integer(
                    fixedPointMatrix = Matrix
                    rotateMatrix = Matrix(
                        {{Cos( rotate[vertex]
                        Cos( rotate[vertex] )
                    );
                    newPointMatrix = factor *
```

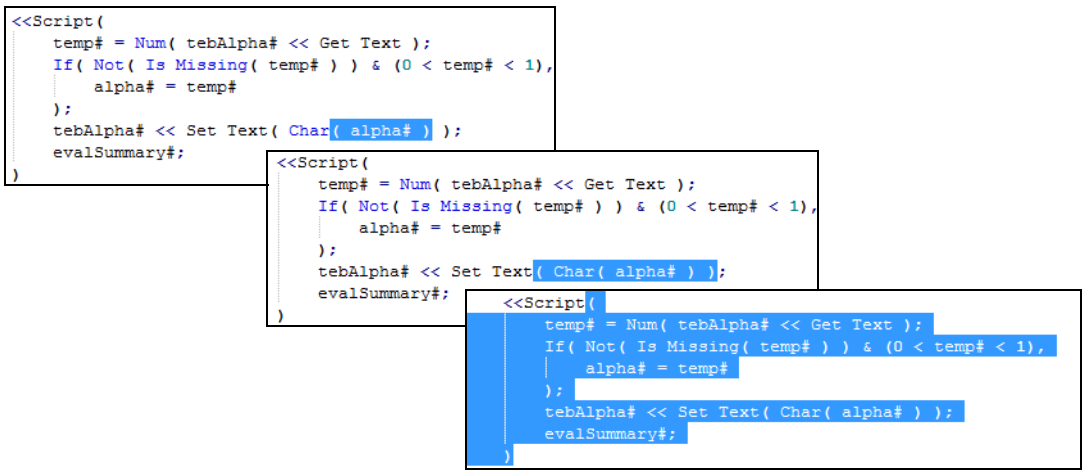
```
New Window( "The Chaos Game",
    V List Box(
        Text Box( "  Drag the vertices to cha
        Text Box( "  Click to designate start
        g = Graph(
            Framesize( 400, 400 ),
            X Scale( -0.2, 1.2 ),
            Y Scale( -0.2, 1.2 ),

        //draw the initial points -- draggabl
            Marker Size( 3 );
            Drag Marker( xcorners, ycorners,
            Marker Size( mSize );

        //wait for mouse click to get seed va
            Mousetrap(
                a = x;
                b = y;
                showpoints = 1;
            );

        //draw the results
            If( showpoints,
                Marker( 2, {a, b} );
                For( i = 0, i < iterations, i
                    vertex = Random Integer(
                    fixedPointMatrix = Matrix
                    rotateMatrix = Matrix(
                        {{Cos( rotate[vertex]
                        Cos( rotate[vertex] )
                    );
                    newPointMatrix = factor *
                    Matrix( {a - fixedPointMa
                    fixedPointMatrix;
                    a = newPointMatrix[1];
                    b = NewPointMatrix[2];
                    c = vertex + 3;
                    Marker( Color State( c ),
```

括弧の自動マッチ

スクリプトエディタでは、**括弧**（丸括弧、角括弧、中括弧）を次の方法でマッチさせられます。

- 開き括弧をタイプすると、閉じ括弧が追加されます。
- 開き括弧または閉じ括弧の横にカーソルを置くと、その括弧とそれにマッチする括弧が青で強調表示されます。マッチする括弧がない場合、括弧は赤で強調表示されます。
- 括弧をダブルクリックすると、マッチする括弧と、その間にあるものがすべて選択されます。
- 式の中にカーソルを挿入し、Ctrl+] キー（Windows）または command+B キー（Macintosh）を押すと、式全体が選択されます （式を囲む括弧を含む）。次に上位にある式を強調表示するには、このプロセスを繰り返します。図 4.6 に例を示します。

図 4.6 括弧の自動マッチの各ステップ



開き括弧をタイプすると、閉じ括弧が自動的に追加されます。括弧の中にコードを入力した後、閉じ括弧をタイプすると、カーソルが自動的に挿入されていた閉じ括弧の後ろに移動します。これは、余計な閉じ括弧が追加されるのを防ぐ機能です。

括弧の自動マッチ機能は「環境設定」ウィンドウでオンまたはオフにできます。詳細は、「スクリプトエディタの環境設定」（59 ページ）の節を参照してください。

四角形のテキストブロックの選択

四角形のテキストブロックを選択するには、Alt キーを押しながらカーソルをブロックの開始位置に置き、そのまま終了位置までドラッグします。選択したブロックは、コピーしたり切り取ったりできます。

次のコードを、コメントマークを除いて選択したいとします。

```
// Y( :Y ),
// X( :X ),
```

Y から始まる四角形の領域を選択します。それをコピーしてペーストすると、次のコードになります。

```
Y( :Y ),
X( :X ),
```

四角形の領域を選択した場合、テキストの構造を維持するために必要な個所に改行（リターン）が挿入されます。次の例では、両方の行の `Get Menu Item State` を選択します。

```
bb << Get Menu Item State(1),
bb << Get Menu Item State(2),
```

これをコピーしてペーストすると、各行の末尾に改行（リターン）が挿入されます。

```
Get Menu Item State
Get Menu Item State
```

テキストのドラッグ＆ドロップ

スクリプトエディタ内、ウィンドウ間、またはデータテーブルからスクリプトエディタウィンドウで、テキストをドラッグ＆ドロップできます。Windows で、テキストのコピーをドロップしたいときは、まず Ctrl キーを押してからドラッグ＆ドロップします。Macintosh の場合、テキストはデフォルトでコピーされます。

テキストをドラッグ＆ドロップするには、次の手順を行います。

- データテーブルの行または列を選択し、少し間を置いてからスクリプトエディタウィンドウにドロップします。
- テキストフィールド内のテキストをダブルクリックし、スクリプトエディタウィンドウにドロップします。データテーブルのセルに含まれているテキストなど、選択可能なテキストすべてに適用できます。

Windows の場合、最小化したウィンドウにテキストをドラッグ＆ドロップすることもできます。

1. テキストを、ウィンドウの右下隅にあるホームウィンドウボタン の上にドラッグします。

   ホームウィンドウが開きます。
2. 「ウィンドウリスト」にある任意のウィンドウ名の上にテキストをドラッグします。

   該当するウィンドウが表示されます。
3. 任意の場所にテキストをドロップします。

検索／置換

スクリプトウィンドウには、正規表現のサポートを含め、多くの検索／置換オプションが用意されています。たとえば、次のような正規表現で検索できます。

```
get.*name
```

これは、「Get Button Name」、「GetFontName」などのメッセージを戻します。

^ および $（それぞれ行の冒頭および行の末尾に一致）や \n（改行文字に一致）などの基本的な正規表現もサポートされています。

「検索」ウィンドウの詳細については、『JMP の使用法』を参照してください。

自動フォーマット

スクリプトエディタでは、読みやすいようにスクリプトの体裁を整えることができます。JMP で（たとえばプラットフォームスクリプトの保存時に）生成されるスクリプトには、適切な場所に自動的にタブや改行が挿入されます。

また、手で入力したスクリプトの場合も、読みにくいようであれば（すべてのコマンドが空白文字を挟まずにつながっているなど）の体裁を整えることもできます。［**編集**］メニューから［**スクリプトを再フォーマット**］を選択します。

ヒント：このコマンドは、スクリプトに括弧が一致していないなどの不具合がある場合に警告メッセージを表示します。

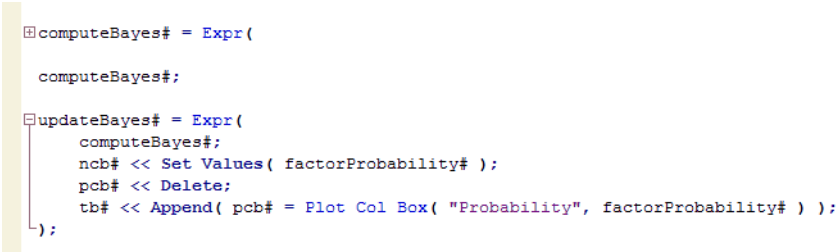
コード折りたたみマーカーの追加

コード折りたたみマーカーを使ってコードのブロックの開始位置と終了位置を指定すると、単独の関数内のコードを折りたたんだり、展開したりできます。

この機能を使用するには、［スクリプトエディタ］の環境設定で［**JSL コードの折りたたみ**］をオンにします。コードのブロックを展開する（または折りたたむ）には、スクリプトを右クリックし、［**詳細**］>［**すべてを展開する**］または［**すべてを折りたたむ**］を選択します。

この環境設定を選択すると、Function と Expr の式が折りたためるようになります。他の式に折りたたみマーカーを追加する方法については、「コード折りたたみのキーワードの追加」（57 ページ）を参照してください。

図 4.7 スクリプトに表示されたコード折りたたみマーカー



デフォルトでは、スクリプトを保存して JMP を再開した場合、コードは折りたたんだ状態で表示されます。コードの折りたたみ状態を保存するには、［スクリプトエディタ］環境設定で［**コードの折りたたみ情報を保存および復元する**］を選択します。

コード折りたたみのキーワードの追加

その他の単独の関数にはカスタムのコード折りたたみを使用できます。jmpKeywords.jsl という名前のファイルを作成し、次のように関数をリスト内に入力します。

```
{"If", "For", "For Each Row", "Where", "Try", "V List Box", "H List Box", "Button
  Box"}
```

オペレーティングシステムに応じて、次のいずれかのディレクトリにファイルを保存します。Windows の場合、次のディレクトリがこの順番に調べられます。

- C:¥ProgramData¥SAS¥JMP¥<バージョン>¥
- C:¥ProgramData¥SAS¥JMP¥
- C:¥ユーザー¥<ユーザ名>¥AppData¥Local¥SAS¥JMP¥<バージョン>¥
- C:¥ユーザー¥<ユーザ名>¥AppData¥Local¥SAS¥JMP¥

Macintoshの場合、次のディレクトリがこの順番に調べられます。

- /Library/Application Support/JMP/<バージョン>/
- /Library/Application Support/JMP/
- ~/Library/Application Support/JMP/<バージョン>/
- ~/Library/Application Support/JMP/

JMP Proを使っている場合も、jmpKeywords.jslは指定されたJMPディレクトリに保存されます。

［**コード折りたたみのキーワードの追加を許可する**］環境設定が選択されている場合、前述のディレクトリ内のjmpKeywords.jslファイルが検索されます。

jmpKeywords.jslサンプルスクリプトには、JSL関数のリストが含まれており、これらをカスタマイズして指定の場所のいずれかに保存することができます。

メモ:

- jmpKeywords.jsl内のリストは大文字と小文字を区別します。
- コードの折りたたみは、メッセージ、プラットフォーム、およびコメントには対応していません。
- ユーザ定義の関数を折りたたむには、その関数を`Expr()`式内で定義します。
- jmpKeywords.jsl内のリストを編集した後、［**コード折りたたみのキーワードの追加を許可する**］環境設定をオフにしてから再度オンにして、変更を有効にします。

詳細オプション

スクリプトエディタでテキストの一部を選択して右クリックし、［詳細］をポイントすると、次のようなオプションが表示されます。

オプション	説明
すべてを展開する	（JSLコードの折りたたみ機能が有効な場合のみ表示されます。）コードのブロックをすべて展開します。
すべてを折りたたむ	（JSLコードの折りたたみ機能が有効な場合のみ表示されます。）コードのブロックをすべて折りたたみます。
ブロックのコメント化	選択したテキストがコメントになります。
ブロックのコメント化解除	選択したコメントがコメントでなくなります。
大文字にする	選択したテキストが大文字に変わります。
小文字にする	選択したテキストが小文字に変わります。

スクリプトエディタの環境設定

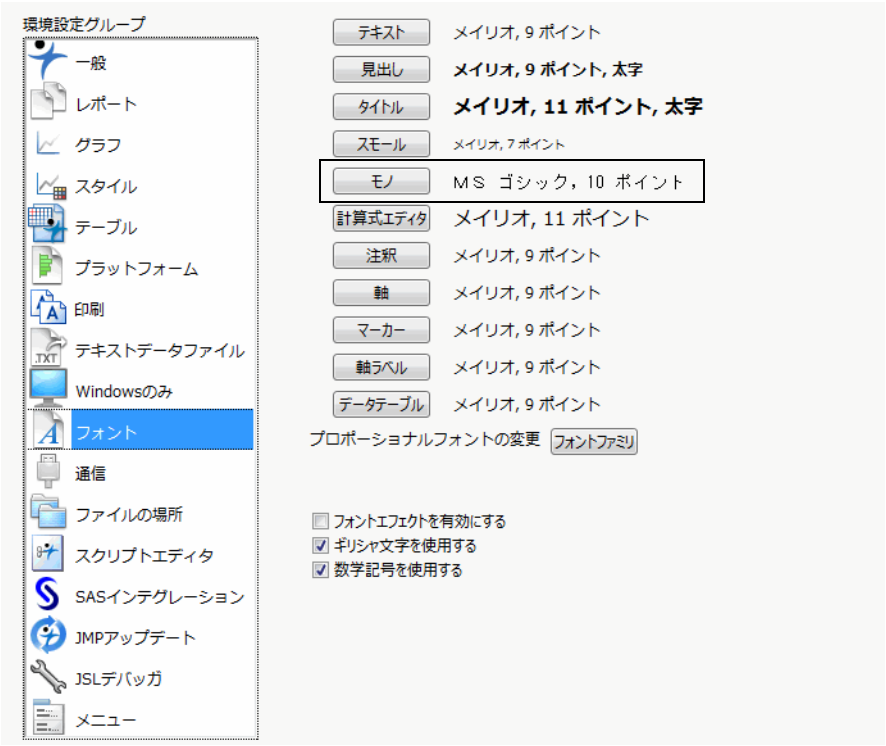
JMP の環境設定では、フォント、色、スペースのオプションといったスクリプトエディタの設定をカスタマイズできます。

フォントの設定

1. ［**ファイル**］>［**環境設定**］を選択します。
2. ［**フォント**］グループを選択します。
3. ［**モノ**］をクリックして、スクリプトエディタ用のフォントを設定します。

フォントの環境設定の詳細については、『JMP の使用法』を参照してください。

図 4.8 スクリプトウィンドウ用のフォントの変更



スクリプトエディタの環境設定

スクリプトエディタをより詳細にカスタマイズするには、［**ファイル**］>［**環境設定**］>［**スクリプトエディタ**］を選択します。

環境設定	説明
タブを使用する	このチェックボックスをオンにすると、スクリプトでタブを使用できます。このオプションは、デフォルトでオンになっています。 チェックボックスをオフにすると、入力したタブ文字がスペースに置き換えられます。
タブの幅	タブ文字1つに相当するスペースの数を入力します。[タブを使用する] チェックボックスをオフにした場合は、入力したタブ文字が、ここで指定した数のスペースに置き換えられます。デフォルト値は4です。
ドキュメントの下部にスペースを空ける	このオプションをオンにすると、スクリプトの最後の空白行からのスクロールアップが有効になります。このオプションは、Windows ではデフォルトでオンになっており、Macintosh ではデフォルトでオフになっています。
括弧を自動マッチさせる	このチェックボックスをオンにすると、スクリプトエディタで、丸括弧、角括弧、中括弧の開き括弧を入力すると、対応する閉じ括弧が自動的に追加されます。このオプションは、デフォルトでオンになっています。
行番号を表示する	スクリプトエディタの左側に行番号を表示する場合は、このチェックボックスをオンにします。このオプションは、デフォルトではオフになっています。
インデントガイドを表示する	インデントを示す薄い縦線を表示する場合は、このチェックボックスをオンにします。このオプションは、デフォルトでオンになっています。
演算子または関数のヒントを表示する	JSL の演算子や関数に対して、ヒントを表示する場合は、このチェックボックスをオンにします。このオプションは、デフォルトでオンになっています。
変数値のヒントを表示する	変数値のヒントを表示する場合は、このチェックボックスをオンにします。このオプションは、デフォルトでオンになっています。
テキストの折り返し	スクリプトエディタでテキストを折り返して表示する場合は、このチェックボックスをオンにします。このオプションはデフォルトではオフになっています。
スクリプトウィンドウ内にログを表示する	スクリプトの編集または実行時に、スクリプトウィンドウ内にログウィンドウを表示する場合は、このチェックボックスをオンにします。このオプションは、デフォルトではオフになっています。

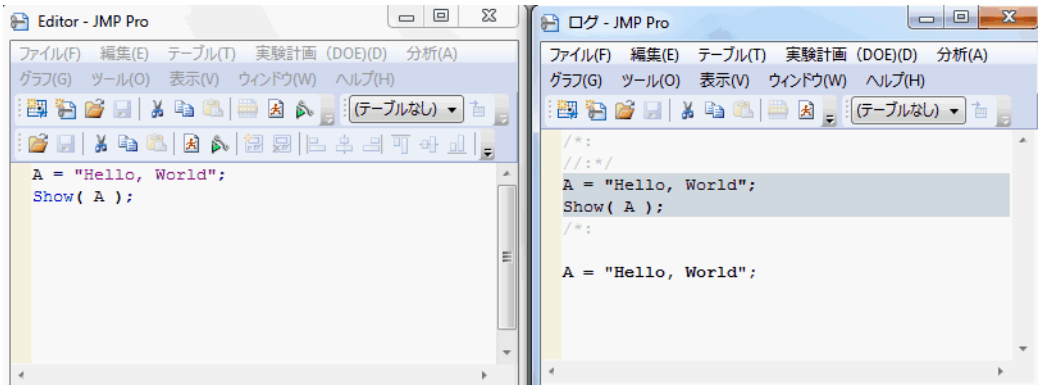
環境設定	説明
不明なオブジェクトメッセージを色分けする	対象であるオブジェクトに対して定義されていないオブジェクトメッセージをカラー表示にするには、このチェックボックスをオンにします。不明なオブジェクトメッセージは、**「不明なメッセージの色」**で指定された色で表示されます。 **注：**このオプションを使用すると、コンテキスト内でメッセージ名が検索されるため負荷がかかり、JSL エディタのパフォーマンスが低下する可能性があります。
コードの折りたたみ情報を保存および復元する	コードが折りたたまれた状態にあるか展開された状態にあるかを記憶し、スクリプトを再度開いたときに、同じ状態に復元します。
括弧の内側にスペースを入れる	丸括弧、角括弧、中括弧と括弧内の文字列との間に自動的にスペースを挿入し、スクリプトの体裁を調整する場合は、このチェックボックスをオンにします。このオプションは、デフォルトでオンになっています。
演算子名にスペースを入れる	演算子名や関数名で使われている単語の間にスペースを自動挿入する場合は、このチェックボックスをオンにします。たとえば、このチェックボックスをオンにすると、`NewWindow` ではなく `New Window` と表示されます。このオプションは、デフォルトでオンになっています。
JSL コードの折りたたみ	スクリプトエディタでコードの折りたたみマーカーを表示する場合は、このチェックボックスをオンにします。このマーカーは、`Function()` 式および `Expr()` 式の開始と終了を示します。このマークが付いたコードブロックは展開および折りたたみができます。このオプションはデフォルトではオフになっています。 マーカーの外観は、**「JSL コードの折りたたみマーカー」**のドロップダウンメニューで選択できます。 スクリプトエディタで折りたたみマーカーのキーワードを追加できるようにするには、[**コード折りたたみのキーワードの追加を許可する**] をオンにします。詳細は、「コード折りたたみのキーワードの追加」（57 ページ）の節を参照してください。
色の選択	リストに表示されている箇所の色を変更するには、該当するカラーボックスをクリックし、目的の色を選択します。デフォルト設定の詳細については、「カラーコーディング」（51 ページ）を参照してください。

スクリプトエディタの環境設定の詳細については、『JMP の使用法』を参照してください。

ログの使用

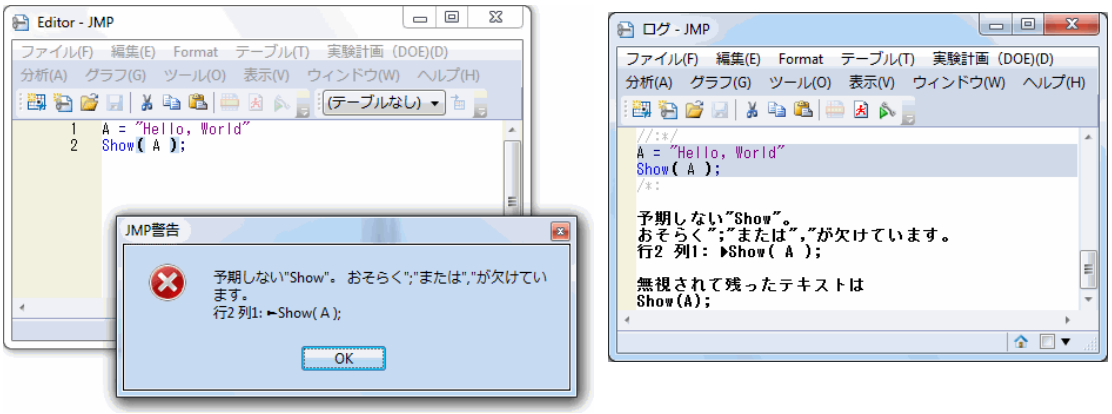
スクリプトを実行すると、ログウィンドウに出力が表示されます。実行されたスクリプトはログ内でグレー表示され、その下に出力が表示されます（図4.9）。

図4.9 スクリプトウィンドウ（左）とログウィンドウ（右）



構文エラーは、行番号、および処理できなかったコードとともにログ内にレポートされます (図4.10)。

図4.10 構文エラーメッセージ



［**表示**］>［**ログ**］を選択してログウィンドウを表示します（Macintoshの場合は［**ウィンドウ**］>［**ログ**］）。

Windowsの場合、ログを表示するタイミングを、JMPが起動したとき、テキストが書き込まれたとき、または明示的に開いたときのいずれかに指定できます。JMPログウィンドウを開く設定を変更するには、［**ファイル**］>［**環境設定**］>［**Windowsのみ**］を選択します。

スクリプトウィンドウ内にログを表示

スクリプトウィンドウ内にログを表示するには、ウィンドウ内を右クリックし、[**ウィンドウ内にログを表示**]を選択します。このオプションを使えば、スクリプトを編集、実行し、変更の結果をすばやく確認してスクリプトの作成を継続することがより簡単になります。

スクリプトの索引のスクリプトウィンドウには、埋め込みのログが常に表示されますが、アプリケーションビルダーおよびデバッガには表示されません。

ログの保存

以下の手順でログをテキストファイルとして保存すると、どんなテキストエディタでも開くことができます。ログのテキストファイルをダブルクリックすると、JMP ではなく、普段使っているテキストエディタでログが開きます。

1. 「ログ」ウィンドウをアクティブにします（「ログ」ウィンドウをクリックして最前面に持ってきます）。
2. [**ファイル**]メニューの[**上書き保存**]（Macintosh では[**保存**]）か、[**名前を付けて保存**]（Macintosh では[**別名で保存**]）を選択します。
3. Windows の場合は、拡張子 .txt をつけたファイル名を指定します。Macintosh の場合は .txt の拡張子では保存できず、常に .jsl の拡張子となります。
4. [**保存**]をクリックします。

スクリプトのデバッグ／プロファイル

開いたスクリプト内で、[**スクリプトのデバッグ**]ボタン  をクリック（または[**編集**] > [**スクリプトのデバッグ**]を選択）すると、スクリプトがJSL デバッガウィンドウに表示されます。次のようなショートカットキーを使用することもできます。

- Ctrl + Shift + R（Windows）
- Shift + Command + R（Macintosh）

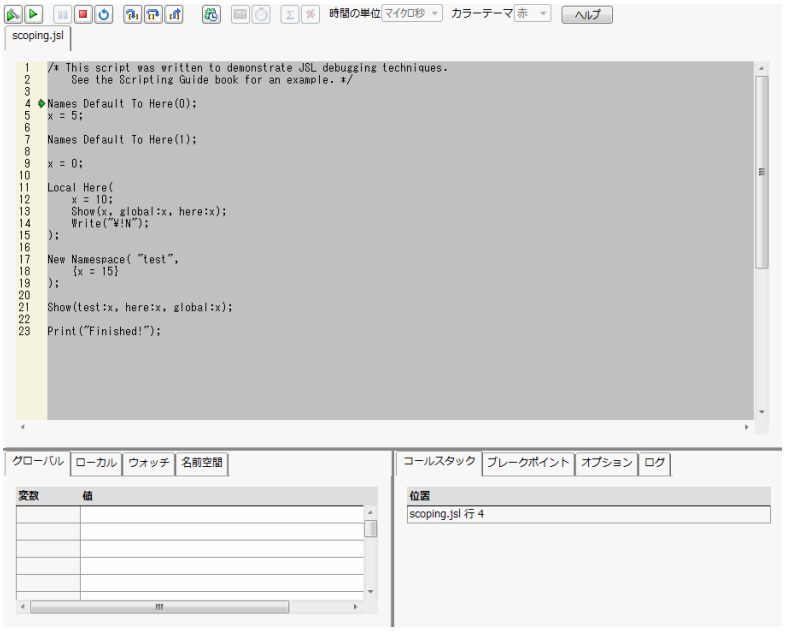
JSL デバッガを使うと、スクリプト内でエラーが発生したり、動作しない場所を特定することができます。スクリプトの一部をコメントアウトしたり `Print()` を追加したりしなくても、デバッガを使って問題を検出できます。

JSL デバッガが開いたら、そのまま作業を行うか、[**JSL スクリプトのプロファイル**]ボタン  をクリックしてJSL プロファイルモードに切り替えます。JSL プロファイルは、スクリプトを効率的にする手助けになります。スクリプトの実行中にプロファイルが作成され、特定の行の実行にかかった時間や、特定の行が実行された回数などがわかります。

デバッガとプロファイルのウィンドウ

デバッガはJMPの新しいインスタンスとして起動されます（図4.11）。元のインスタンスは、スクリプトによって何らかの操作を必要とするものが作成されるまで、操作不可能になります。操作の必要なものが作成された場合、ユーザがそれを実行するまで、デバッガウィンドウの方が操作不可能になります。その後、制御が再びデバッガに戻ります。JMPの元のインスタンスでの作業に戻るには、デバッガを閉じなければなりません。

図4.11 デバッガウィンドウ



デバッガとJSLプロファイルを操作するには、上部にあるボタンを使用します。デバッグまたはプロファイル作成の対象であるスクリプトがタブに表示されます。スクリプトの中に他のスクリプトが含まれているときは、それぞれが新しいタブに表示されます。

デバッガの最下部にあるタブには、変数や名前空間、ログ、現在の実行ポイントを表示するためのオプション、ブレークポイントを操作するためのオプション、および設定オプションがあります。

実行ボタンの使用

デバッガまたはJSLプロファイル内でスクリプトの実行を制御するには、上部にあるボタンを使用します。

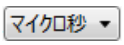
表4.1 デバッガボタンの説明

ボタン	ボタン名	アクション
	実行	デバッガ内のスクリプトを、ブレークポイントまたはスクリプトの末尾まで実行します。

表 4.1 デバッガボタンの説明（続き）

ボタン	ボタン名	アクション
	ブレークポイントなしで実行 プロファイルの実行	スクリプトを、途中で停止することなく最後まで実行します。
	すべて中断	たとえば、非常に長いループを実行しているときなど、スクリプトがビジーであるとき、［すべて中断］をクリックすると、スクリプト内のすべてのアクションが中断されてデバッガまたはJSL プロファイルに戻ります。 実行中のスクリプトが、ダイアログの入力や開いたウィンドウへの操作など、ユーザの操作を待機している場合、デバッガおよびJSL プロファイルは実行を中断できないことがあります。
	停止	スクリプトのデバッグを停止し、デバッガまたはJSL プロファイルを終了します。
	再開	デバッガの現在のセッションを閉じ、新しいセッションを開きます。
	ステップイン	関数内またはインクルードされたファイル内にステップインします。それ以外は、ステップオーバーと同じ働きをします。
	ステップオーバー	呼び出された式、関数、または `Include()` ファイルにステップインすることなく、1行のすべての式、または複数行にまたがる複雑な式を実行します。
	ステップアウト	現在のスクリプトまたは関数をブレークポイントまたは最後まで実行し、呼び出しポイントまで戻ります。主スクリプトにおいてデバッガが最後まで到達した場合は、「プログラムの実行が終了しました」というメッセージが表示されます。デバッガ自体は開いたままなので、プログラムの最終的な状態を点検することができます。
	JSL スクリプトのプロファイル	JSL プロファイルを開きます。（JSL プロファイルを開始するには、［プロファイルの実行］ボタンを押します。）JSL プロファイルを使うと、特定の行の実行にかかった時間や、特定の行が実行された回数などがわかります。次の点を念頭に置いてください。 • デバッガと JSL プロファイルの間を自在に切り替えることができるのは、プログラムの開始前のみです。 • デバッガのボタンの一部は、プロファイルの実行中は無効になります。 • JSL プロファイルモードでは、ブレークポイントがすべて無効になります。

表 4.1 デバッガボタンの説明（続き）

ボタン	ボタン名	アクション
	行の実行回数のプロファイルを表示	各行が何回実行されたかを表示します。
	行の処理時間のプロファイルを表示	行の実行にかかった時間を表示します。
	総数でプロファイルを表示	実行回数の場合は、その行が何回実行されたかを表示します。処理時間の場合は、その行が完了するまでにかかったマイクロ秒数（またはミリ秒数、秒数）を表示します。
	パーセントでプロファイルを表示	実行回数の場合は、個々の行の実行回数を総数で割った値を表示します。処理時間の場合は、個々の行にかかった時間の割合（行の処理時間÷合計時間×100）を表示します。
	時間の単位	時間の単位をマイクロ秒、ミリ秒、秒のいずれかに設定します。
	カラーテーマ	JSL プロファイルの陰影の色を設定します。

変数リスト

デバッガの左下にあるタブでは、グローバル変数やローカル変数、ウォッチ変数、名前空間内の変数が確認できます。

グローバル　［グローバル］タブにはグローバル変数がリストされ、スクリプトの各ステップを実行するにつれて値が更新されます。変数は、初期化の際に追加されます。実行済みのスクリプトによってグローバル変数が定義されている場合は、デバッガの起動時に、それらの変数と現在の値が表示されます。「変数の確認」（70 ページ）を参照してください。

ローカル　［ローカル］タブにはすべての変数がスコープ別にリストされ、スクリプトの各ステップを実行するにつれて値が更新されます。メニューからスコープを選択できます。「変数の確認」（70 ページ）を参照してください。

ウォッチ　コードの各ステップを実行していく際、値を確認したい変数または式の値がある場合は、ここに追加します。これは、スクリプトで多くの変数を使用していて、［グローバル］や［ローカル］のリスト内で確認するのが難しい場合に特に便利です。「ウォッチの操作」（70 ページ）を参照してください。

名前空間　定義された名前空間は、このメニューに追加されます。名前空間を選択すると、名前空間内で使用されている変数とその値が表示されます。「変数の確認」（70 ページ）を参照してください。

デバッガオプション

デバッガの右下のタブでは、コールスタックの表示、ブレークポイントの操作、オプションの設定、ログの表示ができます。

コールスタック　コールスタックには、スクリプト内、関数内の現在の実行ポイントがリストされます。主スクリプトは常にリストの最初に表示されます。関数を呼び出すと、その関数は呼び出しスクリプトの上に追加されます。同様に、インクルードされたファイルもリストの一番上に追加されていきます。関数またはスクリプトの実行を終了すると、それがリストから削除され、次の関数またはスクリプトに戻ります。各ステップを実行するに従って、現在の行番号が更新されます。

コールスタック内の行をダブルクリックすると、カーソルがその行に移動します。

ブレークポイント　ブレークポイントを追加、編集、削除し、無効または有効にします。「ブレークポイントの操作」（67ページ）を参照してください。［ブレークポイント］タブで行をダブルクリックすると、カーソルがその行に移動します。

オプション　このタブでは、デバッガの環境設定をインタラクティブに設定できます。「デバッガでの環境設定の変更」（71ページ）を参照してください。

対数　デバッグ中のスクリプトのログが表示されます。

ブレークポイントの操作

ブレークポイントはスクリプトの実行を中断します。スクリプトのデバッグは一行ずつ進めることができますが、スクリプトが長い、または複雑な場合、退屈で時間のかかる作業になってしまいます。そのような場合、確認したい位置にブレークポイントを設定し、デバッガでスクリプトを実行することができます。スクリプトは、ブレークポイントに達するまで通常どおりに実行されます。ブレークポイントの位置で実行が中断されるので、変数の値を確認したり、そこから一行ずつのデバッグを開始したりできます。

JMP は、セッションを越えてブレークポイントを保持するため、JMP を閉じて再度開いても、ブレークポイントは同じ位置に表示されます。

ヒント：行番号をオンにするには、スクリプト内を右クリックし、**［行番号を表示する］**を選択します。すべてのスクリプトでデフォルトで行番号を表示させるには、スクリプトエディタの環境設定で［行番号を表示する］をオンにします。

ブレークポイントの作成

ブレークポイントを作成する際、条件やブレーク時の動作を指定することができます。それには、［ブレークポイント］タブの  をクリックし、ブレークポイントの情報を入力します。

また、次のいずれかの手順ですばやくブレークポイントを作成できます。

- デバッガの余白で、該当する行をクリックします（行番号が表示されている場合は、行番号の右側）。
- デバッガの余白で、ブレークポイントを設定したい位置を右クリックし、**［ブレークポイントの設定］**を選択します。

ブレークポイントを挿入した箇所と［ブレークポイント］タブに。赤いブレークポイントアイコンが表示されます。

ブレークポイントの削除

次のいずれかを実行します。

- デバッガの余白で、ブレークポイントアイコンをクリックします。
- デバッガの余白で、ブレークポイントアイコンを右クリックし、［**ブレークポイントのクリア**］を選択します。
- ［ブレークポイント］タブでブレークポイントを選択し、  をクリックします。
- ［ブレークポイント］タブで  をクリックし、（選択したブレークポイントだけではなく）すべてのブレークポイントを削除します。

赤いブレークポイントアイコンが消え、［ブレークポイント］タブにもブレークポイントが表示されなくなります。

ブレークポイントの有効化および無効化

スクリプトのエラーを修正した後、そのブレークポイントを通過して正常に実行できるかどうかを確認するには、ブレークポイントを無効にします。ブレークポイントは、いつでも必要なときに有効にできるため、作成し直す必要がありません。

次のいずれかを実行します。

- デバッガの余白で、ブレークポイントアイコンを右クリックし、［**ブレークポイントを有効にする**］または［**ブレークポイントを無効にする**］を選択します。
- ［ブレークポイント］タブで、ブレークポイントのチェックボックスを選択するか、選択を解除します。
- ［ブレークポイント］タブで  をクリックし、すべてのブレークポイントを無効または有効にします。

無効になったブレークポイントのアイコンは白、有効になったブレークポイントのアイコンは赤で表示されます。

ブレークポイントへの条件式の指定およびクリア

コードを 1 ステップずつデバッグする代わりに、ブレークポイントに条件を設定することができます。1 ステップごとに実行して各式の変数を確認するのではなく、条件が合ったときにだけ、スクリプトを中断するよう指定できます。中断したら、コードをステップごとに実行して問題が生じる位置を特定できます。

たとえば、スクリプト内の計算が間違っていて、問題が起こるのは `i==19` のときだと推測しているとしましょう。この場合、`i==18` に条件付きブレークポイントを設定しておくと、その条件が満たされた時点でデバッガによる実行が中断されます。その後、コードを 1 ステップずつ確認して問題を特定します。

ブレークポイントの条件の指定

1. ブレークポイントアイコンを右クリックし、［**ブレークポイントの編集**］を選択します。
2. ［条件］タブで［**条件**］を選択し、条件式を入力します。
3. 式が［**真の場合**］または［**変更された場合**］のどちらに中断するかを指定します。
4. ［**OK**］をクリックします。

条件の無効化または有効化

1. ブレークポイントアイコンを右クリックし、［**ブレークポイントの編集**］を選択します。
2. ［条件］タブで、［**条件**］を選択解除または選択します。

条件の削除

［ブレークポイント］タブで、ブレークポイントの条件フィールドをクリックし、Deleteキーを押します。

ブレークオプションの指定

ブレークポイントを右クリックして［**ブレークポイントの編集**］を選択すると、ブレークポイントの動作を簡単に管理できます。または、［ブレークポイント］タブでブレークポイントを選択し、  をクリックします。どちらの方法でも「ブレークポイント情報」ウィンドウが開くので、［ヒットカウント］タブと［アクション］タブで設定をカスタマイズします。

ヒットカウントの変更

ブレークポイントがヒットする回数を指定して、中断のタイミングを制御します。たとえば、条件が2回満たされたときに中断させるには、［ヒットカウント］タブで［**ヒットカウントが指定された値と等しい場合に中断:**］を選択し、「2」をタイプします。

アクションの定義

ブレークポイントがヒットしてスクリプトの実行が中断されたときに、デバッガにスクリプトを実行させることができます。このスクリプトを**アクション**といいます。［アクション］タブで、実行するスクリプトを入力します。

カーソルの位置までスクリプトを実行

右クリックして［**カーソルの位置まで実行**］を選択すると、カーソルの位置より前にあるすべての式が実行されます。現在の行までの値だけを確認したいときにこのオプションを選択します。各式が実行されたときの値を確認するには、ステップオプションを使用します。

ブレークポイントの設定に関するヒント

- あるループの中でエラーの確認が不要な場合は、ループが終わった後の位置にブレークポイントを設定します。デバッガは、ループを1行ずつ実行するのではなく、次のブレークポイントまで実行します。
- アクションをトリガしない行（コメント、改行、閉じ括弧など）にはブレークポイントを挿入しないでください。これらの行では中断できません。
- いったんブレークポイントを挿入すると、デバッガを閉じてスクリプトを編集しても、ブレークポイントが元の行番号とともに残ります。必要なら、ブレークポイントを削除し、挿入し直してください。
- ブレークポイントはデバッガのセッションを越えて保存されます。つまり、ブレークポイントリストには、現在デバッグを行っているスクリプトだけでなく、すべてのスクリプトに設定されたブレークポイントが含まれます。

- ブレークポイントは、スクリプトにではなく、デバッガセッションによって保存されます。つまり、移動または削除されたスクリプトのブレークポイントであってもリストされます。
- ［ブレークポイント］タブで をクリックすると、スクリプトが開いているかどうか、スクリプトがまだ存在するかどうかに関係なく、スクリプト内のブレークポイントがすべて削除されます。

変数の確認

変数のリストの情報は、スクリプト内で変数が使用されるたびに集められます。変数の値は、スクリプトによって変更されるたびに更新されます。スクリプトの実行中、変数がなぜその値になるのかが不明な場合は、ステップごとの変数の値をウォッチして、経過を確認できます。

また、変数に任意の値を割り当てることもできます。たとえば、`For()` ループを 1 ステップずつ実行していて、特定の回の反復処理を開始したときに何が起こっているのかということだけに関心がある場合は、反復を制御している変数にその値を割り当てることができます。ループの冒頭でその変数に値を割り当てる部分をとばし、その後、変数リストの中でその変数に値を割り当てます。そして 1 ステップずつ実行すると、ループがその位置から始まります。

変数の管理に関するヒント

- グローバルスコープを使用する複数のスクリプトを実行した場合は、グローバル変数をクリアまたは削除しましょう。そうすることで、デバッガの変数のリストが、余計なものを含まないコンパクトなものになります。削除には、`Delete Symbols()` 関数を使います。また、JMP を閉じて再起動してもグローバルの名前空間がクリアされます。
- スクリプトに使用している変数の数が多く、変数リストで見つけてウォッチするのが難しい場合、関心のある変数にウォッチを追加します。

ウォッチの操作

JMP は、セッションを越えてウォッチ変数を保持するため、JMP を閉じて再度開いても、ウォッチ変数は［ウォッチ］タブにリストされたままになります。

ウォッチの作成

- ［ウォッチ］タブで をクリックし、ウィンドウに値を入力します。
- デバッガで、ウォッチする行を右クリックして［**ウォッチに追加**］を選択し、ウィンドウに変数名を入力します。
- デバッガで、変数名またはその横にカーソルを置き（または変数名を選択し）、右クリックして［**ウォッチに追加**］を選択します。
- ［ウォッチ］タブで、空の「**変数**」フィールドに変数を入力します。

ウォッチの変更

ウォッチする変数を変更するには、［ウォッチ］タブで次のいずれかを行います。

- ウォッチを選択し、  をクリックします。
- 「変数」フィールドをクリックし、新しい変数名を入力します。

ウォッチの削除

ウォッチを削除するには、［ウォッチ］タブで次のいずれかを行います。

- ウォッチを選択し、  をクリックします。
- すべてのウォッチを削除するには、  をクリックします。

デバッガでの環境設定の変更

デバッガでの作業に適用される環境設定は、変更が可能です。［オプション］タブには、次のような設定があります。

行番号を表示　デバッガ内でスクリプトの行番号の表示/非表示を切り替えます。

行あたりに複数のステートメントがあれば中断　1行に複数の式がある場合、各式ごとにスクリプトの実行を中断します。

例外時に中断　`Try()` 関数内で例外エラーがスローされたときにスクリプトの実行を中断します。なお、`Try()` の括弧内で `Throw()` を実行させても、例外エラーをスローできます。デバッガは、式の残りを実行せずに `Throw()` で中断します。これにより、どこで問題が生じているのかが特定でき、デバッグに戻れます。

実行エラー時に中断　エラー発生時に、デバッガを停止するのではなく、スクリプトの実行を中断します。

条件に割り当て式が入力されたら警告　ブレークポイントに割り当て式を含む条件が入力されると、警告を表示します。たとえば、`x = 1` にあるブレークポイントに `x = 1` という条件を追加すると、`x` の割り当て式を確認するための警告が表示されます。

プログラム終了後デバッガに戻る　JSL プログラムの実行が終わったとき、デバッガを開いたままにします。デフォルトではオンになっていて、実行したプログラムの状態を確認することができます。

デバッガセッションの持続性

ブレークポイントやウォッチは、ユーザが削除するまで保存されたままです。タブの幅やデバッガウィンドウのサイズなど、ユーザ固有の設定は、JMP のセッションの間保持されます。

これらの設定は JMP.jdeb という名前のファイルにまとめられ、USER_APPDATA 変数で定義された場所に保存されます。

- Windows 7: "C:¥ユーザー ¥< ユーザ名 >¥AppData¥Local¥SAS¥JMP¥< バージョン番号 >¥"
- Macintosh: "/Users/< ユーザ名 >/Library/Application Support/JMP/< バージョン番号 >/"

ローカル変数やグローバル変数、名前空間の値は、通常どおり、JMP を閉じてもう一度開くとクリアされます。

スクリプトのデバッグとプロファイルの例

ここでは、変数をウォッチするためのブレークポイントの設定、式のステップイン、ステップオーバー、ステップアウト、異なるスコープや名前空間での変数のウォッチ、インタラクティブなスクリプトのデバッグ、JSL プロファイルを使ったスクリプトのデバッグの例を紹介します。

例で使っているスクリプトは「Samples¥Scripts」フォルダにあります。

ヒント：まず、デバッガの［オプション］タブで［**行番号の表示**］が選択されていることを確認してください。

ブレークポイントの使用とグローバル変数のウォッチ

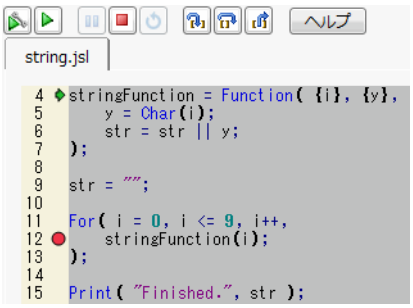
次の例では、ループ内にブレークポイントを設定し、ループの各反復における変数の変化をウォッチする方法を紹介します。

1. 「string.jsl」サンプルスクリプトを開き、［**スクリプトのデバッグ**］ボタンをクリックします。
2. 12 行目の余白をクリックしてブレークポイントを追加します（図 4.12）。

   これにより、`For()` ループの中の次の式にブレークポイントが設定されます。

   ```
   stringFunction(i);
   ```

図 4.12 ブレークポイントの設定



3. ［**実行**］をクリックします。

   最初の 2 つの式が評価されます。

   – `stringFunction` は関数として定義されます。

   – `str` は空の文字列として定義されます。

   両方の変数が、そのタイプと値とともに［グローバル］リストに追加されました。さらに、`For()` ループが、図 4.13 で示したブレークポイントのある行まで評価されました。

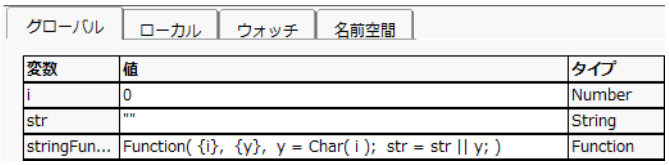
   – `i` に 0 が割り当てられました。

   – `i` が、その値とタイプとともに［グローバル］リストに追加されました。

   – `i` は 9 以下であると判断されました。

   – `stringFunction()` はまだ呼び出されていません。

図 4.13 開始時のグローバル変数

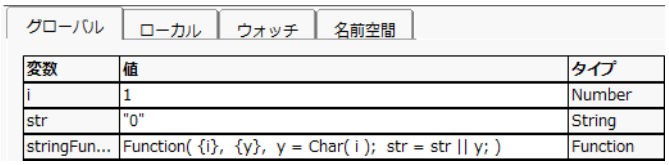


4. もう一度［**実行**］をクリックします。

   スクリプトはブレークポイントに達するまで実行されます。結果は図 4.14 のようになります。

   – `stringFunction()` が呼び出され、評価され、その後、ループに戻りました。
   – `i` がインクリメントされ、9 以下であると判断されました。
   – ［グローバル］リストで、`i` は 1、`str` は “0” に変更されました。

図 4.14 1 回目のブレークポイントでのグローバル変数



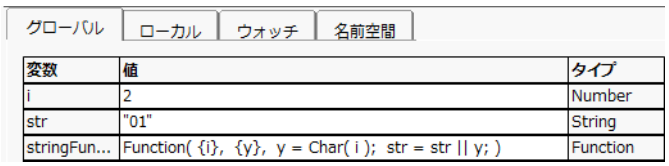
5. もう一度［**実行**］をクリックします。

   スクリプトはブレークポイントに達するまで実行されます。結果は図 4.15 のようになります。

   – `stringFunction()` が呼び出され、評価され、その後、ループに戻りました。
   – `i` がインクリメントされ、9 以下であると判断されました。

   ［グローバル］リストで、`i` は 2、`str` は “01” に変更されました。

図 4.15 2 回目のブレークポイントでのグローバル変数



続けて［**実行**］をクリックし、ループの各反復での `i` と `str` の変化をウォッチすることができます。または、［**ブレークポイントなしで実行**］をクリックしてスクリプトの実行を完了し、デバッガを終了します。

ステップイン、ステップオーバー、ステップアウト

［**ステップイン**］、［**ステップオーバー**］、［**ステップアウト**］を使うと、スクリプトに式や関数、他の JSL ファイルが含まれている場合に、より柔軟にデバッグできます。

1. 「scriptDriver.jsl」サンプルスクリプトを開き、[**スクリプトのデバッグ**] ボタンをクリックします。
2. このスクリプトはログに情報を書き込むので、メッセージを確認できるよう、デバッガの最下部にある［ログ］タブを選択します。
3. [**ステップオーバー**] をクリックします。

   スクリプトの最初の行が評価されます。
4. [**ステップオーバー**] をもう一度クリックします。

   現在の式が評価され、デバッガは次の行に移動します。この例の式は数行にわたり、変数に式が割り当てられています。
5. [**ステップオーバー**] をもう一度クリックします。

   この式は数行にわたり、変数に関数が割り当てられています。

   30行目は、先に作成されていた式を呼び出します。
6. [**ステップオーバー**] をクリックします。

   デバッガは式の中にステップインし、1行ずつ実行します。
7. 式が終了するまで [**ステップオーバー**] をクリックし続けます。

   デバッガは、式の呼び出しに続く行に戻ります。

   31行目は先に定義されていた関数を呼び出します。
8. [**ステップオーバー**] をクリックし、ステップインせずに関数を実行します。デバッガは関数全体を実行した後、関数の呼び出しに続く行に戻ります。

   33行目では別のスクリプトをインクルードしています。
9. [**ステップイン**] をクリックします。

   デバッガはインクルードされたスクリプトを別のタブで開いて待機します。
10. [**ステップオーバー**] をクリックします。

    インクルードされたスクリプトの次の行が実行されます。
11. [**ステップアウト**] をクリックします。

    デバッガはインクルードされたスクリプトの残りを実行した後、`Include()` 関数の次の行に戻ります。

異なるスコープおよび名前空間内の変数のウォッチ

デバッガウィンドウの下にあるタブでは、変数が作成され、変更される様子をウォッチできます。この例では、複数のスコープと1つの名前空間内の変数について見ていきます。

1. 「scoping.jsl」サンプルスクリプトを開き、[**スクリプトのデバッグ**] ボタンをクリックします。
2. [**ステップオーバー**] をクリックします。

   1行目（行番号4）は `Names Default To Here` をオフにします。このスクリプトを同じJMPセッションの中でもう一度実行した場合、この行はスコープをリセットし、最初に作成された変数をグローバルスコープに置きます。

3. ［**ステップオーバー**］をクリックします。

   *x*という名前のグローバル変数が作成されます。［グローバル］タブのリストに*x*が追加され、値が5、タイプが数値であることが示されます。

4. ［**ローカル**］タブを選択し、スコープのリストから［**Global**］を選択します。

   グローバル変数の*x*がここにも表示されます。

5. ［**ステップオーバー**］を2回クリックします。

   `Names Default To Here`がオンになり、スクリプトの残りがHereスコープ内に置かれます。その後、そのスコープ内に新しい変数*x*が作成されます。

   グローバル変数*x*の値は変わらないことに注意してください。

6. ［ローカル］タブのリストから［**Here**］を選択します。

   ［**Here**］の下に、ローカル変数*x*がその値やタイプとともにリストされます。

7. ［**ステップオーバー**］をクリックします。

   `Local Here`スコープが作成されます。［ローカル］のリストに2つ目のHereスコープが表示されます。

8. ［**ステップオーバー**］をクリックします。

   このHereスコープ内に新しい変数*x*が作成されます。［ローカル］タブにリストされた3つのスコープ（**Here**、**Here**、**Global**）のそれぞれを選択して、3つの異なる*x*変数を確認します。

9. ［**ステップオーバー**］をクリックします。

   デバッガのログで出力を確認します。*here:x*スコープはローカルのhereを示し、スクリプトウィンドウのhereを示すものではありません。

10. ［**ステップオーバー**］をクリックします。

    スクリプトは、ログに空の1行を挿入した後、Local Hereスコープを終了します。2つ目のHereが、その*x*変数とともに［ローカル］リストから消えます。

11. ［**ステップオーバー**］をクリックします。

    “test” という名の名前空間が、*x*という別の変数とともに作成されます。それを確認するには、［**名前空間**］タブを選択します。

12. ［**ステップオーバー**］をクリックし、ログを確認します。

13. ［**ステップオーバー**］をクリックし、デバッガを終了します。

インタラクティブなスクリプトでのデバッガを使用

スクリプトがインタラクティブな要素を作成する場合、ユーザによる操作ができるようにJMPの主インスタンスに制御が戻されます。ユーザが操作を終了したら、制御はまたデバッガに戻ります。

1. 「interactive.jsl」サンプルスクリプトを開き、［**スクリプトのデバッグ**］ボタンをクリックします。

2. ［**ステップオーバー**］を2回クリックします。

   `New Window`式が評価された後、モーダルウィンドウが開いて入力を待ちます。デバッガを移動しないと新しいモーダルウィンドウが見えないことがあります。

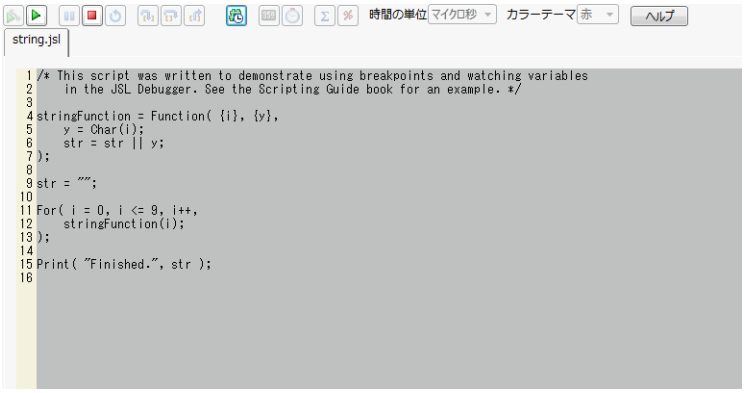
3. 「Assign X and Y」ウィンドウに2つの数値を入力し、[**OK**] をクリックします。
   制御がデバッガに戻ります。
4. [**ステップオーバー**] を3回クリックし、デバッガのログを確認します。
   ログには先ほど入力した2つの数値が表示されます。
5. [**ステップイン**] をクリックし、デバッガを終了します。

JSL プロファイルの使用

JSL プロファイルを使うと、特定の行の実行にかかった時間や、特定の行が実行された回数などがわかります。

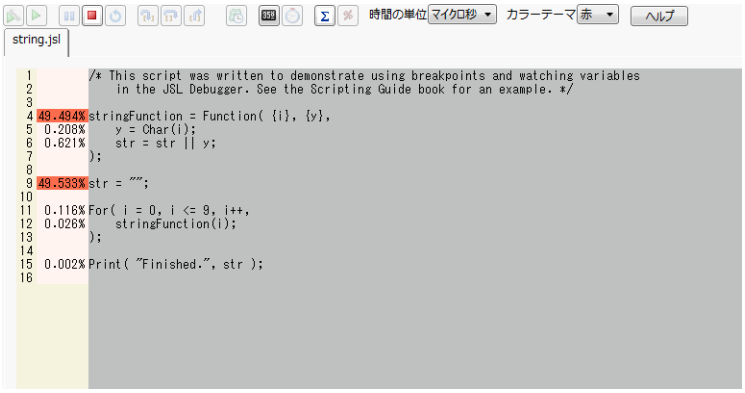
1. 「string.jsl 」サンプルスクリプトを開きます。
2. [**スクリプトのデバッグ**] ボタン をクリックします。
3. [**JSL スクリプトのプロファイル**] ボタン をクリックします。

図 4.16 開始時の JSL プロファイルウィンドウ



4. [**実行**] ボタン をクリックしてプロファイルを開始します。
   プロファイルは、あるステートメントが実行された回数、および実行にかかった時間の情報を収集します。実行時間は累積値で、JSL ステートメントが実行されるたびに記録されます。

図 4.17 スクリプトのプロファイル



左側の余白に、指定した統計量が表示されます。デフォルトでは、処理時間のパーセントが表示されます。代わりにステートメントの実行回数のパーセントを表示するには、[**総数でプロファイルを表示**] ボタン Σ をクリックします。左側の余白に表示される値は、色分けされているため、パフォーマンスの問題を引き起こしている可能性のある部分が一目で特定できます。

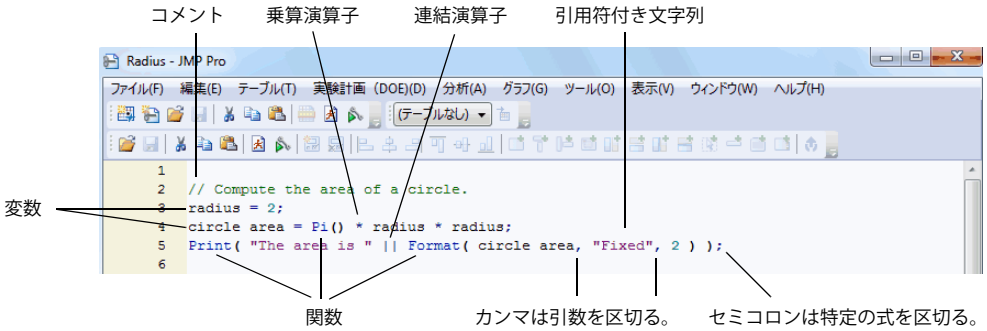
第 5 章

JSL の構成要素

JSL の基礎を学ぶ

初心者であれ上級ユーザであれ、JSL の構文と基礎を学ぶことは重要です。概念の一部（ループや変数など）は他のスクリプト言語と共通ですが、記述ルールは JSL に独特なものです。

図 5.1 JSL スクリプトの例



この章では、構文規則からファイルパス規則、条件および名前空間まで、JSL の基本概念について説明します。

目次

JSL の構文規則 81
値の区切り文字 81
数 84
名前 84
コメント 85
演算子 86
グローバル変数とローカル変数 89
ローカル名前空間 90
名前付き名前空間 90
Show Symbols、Clear Symbols、Delete Symbols 90
シンボルのロックおよびロック解除 91
名前解決のルール 92
適用範囲が指定されていない名前の解決 92
変数と列名のトラブルシューティング 96
変数とキーワードのトラブルシューティング 97
式を結合する他の方法 98
反復 99
For 99
While 100
Summation 101
Product 102
Break および Continue 102
条件付き関数 104
条件 104
Match 106
Choose 107
Interpolate 108
Step 109
不完全または一致しないデータの比較 109
問い合わせ関数 112

JSL の構文規則

スクリプト言語やプログラミング言語は、独自の構文規則を持っています。C や Java といった言語でのプログラミングの経験がある人なら、JSL を見慣れた言語と感じるでしょう。ただし、JSL では、記述ルールやスペースに関する規則が異なります。

以下の節で、基本的な要素の JSL 構文規則を説明します。

- 「値の区切り文字」（81 ページ）
- 「数」（84 ページ）
- 「名前」（84 ページ）
- 「コメント」（85 ページ）

値の区切り文字

JSL では、ワードを括弧やカンマ、セミコロン、スペース、各種演算子（＋、－など）で区切ります。この節では、区切り文字を使用するときの規則を説明します。

カンマ

カンマは、リスト内の項目、行列内の行、関数の引数といったアイテムを区切ります。

```
my list = {1, 2, 3};
your list = List(4, 5, 6);
my matrix = [3 2 1, 0 -1 -2];
If(Y < 20, X = Y);
If(Y < 20, X = Y; Z < 3, A);
TableBox(
      stringColBox("年齢",a),
      NumberColBox("度数",c),...)
```

注：複数のコマンドをつないで関数内の 1 つの引数とするには、各コマンドをセミコロンで区切ります。詳細は、「セミコロン」（82 ページ）を参照してください。

括弧

括弧は、JSL 内でいくつかの目的を持ちます。

- 括弧は式の中の演算をグループ化します。次の括弧は、`If()` 式内の演算をグループ化しています。

```
If(
   Y < 20, X = Y,
   X = 20
);
```

- 関数の引数を括弧で囲みます。次の例では、Open()関数の引数を括弧が囲んでいます。

  ```
  Open("$SAMPLE_DATA¥Big Class.jmp")
  ```

- 括弧は、引数が不要な場合でも、関数名の最後を示します。たとえば、Pi関数は引数を持ちません。しかし、JMPがPiを関数と識別するには括弧が必要です。

  ```
  Pi();
  ```

注: 括弧は必ず対で使用してください。どの (にも) が必要です。対になっていない場合はエラーとなります。

スクリプトエディタでは、括弧（丸括弧、角括弧、大括弧）の対を確認することができます。スクリプト内の任意の場所にカーソルを置き、Ctrlキーを押しながら] を押します（Macintoshの場合はcommandキーを押しながらb）。スクリプトエディタが括弧を検索し、最初の開き括弧と閉じ括弧の間のテキストを強調表示します。次に上位の括弧を強調表示するには、このプロセスを繰り返します。例については、「スクリプト作成のツール」の章の「括弧の自動マッチ」（54ページ）を参照してください。

セミコロン

セミコロンで区切った式は連続して評価され、最後の式の結果が戻されます。次のコードでは、まず変数iに0が割り当てられ、次に変数jに2が割り当てられます。

```
i=0;
j=2;
```

セミコロンは、次のIf()式のように、カンマで区切られた引数をつなぐこともできます。

```
If(x <5, y = 3; z++; ...);
```

他の言語では、式の終わりを表す文字としてセミコロンを使用します。JSLのセミコロンは、コマンドが続く可能性を示すものとして機能します。セミコロンで式を区切る方法については、「式を結合する他の方法」（98ページ）を参照してください。

スクリプトの最後、または一連の引数の最後にセミコロンをつけても問題ないので、セミコロンを終わりの文字と考えることもできます。一連のスクリプトの最後や、閉じ丸括弧または閉じ中括弧の後にセミコロンを付けることができます。実際、最後にセミコロンを付ける習慣をつけておくと、短いスクリプトを大きなスクリプトに貼り付ける際に間違いを避けることができます。

セミコロンと同じ機能を持つ関数はGlue()です。セミコロンとGlue()の詳細については、「演算子」（86ページ）を参照してください。

二重引用符

二重引用符はテキスト文字列を囲みます。スペースや大文字／小文字を含め、二重引用符内のすべてが入力したとおりに受け取られ、評価はされません。"Pi() ^ 2"を二重引用符で囲んだ場合、それは単なる文字の羅列であり、10に近い値を指すわけではないのです。

テキスト文字列の使用には十分な注意が必要です。二重引用符内のテキスト文字列は、ユーザの入力したとおり使用されるため、スペースや句読点も出力に影響します。

引用符付き文字列の中で二重引用符を使用する場合は、各二重引用符の前にエスケープシーケンス\!（バックスラッシュと感嘆符）を挿入します。たとえば、次のスクリプトを実行してウィンドウのタイトルを見てみましょう。

```
New Window( "\!" こんにちは \!" は引用符付き文字列です ",
   Text Box( Char( Repeat( "*", 70 ) ) )
);
```

表 5.1 引用符付き文字列のエスケープシーケンス

\!b	スペース
\!t	タブ
\!r	キャリッジリターンのみ
\!n	改行のみ
\!N	ホスト環境に適した改行文字を挿入[a]
\!f	改ページ（ページ区切り）
\!0	ヌル文字 **注：**ヌル文字は通常、文字列の終了とみなされるので、使用するのは危険です。ローマ字の O ではなく、数値のゼロをタイプするよう注意してください。
\!\	円記号
\!"	二重引用符

a. Macintosh では、このエスケープシーケンスは CR（キャリッジリターン、16 進数‘0D’）です。Windows では、このエスケープシーケンスは CR LF（キャリッジリターンとその後の改行、16 進数‘0D0A’）です。

あるまとまった箇所で、エスケープ文字を何回も指定しなければならない場合があります。そのような場合は、\[...]\ を使用すれば、括弧内でエスケープシーケンスが不要（使用不可能）になります。次に、二重引用符文字列の中で \[...]\ を使用している例を示します。

```
jslPhrase = " これを JSL で実行するには、次のように記述します。\[
   a = "hello";
   b = a|| " world.";
   show(b);
   ]\ そして、スクリプトを実行します。";
```

スペース

JSLでは、名前の中に空白文字を挿入することが許可されています。また、JSLワード内またはJSLワード間のスペース、タブ、改行、および空白行は無視されます。これは、ほとんどのJSLワードがユーザインターフェースから採用されており、これらのコマンドのほとんどにスペースが含まれているからです。たとえば、**「モデルのあてはめ」**プラットフォームのJSL式は、`Fit Model()`または`FitModel()`のどちらでも動作します。

演算子内のスペース、および1つの数値内の桁間のスペースは許可されません。そのようなケースではエラーが生じます。たとえば、`i++`にある2つの+の間にスペースを入れたり（`i+ +`）、数値の途中にスペースを挿入したりすることはできません（`4 3`は`43`と同じではありません）。

注：JSLで名前の中に空白文字が使えるのはなぜでしょうか。理由のひとつは、JSLのコマンドやオプションにはJMPメニューやウィンドウにあるコマンドがそのまま使われていることです。また、データテーブルの列名によくスペースが使われることももうひとつの理由です。

数

数は整数、小数、日付、時間、日付時間値として記述できます。また、Eの後ろに10のべきを付けた指数表記も使用できます。たとえば、次に挙げるものはすべて数です。

```
.   1   12   1.234   3E3   0.314159265E+1 1E-20 01JAN98
```

注：ピリオド1つだけが入力されているときは、欠測数値（または、**NAN**: not a number）です。

日付、時間、日付時間値について詳しくは、「データタイプ」の章の「日付時間の関数と形式」（123ページ）を参照してください。数値に通貨記号を付けて表示させる方法については、「データタイプ」の章の「通貨」（133ページ）を参照してください。

名前

名前は、ずばり、ものを呼ぶためのものです。たとえば、変数に3という数値を割り当てる式`a=3`では、「a」が名前です。

コマンドと関数にも名前があります。式`Log( 4 )`では、「Log」が対数関数の名前です。

名前には、次のようなルールがあります。

- 名前は**必ず**英文字または下線で始まり、その後ろに次の文字を使うことができます。
  - 英文字（a-z A-Z）
  - 数字（0-9）
  - 空白文字（スペース、タブ、改行、改ページ）
  - 数学記号（⊆など）

  - 特殊文字（アポストロフィ（'）、パーセント記号（%）、ピリオド（.）、バックスラッシュ（\）、および下線（_））
- 名前を比較するとき、空白文字（スペース、タブ、改行など）は無視され、大文字と小文字も区別されません。たとえば、Forage と for age は、空白文字と大文字／小文字の違いに関わらず、同じ名前として扱われます。

実際には、どんな文字列でも名前として入力できますが、前述のルールに従っていないものは、引用符で囲み、Name() という特殊な命令の中に入れなければなりません。たとえば、taxable income(2011) という名前のグローバル変数を指定するときは、この変数をスクリプトで使うたびに、Name() の中に入れなければなりません。

```
Name("taxable income(2011)") = 456000;
tax = .25;
Print(tax * Name("taxable income(2011)"));
   114000
```

必要ないときに Name() を使っても、問題となることはありません。たとえば、tax と Name("tax") はまったく同じものを表します。

JMP による名前の解釈については、「名前解決のルール」（92 ページ）を参照してください。

コメント

コメントは、ユーザがコードに追加するメモで、JSL プロセッサ（**構文解析プログラム**）では無視されます。コメントは、その部分のスクリプトが何をしているかなどを説明するために追加します。また、スクリプトの一部を一時的に無効にする場合にも便利です。たとえば、エラーを生じさせている可能性があるコードの前後にコメント記号を挿入し、スクリプトを再実行します。

コメントにしたいコードの前後にコメント記号を入力します。次の例では、コードの途中で /* */ に囲まれてコメントになっているところがあります。JMP は次の 2 つのスクリプトをまったく同じものとして扱います。

```
tax /*percentage*/ = .25;
tax = .25;
```

表 5.2 に、コメント記号の種類を示します。

表 5.2 コメント記号

記号	構文	説明
//	// comment	コメントの開始を示す。行頭に置く必要はありません。この記号から行末までがすべてコメントとなります。
/* */	/* comment */	コメントの開始と終了を示す。行の途中にも挿入でき、コメントの前後のスクリプトには影響が及びません。
//!	//!	//! を スクリプトの最初の行に追加すると、JMP を開いたときにスクリプトが自動的に実行される。（スクリプトエディタは開かない。）

演算子

演算子は一般的な数値演算を行う1文字または2文字の記号です。演算子にはさまざまな種類があります。

- **二項演算子**（3 + 4の+、a = 7の=のように、両側にオペランドをとるもの）
- **接頭演算子**（論理否定の!aのように、右側にオペランドを1つとるもの）
- **接尾演算子**（aをインクリメントするa++のように左側にオペランドを1つとるもの）

JSLのすべて演算子に、同じ機能を持つ関数があります。

式を簡単に記述できるよう、JSLでは特定の文字演算子（四則演算の記号など）を使用できます。これらの演算子は、関数として記述した場合と同じ意味を持ちます。たとえば、次の2つのステートメントは等価です。

```
税引き後純利益 = 純利益 - 税金;
Assign(税引き後純利益, Subtract(純利益, 税金));
```

割り当て演算を書くときは、Assign()関数か、二項演算子の=を使います。同様に、引き算を行うには、Subtract()関数またはマイナス記号を使います。どちらも、JMP内部では同じものとして扱われます。

注：通常、JSLでは空白が無視されるので、“netincomeaftertaxes”と“net income after taxes”は同じものとみなされます。ただし、2文字の演算子の文字間にスペースを**入れてはなりません**。次の演算子は、必ずスペースなしで入力してください。

||、|/、<=、>=、!=、==、+=、-=、*=、/=、++、--、<<、::、:*、:/、/*、*/

他に、一般的な演算子としてセミコロン（;）があります。セミコロンは次のような目的で使用します。

- プログラミングシーケンスで、複数の式を区切り、同時につなげる。セミコロンは最後の式の結果を戻します。つまり、a;bはGlue(a,b)と等価です。
- 式の最後。セミコロンを式の最後に付けることはできますが、他の言語のようにステートメントの終わりを表すものとして機能するわけではありません。

式には複数の演算子を含めることができます。その場合、演算子は優先度順にグループ化されていきます。たとえば、*は+より優先されます。

```
a + b * c
```

まずb * cの掛け算が行われた後、その結果がaに加算されます。

+は-より優先されます。

```
a + b * c - d
```

まずb * cの掛け算が行われた後、その結果がaに加算されます。その後、a + b * cの結果からdが引かれます。

表5.3は、演算子と、対応する関数をまとめたものです。演算子は優先順位の高い順にリストしてあります。

表 5.3 演算子、および対応する関数（優先順位の高い順）

演算子		関数構文	説明
{ }	リスト	{*a*,*b*} List(*a*,*b*)	リストを作成する。
[ ]	添え字	*a*[*b*,*c*] Subscript(*a*,*b*,*c*)	データ要素 *a* の中にある特定の要素を指定する。*a* は、リスト、行列、データ列、プラットフォームのオブジェクト、ディスプレイボックスのどれでもかまいません。
++	Post Increment	*a*++ Post Increment(*a*)	*a* に 1 を加えた値を *a* に格納する。
--	Post Decrement	*a*-- Post Decrement(*a*)	*a* から 1 を引いた値を *a* に格納する。
^	検出力	*a*^*b* Power(*a*,*b*) Power(*x*)	*a* を *b* 乗する。引数を 1 つだけ指定したときは、2 乗と解釈されます。たとえば、Power(*x*) と指定した場合は、x^2 が計算されます。
-	Minus	-*a* Minus(*a*)	*a* の符号を反転する。
!	Not	!*a* Not(*a*)	論理否定。0（偽）を 0 でない値（または真）にする。1（真）を 0 の値（偽）にする。
*	Multiply	*a**b* Multiply(*a*,*b*)	*a* に *b* を掛ける。
:*	EMult	*a*:**b* EMult(*a*,*b*)	行列 *a* と *b* を要素ごとに掛ける。（行列 *a* の各要素に行列 *b* の各要素を掛ける。）
/	Divide	*a*/*b* Divide(*a*,*b*) Divide(*x*)	Divide(*a*, *b*) は *a* を *b* で割る。 Divide(*x*) と指定した場合には、分母が x、分子が 1 と解釈され、1/*x* が求まります。
:/	EDiv	*a*:/*b* EDiv(*a*,*b*)	行列 *a* を *b* で要素ごとに割る。（行列 *a* の各要素を行列 *b* の各要素で割る。）
+	追加	*a*+*b* Add(*a*,*b*)	*a* と *b* を足す。
-	Subtract	*a*-*b* Subtract(*a*,*b*)	*a* から *b* を引く。

表5.3 演算子、および対応する関数（優先順位の高い順）（続き）

演算子		関数構文	説明
||	Concat	*a*||*b* Concat(*a*,*b*)	2つ以上の文字列または2つ以上のリストを接合する。行列を横方向に連結する。詳細については、「データ構造」の章の「リストの連結」（154ページ）または『スクリプト構文リファレンス』を参照してください。
|/	VConcat	*matrix1*|/*matrix2* VConcat(*matrix1*, *matrix2*)	行列を縦方向に連結する。（行列を横方向に連結する場合は||またはConcat()を使用。）
::	Index	*a*::*b* Index(*a*,*b*)	*a*～*b*の整数の要素を持つ行列を作成する。 （コロンは、有効範囲を指定する演算子としても使われます。その場合、:aはデータテーブル列aを、::aはJSLのグローバル変数*a*を意味します。「スコープ演算子」（93ページ）を参照。）
<<	Send	object << message Send(*object*, *message*)	オブジェクト（*object*）にメッセージ（*message*）を送る。
==	Equal	*a*==*b* Equal(*a*,*b*)...	比較のためのブール値。これらはすべて、真のときは1、偽のときは0を戻します。*a*または*b*に欠測値があると、欠測値が戻され、真も偽も評価されません。欠測値の扱いについては、「数値の欠測値」（111ページ）を参照。
!=	Not Equal	*a*!=*b* Not Equal(*a*,*b*)...	
<	Less	*a*<*b* Less(*a*,*b*)...	
<=	Less or Equal	*a*<=*b* Less or Equal(*a*,*b*)	
>	Greater	*a*>*b* Greater(*a*,*b*)	
>=	Greater or Equal	*a*>=*b* Greater or Equal(*a*,*b*)	
<=, <	Less Equal Less	*a*<=*b*<*c* Less Equal Less(*a*,*b*,*c*)	範囲チェック。真のときは1、偽のときは0を戻します。*a*か*b*のどちらかが欠測値のときは、欠測値になります。
<, <=	Less Less Equal	*a*<*b*<=*c* Less Less Equal(*a*,*b*,*c*)	

表 5.3 演算子、および対応する関数（優先順位の高い順）（**続き**）

演算子		関数構文	説明
&	And	*a*&*b* And(*a*,*b*)	論理積。両方が真のとき、真を戻します。左の値が偽の場合、右の値は評価されません。欠測値の扱いについては、「数値の欠測値」（111 ページ）を参照。
|	Or	*a*|*b* Or(*a*,*b*)	論理和。どちらかまたは両方が真のとき、真を戻します。欠測値の扱いについては、「数値の欠測値」（111 ページ）を参照。
=	Assign	*a*=*b* Assign(*a*,*b*)	*b* の値を *a* に代入する。*a* の現在の値が置き換わります。
+=	Add To	*a*+=*b* AddTo(*a*,*b*)	*b* に a を足し、結果を *a* に代入する。
-=	Subtract To	*a*-=*b* SubtractTo(*a*,*b*)	*a* から *b* を引き、結果を *a* に代入する。
*=	Multiply To	*a**=*b* MultiplyTo(*a*,*b*)	*a* に *b* を掛け、結果を *a* に代入する。
/=	Divide To	*a*/=*b* DivideTo(*a*,*b*)	*a* を *b* で割って、結果を *a* に代入する。
;	Glue	*a*;*b* Glue(*expr*, *expr*, ...)	*a*、*b* の順に実行する。

グローバル変数とローカル変数

変数は、後にスクリプト内で参照する値を保持した名前です。変数には 2 種類あります。

- **グローバル変数**は、JMP セッション内のスクリプトすべてで共有されます。
- **ローカル変数**は、それを定義したスクリプトのコンテキストだけに適用されます。特定の関数に適用される変数や、スクリプトの一部だけに適用される変数もあります。

変数の使用範囲を制限するには、**名前空間**内で変数を定義します。名前空間とは、変数、関数、その他の一意の名前の集まりです。JMP には、すべてのスクリプトでデフォルトで使用されるグローバル変数の名前空間が 1 つあります。名前を限定的な構文なしでそのまま使用すると、**範囲指定のない**変数となり、グローバル名前空間に入ります。

```
x = 1;
```

ローカル名前空間

変数をグローバル名前空間に入れると、競合が生じることがあります。2つのスクリプトで同じ名前の変数を使用した場合、最初のスクリプトの変数の値が、あとで実行されたスクリプトにより変更されてしまいます。

この問題を回避するには、各スクリプトの冒頭に次のような行を挿入します。

```
Names Default To Here(1);
```

Names Default To Here(1);関数は、スクリプトの中にある範囲指定されていない変数すべてがそのスクリプトのローカル変数であることを宣言し、グローバル変数名前空間に影響を与えないようにします。「高度な適用範囲指定と名前空間」（218ページ）に詳しい説明があります。

注：カスタムメニューおよびツールバーボタンのスクリプトについて、Names Default to Hereオプションは、デフォルトで真になっています。カスタムメニュー項目を選択するか、またはカスタムツールバーボタンをクリックした際に実行されるスクリプトは、グローバル変数に影響を与えません。

名前付き名前空間

変数を特定の名前空間内に作成することもできます。次の例は、aaという名前空間の中に変数xを作成します。

```
aa:x = 1;
```

ローカル変数の前にLocal()関数を置くという方法もあります。次の例では、aもbもローカル変数です。

```
Local( {a = 1, b}, ... )
```

スコープ演算子も、グローバル変数とローカル変数を区別します。詳細は、「名前解決のルール」（92ページ）を参照してください。

以下の節で、変数の管理に役立つ関数を紹介します。

Show Symbols、Clear Symbols、Delete Symbols

Show Symbols()関数は、グローバルとローカルの両方で定義されている変数と名前空間を、現在値とともにリストします。以下は、ログに表示されたShow Symbols()メッセージの結果の例です。

```
Show Symbols();
   // Here
   a = 5;
   b = 6;
   // 2 Here

   // Global
   c = 10;
   // 1 Global
```

ヒント：JSL デバッガでも、変数と名前空間の値を表示することができます。詳細については、「スクリプト作成のツール」の章の「スクリプトのデバッグ／プロファイル」（63 ページ）を参照してください。

Clear Symbols() 関数は、グローバルとローカルの両方で定義されている変数の設定値をクリアします。たとえば、Clear Symbols の後に Show Symbols を使うと、変数が空になっていることがわかります。

```
Clear Symbols();
Show Symbols();
   // Here
   a = Empty;
   b = Empty;
   // 2 Here

   // Global
   c = Empty;
   // 1 Global
```

注：以前のバージョンで使用されていた Show Globals() 関数と Clear Globals() 関数は、新しい Show Symbols() 関数と Clear Symbols() 関数の別名です。

すべてのグローバル変数および名前空間を削除するには、Delete Symbols() 関数を使用します。次のスクリプトの最後にある Show Symbols() が実行されても、ログには何も表示されません。すべての変数がメモリから完全に削除されるためです。

```
Delete Symbols();
Show Symbols();
```

すべての名前空間内の変数をリストするには、Show Namespaces() を使用します。特定の名前空間のみを削除するには、ns << Delete を使用します。Clear Symbols() と Delete Symbols() は、名前空間への参照を含む変数をクリアまたは削除しますが、各名前空間内の変数をクリアまたは削除するわけではありません。範囲指定のない変数については、「名前解決のルール」（92 ページ）を参照してください。

注：Clear Symbols() と Delete Symbols() は、現在使用されているすべてのスクリプトを中断します。これらの関数は、プログラミング環境やデバッグ環境に非常に便利ですが、配布予定のスクリプトには含めないようにしてください。スクリプトに Names Default To Here(1) を含めた場合、グローバルシンボルのクリアや削除は必要ありません。

シンボルのロックおよびロック解除

変数が変更されるのを回避するためにロックしたい場合、Lock Symbols() 関数を使用します。（Lock Globals() は別名です。）

```
Lock Symbols (name1, name2, ...)
```

ロックを解除してグローバル変数の変更を可能にするには、Unlock Symbols() 関数を使用します。（Unlock Globals() は別名です。）

```
Unlock Symbols (name1, name2, ...)
```

この2つのコマンドの主な用途は、変数が誤って変更されるのを防ぐことです。たとえば、別のスクリプトで使用している変数がClear Symbols()でクリアされると困る場合に、その変数をロックしておきます。

注：Lock Symbols()を、名前空間をロックする目的で使用することはできません。ns << Lockを使用してください。

名前解決のルール

次の種類のオブジェクトは、名前で識別できます。

- データテーブル内の列とテーブル変数
- セッションの間、値を保持するグローバル変数
- スクリプトで記述することが可能なオブジェクトタイプ
- 計算式内の引数とローカル変数

オブジェクトを参照するには、ほとんどの場合、オブジェクトの名前を直接使うことができます。次の例を見てみましょう。

```
比率 = :Name("身長(インチ)") / :Name("体重(ポンド)");
```

スクリプトの複雑さにもよりますが、「比率」が変数で、「身長(インチ)」と「体重(ポンド)」がデータテーブルの列名だというのはすぐわかりますね。では、意味がはっきりしない場合はどうでしょうか。「比率」をグローバル変数や列名として使用しているスクリプトもあるかもしれません。

適用範囲が指定されていない名前の解決

JMPは**名前の解決**を使ってオブジェクト名を解釈します。適用範囲が明示されていない名前には、次の規則が順に適用されます。

1. 名前がオブジェクトに送るスクリプトの一部である場合、それは通常、オブジェクト内のオプションまたはメソッドの名前です。たとえば、Show Points()はBivariateオブジェクト内のオプションの1つです。

   ```
   obj = Bivariate(y(:Name("体重(ポンド)")), x(:Name("身長( インチ)")));
   obj << Show Points(1);
   ```

2. 接頭部に:スコープ演算子が付いている名前の場合、GlobalやHereなどの名前空間の中を検索します。
3. 後ろに1組の丸括弧()がついている名前の場合、（ユーザ定義の関数ではなく）ビルトイン関数として検索します。
4. 接頭部に:スコープ演算子が付いている名前の場合、データテーブル列またはテーブル変数として検索します。
5. 接頭部に::スコープ演算子が付いている名前の場合、グローバル変数として検索します。

6. ローカル変数として検索します。
7. プラットフォーム起動名（Distributionや Bivariateなど）として検索します。
8. 割り当て式の左辺（L-value）として使用されている名前で、かつ、スクリプトの冒頭に Names Default To Here(0)がある場合は、グローバル変数を作成して使います。

例外

- 一部の名前は、データテーブル、データ列、プラットフォームなどのオブジェクトを参照する変数であり、値の取得や設定には使用されません。これらの名前は解決されず、渡されます（文字どおり解釈される）。
- 関数の定義、列計算式、および「非線形回帰」プラットフォームの計算式において、スコープは列内のすべての行で同じです。
- 名前が、閉じているデータテーブル内の列を直接参照している場合、その名前はテーブルが再度開いたときに該当の列に対して解決されます。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
Show( Name("体重(ポンド)") << Get As Matrix ); // "体重(ポンド)"は列名として解決される
Close( dt, NoSave );
Show( Name("体重(ポンド)") << Get As Matrix ); // "体重(ポンド)"は解決されない
/* データテーブルを再度開く */
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
Show( Name("体重(ポンド)") << Get As Matrix ); // "体重(ポンド)"は列名として解決される
```

ただし、次の例では、変数をデータテーブルの2番目のインスタンスとして解決しません。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
col = Column( dt, 1 ); // colはColumn( "体重(ポンド)" )
Close( dt, NoSave );
/* データテーブルを再度開く */
dt = Open( "$SAMPLE_DATA\Big Class.jmp " );
Show( col << Get As Matrix ); // 最初のデータテーブルへの参照はもう存在しない
```

以下の節で、データテーブルの列の名前がどのように解決されるかを説明します。名前の解決について詳しくは、「プログラミング手法」の章の「高度な適用範囲指定と名前空間」（218ページ）を参照してください。

スコープ演算子

スコープ演算子は、名前が曖昧な場合（たとえば、変数と列名の両方を指している場合）の解決の助けとなります。

次の例では、接頭部の二重コロン（::）により、zがグローバル変数であることを示しています。接頭部の一重コロン（:）は、xとyが列名であることを示します。

```
::z = :x + :y;
```

ヒント：Names Default to Here(1) 関数も名前の解決に影響を与えます。詳細は、「プログラミング手法」の章の「Names Default To Here とグローバル関数」（220 ページ）を参照してください。

スコープ演算子と同様に使用できる JSL 関数が 2 つあります。表 5.4 は、その関数と構文を示したものです。

表 5.4 スコープ演算子

演算子と等価の関数	関数構文	説明
: As Column	:name dt:name As Column(*dt, name*)	*name* をデータテーブル列として評価させる。オプションのデータテーブル参照引数 *dt* は、現在のデータテーブルを設定します。例については、「適用範囲が指定された列名」（94 ページ）を参照。
:: As Global	::name As Global(name)	*name* をグローバル変数として評価させる。 **注：**二重コロンは、範囲を表す二項演算子としても使われることに注意。

適用範囲が指定された列名

列名の適用範囲を指定すれば、変数名の競合を簡単に回避できます。スコープ演算子を使い、スクリプト内の名前を列への参照として評価させます。

1. 接頭部のコロン（:）は、名前がグローバル変数ではなく、テーブル列またはテーブル変数を参照することを意味します。接頭部のコロンは、現在のデータテーブルの内容を参照します。

   ```
   :年齢;
   ```

2. 二項演算子のコロン（:）は、この考え方を拡張して、データテーブル参照を使って、どのデータテーブルの列なのかを指定します これは、スクリプト内で複数のデータテーブルが参照されている場合に特に重要です。

   次の例では、dt 変数が「Big Class.jmp」へのデータテーブル参照を設定します。二項演算子のコロンは、データテーブル参照と「年齢」列を分離します。

   ```
   dt = Data Table("Big Class.jmp");
   dt:年齢 // コロンは二項演算子。
   ```

   As Column の場合も同じ結果となります:

   ```
   dt = Data Table("Big Class.jmp");
   As Column(dt, 年齢);
   ```

したがって、「Big Class.jmp」が開いている場合に限り、次の式も等価です。

```
:年齢;
As Column(dt, 年齢);
dt:年齢;
```

Column 関数は列を特定する目的で使用することもできます。「Big Class.jmp」では、以下の式はすべてテーブル内の 2 番目の列である「**年齢**」を参照します。

```
Column(" 年齢 ");
Column(2);
Column(dt, 2);
Column(dt, " 年齢 ");
```

列名と変数名の競合を回避する

列名と同名の変数があるスクリプトを実行すると、「無効な行番号です。」というエラーが発生します。この問題を防ぐには、列名と変数名が重複しないように注意するか、次のように名前の適用範囲（スコープ）を指定します。

- 次の例では、グローバル変数と列が、どちらも「年齢」という名前です。1行目では、「年齢」の適用範囲をグローバル変数として指定します（演算子 `::` がそれを示しています）。これでスコープの指定されていない「年齢」はグローバル変数として解決されるため、2行目では列名の**「年齢」**の適用範囲だけを指定しています。

  ```
  :: 年齢 = [];
  年齢 = : 年齢 << Get As Matrix;
  ```

- 次の例では、列名の**「年齢」**からグローバル変数の「年齢」を作成しています。グローバル変数の「年齢」は適用範囲を指定する必要がありますが、列名の**「年齢」**の範囲を指定する必要はありません。

  ```
  :: 年齢 = 年齢 << Get As Matrix;
  ```

  変数「年齢」が決まったので、スクリプト全体での列名の**「年齢」**のみ、適用範囲を指定します。

JMP は、列にある連続したセルすべてについて列計算式を評価します。そのため、通常は列名の適用範囲を指定する必要はありません。ただし、計算式内に割り当てられた変数が列と同名の場合、その列名の適用範囲を指定する必要があります。詳細については、「プログラミング手法」の章の「適用範囲が指定された名前」（221 ページ）を参照してください。

適用範囲が指定されていない列名

適用範囲が指定されていない名前に対して値を設定したり、値を取得する場合があります。次のような場合、JMP は名前を（グローバル変数ではなく）データテーブル内の列として解決します。

- その名前をすでに使用しているグローバル変数、ローカル変数、または引数が存在しない
- かつ、コンテキスト内のデータテーブルに同名の列がある
- かつ、次のどちらかである
  - 現在の行の番号が正の値に設定されている
  - 名前に添え字がある（たとえば、`:Name(" 体重 ( ポンド )")[1]` の添え字 `[1]` は、**「体重 (ポンド)」**列の最初の値を選択する）

データテーブルにその名前のテーブル変数がある場合、テーブル変数が優先します。その他のすべてのケースでは、名前はグローバル変数、ローカル変数、または引数に結び付けられます。グローバル変数とローカル変数についての詳細は、「グローバル変数とローカル変数」（89 ページ）を参照してください。

例外

列計算式と非線形回帰の計算式では、列名がグローバル変数よりも優先します。

現在のデータテーブル行の設定

デフォルトでは、現在の行の番号は0で、これは無効な数値です。そのため、次の式はグローバル変数**ratio**に欠測値を割り当てます。

```
ratio = :Name(" 身長 ( インチ )") / :Name(" 体重 ( ポンド )");
```

Row()関数を使って行番号を指定します。次の例は、行を3に設定します。その行の身長を体重で割り、その結果をグローバル変数**ratio**に割り当てます。

```
Row() = 3;
ratio = :Name(" 身長 ( インチ )") / :Name(" 体重 ( ポンド )");
```

別の方法としては、添え字を使って行番号を指定する方法があります。次の式は、行3の身長を行4の体重で割ります。

```
ratio = :Name(" 身長 ( インチ )")[3] / :Name(" 体重 ( ポンド )")[4];
```

スクリプトがデータテーブルの各行に対して実行される場合、行番号の指定は必要ありません。次の例は、**「比率」**列を作成します。各行の身長を体重で割っています。

```
New Column(" 比率 ");
For Each Row(: 比率 = :Name(" 身長 ( インチ )") / :Name(" 体重 ( ポンド )"));
```

JMPは計算式を評価し、列全体にわたって反復して事前評価された統計量を計算します。このような場合も、行番号の指定は必要ありません。(事前計算された統計量は、「データテーブル」の章の「事前計算される統計量」(343ページ)で説明するように、データテーブルから計算される単一の数値です。)

変数と列名のトラブルシューティング

As Name()を使用して列名を参照し、Names Default To Here(1)を設定した場合、JMPは変数参照を戻します。そして、その参照は標準の参照ルールを使って処理されます。

次の例では、Here:スコープに**「身長(インチ)」**変数はありません。そのため、JMPはエラーを戻します。

```
Names Default To Here( 1 );
Open( "$SAMPLE_DATA\Big Class.jmp" );
As Name( " 身長 ( インチ )" )[3];
   As Name( " 身長 ( インチ )" )[/*###*/3];
```

この問題を回避するには、次のいずれかの方法を使用します。

- As Name()ではなくAs Column()を使用します。

```
Names Default To Here( 1 );
Open( "$SAMPLE_DATA\Big Class.jmp" );
As Column( " 身長 ( インチ )" )[3];
```

- As Name() を使って明示的に「**身長(インチ)**」の適用範囲を指定します。

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt:(As Name( "身長(インチ)" ))[3];
```

これらのスクリプトは、「Big Class.jmp」の3行目の「**身長(インチ)**」の値である55を戻します。

変数とキーワードのトラブルシューティング

変数と引用符なしのキーワードが同じ名前を持っている場合も、名前解決のエラーが発生する可能性があります。たとえば、<<Preselect Role() の1つの引数が「Y」であるとします。スクリプト内でYを変数としても使用している場合は、この引数を引用符で囲む必要があります。

名前解決に関するよくある質問

適用範囲（スコープ）を指定する必要がありますか。

はい。迷った場合は、適用範囲（スコープ）を指定してください。特に、多くの人がさまざまなデータテーブルとともに使う可能性があるスクリプトでは、必ず適用範囲を指定してください。気づかないで、同じ名前を2つのコンテキストで使用している場合（たとえば、データ列と同じ名前のグローバル変数を定義する、など）があるからです。

そのようなスクリプトを書く場合は、適用範囲と名前空間を明確に指定してください。詳細については、「プログラミング手法」の章の「高度な適用範囲指定と名前空間」（218ページ）を参照してください。

接頭スコープ演算子は、いったん解決された後は、実行時オーバーヘッドがかかりません。二項スコープ演算子は、常に、実行時オーバーヘッドがかかります。

名前（グローバル変数）によって参照される列と、直接列名を指定する場合の違いは何ですか。グローバル変数によって列を参照する場合、どのようにその列のセルに値を割り当てるのでしょうか。

列を参照すると、列の特定の属性を変更するメッセージや、その値（セルの色付けや計算式の設定など）にアクセスするメッセージを送ることができます。

グローバル変数に列が割り当てられている場合は、添え字を使って列内のセルに値を割り当てます。列名「**身長(インチ)**」が変数xに割り当てられているとします。

```
x = Column("身長( インチ)");
```

「**身長(インチ)**」列の3行目に、次のように値を割り当てます。

```
x[3] = 64 // 身長の 3 行目の値を 64 に設定する
```

注：JSL スクリプトの現在行は、行を選択することや行にカーソルを置くことで**決定されるのではありません**。現在行は、デフォルトではゼロ（行なし）になっています。Row() を使って（例：Row() = 3）現在行を設定できますが、このような設定はそのスクリプトが実行されている間だけ有効で、スクリプトの実行が終了すると、Row() はデフォルトの値ゼロに戻ります。そのため、一度にスクリプトすべてを実行したときと、一度に数行ずつスクリプトを実行したときとでは、異なる結果になる場合があることに注意してください。

スクリプトの現在の行を設定するもう1つの方法は、For Each Row() で囲む方法です。これは、現在のデータテーブルの行ごとにスクリプトを1回ずつ実行します。例として、「条件」（104ページ）を参照してください。データテーブルの操作方法については、「データテーブル」（257ページ）の章を参照してください。

スコープ演算子は、ずっとその名前に有効ですか。

はい。いったんスコープ演算子を名前に使うと、以後その名前が出てきた場合、名前は常にスコープ演算子として解決されます。たとえば、スクリプトに age という名前の列と変数があるとします。スクリプトの冒頭で、スコープ演算子 : : を使ってグローバル変数 age を宣言したら、そのスクリプト内で age は常にグローバル変数として解釈されます。「age」列内の値は変数によって影響を受けません。

```
::age=70;
Open("$SAMPLE_DATA¥Big Class.jmp");
age=5; // age はグローバル変数。
Show(age); // age はまだグローバル変数。
```

スコープ演算子を使う場合、":" と "[]" でどちらの方が優先されますか。

スコープ演算子（:）は、添え字（[]）より前に評価されます。これは、次の2つの行が等価であることを示します。

```
dataTable:colName[i]
(dataTable:colName)[i]
```

式を結合する他の方法

式は、同じ行であっても異なる行であっても、セミコロンを使って分離できます。JMP は、それらの式を順番に評価し、最後の結果を戻します。ここに、まず a を2に設定し、続いて b に3を設定する式があります。

```
a = 2;
b = 3;
```

セミコロンは2つの式を結合し、最後の式の値を戻します。つまり、x = (a = 2; b = 3) の場合、x の値は3です。

Glue() 関数は最後の式の結果を戻します。この関数を使うと、セミコロンを使ったときと同じ結果になります。次の式は、どちらも3を戻します。

```
Glue(a=2, b=3)
a = 2; b = 3
```

First() 関数も、各引数を順々に評価しますが、最初の式の結果を戻します。次の式は 2 を戻します。

```
First(a=2,b=3)
```

例

次のスクリプト内の First() は何を戻すでしょうか。

```
x = 1000;
First(x, x=2000)
```

First 関数は x の値（1000）を戻します。その後、x に 2000 が割り当てられます。

反復

JSL では、指定の条件に従ってアクションを繰り返す（**反復**する）関数として For()、While()、Summation()、Product() が用意されています。

注：For Each Row() という関数は、データテーブルの行に対してアクションを反復します。例については、「条件」（104 ページ）を参照してください。テーブルの行にわたる反復については、「データテーブル」の章の「データ値にアクセスするその他の方法」（339 ページ）にも説明があります。

For

For() 関数は、カンマで区切った 4 つの引数をとります。最初の 3 つの引数はループを何回繰り返すかを指定するためのもので、4 つ目の引数が繰り返し実行する内容です。

次に、For() の基本的な構文を示します。

```
For(initialization, while, iteration, body);
```

たとえば、次のスクリプトは 0 ～ 20 までの整数を合計しています。

```
s = 0;
For(i = 0, i <= 20, i++, s += i);
```

このスクリプトは次のように機能します。

s = 0;	s 変数を 0 に設定する。後でこの変数に合計が代入されます。
For(	For() ループを開始する。
i = 1,	ループの制御カウンタの変数（i）を 0 に設定する。この式は一度だけ実行されます。
i <= 20,	ループが開始されるたびに i を 20 と比較する。i が 20 以下である限りループの評価を繰り返し、i が 20 を超えるとループを抜けます。

`i++,`	ループの最後に`i`を1だけ増やす。このステップはループのbody部分（次の行）の評価後に実行されます。
`s += i`	ループのbody部分を評価する。`i`の値を`s`に加えます。bodyを実行した後、*i*の値を増やします（前の行）。
`);`	ループを終了する。

無限ループ

常に真と評価される`For`ループは無限ループを生成し、終わることがありません。スクリプトの実行中にループを止めるには、WindowsではEscキーを、Macintoshではcommandキーとピリオドキーを同時に押します。

JSLのForループとCおよびC++

JSLの`For()`ループは、C（およびC++）の場合と句読法は異なりますが、同様に機能します。

ヒント：Cをご存知のユーザは、セミコロンとカンマの使い方の違いに注意してください。JSLの場合、`For()`は、カンマで引数を区切り、セミコロンで式を結合する関数です。Cの場合、`for`は、セミコロンで引数を区切り、カンマで式を結合する特別な節です。

While

関連する関数に`While()`があります。この関数は、条件を繰り返し検定し、条件が真であればbodyのスクリプトを評価します。構文は次のとおりです。

```
While(condition, body);
```

たとえば、次に示す2つのプログラムは、x (287)以上かつ最小の2のべき乗を見つける`While()`ループを使っています。どちらのプログラムでも結果は512になります。

```
x = 287;

// ループ1:
y = 1;
While(y < x, y *= 2);
Show(y);

// ループ2:
k = 0;
While(2 ^ k < x, k++);
Show(2 ^ k);
```

このスクリプトは次のように機能します。

x = 287;	x を 287 に設定する。
// ループ 1	
y = 1;	y を 1 に設定する。
While(	While() ループを開始する。
y < x,	y が x より小さい限りループの評価を繰り返す。
y *= 2	1 に 2 を掛け、結果を y に割り当てる。その後、y が 287 より大きくなるまで、ループを繰り返します。
);	ループを終了する。
Show(y);	y (512) の値を表示する。
// ループ 2	
k = 0;	k を 0 に設定する。
While(	While() ループを開始する。
2 ^ k < x,	2 を k の指数でべき乗し、結果が 287 より小さい限り評価を繰り返す。
k++	k を 1 増やす。その後、ループを 2^k が 287 より大きくなるまで繰り返します。
);	ループを終了する。
Show(2 ^ k);	2^k の値（512）を表示する。

For() ループと同様、常に真と評価される Which() ループも、終わりのない無限ループを生成します。スクリプトの実行中にループを止めるには、Windows では Esc キーを、Macintosh では command キーとピリオドキーを同時に押します。

Summation

Summation() 関数は、i の値すべてについて body 部分の式の結果を加算します。構文は次のとおりです。

```
Summation(initialization, limitvalue, body);
```

たとえば、次のような設定が行えます。

```
s = Summation(i = 1, 10, i);
```

は、1+2+3+4+5+6+7+8+9+10 の結果である 55 を戻します。

このスクリプトは次のように機能します。

s =	s変数に関数の値を設定する。
Summation(	Summation()ループを開始する。
i = 1,	iを1に設定する。
10,	iの限界を10に設定する。
i	1から10までのiのすべての値を合計し、結果の55を戻す。
);	ループを終了する。

この動作は計算式エディタのΣと同じです。

```
Summation(i = 1, N Row(), x ^ 2);
```

計算式エディタでこのJSLと等しいものは、次のようになります。

$$\sum_{i=1}^{\text{NRow}} x^2$$

Product

Product()関数は、body部分の式の結果を加算するのではなく乗算することを除けばSummation()と同じように動作します。構文はSummation()と同じです。たとえば、次のような設定が行えます。

```
p = Product(i = 1, 5 , i);
```

は、1*2*3*4*5の結果である120を戻します。

この例では、iの開始値が1、上限が5で、iの5までの整数がすべて掛け合わされます。

計算式エディタでこのJSLと等しいものは、次のようになります。

$$\prod_{i=1}^{5} i$$

BreakおよびContinue

Break()とContinue()を使うと、ループをさらに制御することができます。Break()は、ループを直ちに中断し、ループの後に続く式を実行します。Continue()は、Break()を少し柔和にしたもので、現在のループの反復を直ちに中断し、次の反復を続行します。

Break

Break()は通常、条件文の中で使用します。たとえば、次のような設定が行えます。

```
For(i = 1, i <= 5, i++,
   If(i == 3, Break());
   Print("i=" || Char(i));
);
```

の結果は、次のようになります。

```
"i=1"
"i=2"
```

このスクリプトは次のように機能します。

For(	For() ループを開始する。
i = 1,	i を1に設定する。
i <= 5,	i が5以下である限りループの評価を繰り返す。
i++,	i を1だけ増やす。このステップは IF ループの評価後に実行されます。
If(	If() ループを開始する。
i == 3, Break()	i が3に等しい場合、ループを中断する。
);	ループを終了する。
Print(	i が3に等しい場合、Print() ループを開く。
"i="	文字列 "i=" をログに出力する。
||	後続の値を、"i=" と同じ行に出力する。
Char(i));	i の値をログに出力する。 その後、i の値が5以下の間は For() ループを繰り返し、i が3より小さい場合は中断して出力します。
);	ループを終了する。

Print() の後に If() 式と Break() 式が続く場合、"i=3" が出力された後にループが中断するので、1～3の i の値が出力されることになります。

```
"i=1"
"i=2"
"i=3"
```

Continue

Continue() も、Break() と同様に条件式の中で使用されます。たとえば、次のような設定が行えます。

```
For(i = 1, i <= 5, i++,
   If(i < 3, Continue());
   Print("i=" || Char(i));
);
```

の結果は、次のようになります。

```
"i=3"
"i=4"
"i=5"
```

このスクリプトは次のように機能します。

For(	For() ループを開始する。
i = 1,	i を1に設定する。
i <= 5,	1が5以下であると評価する。
i++,	i を1だけ増やす。このステップはIFループの評価後に実行されます。
If(	If() ループを開始する。
i < 3, Continue()	i を1と評価し、i が3より小さい限り続行する。
);	If() を終了する。
Print(	i が3以上になったら、Print() ループを開く。
"i="	文字列"i=" をログに出力する。
||	後続の値を、"i=" と同じ行に出力する。
Char(i));	i の値をログに出力する。 その後、i の値が5以下の間は For() ループを繰り返し、i が3以上の場合は続行して出力します。
);	ループを終了する。

条件付き関数

JSLには、If()、Match()、Choose()、Interpolate()、Step() という、条件付き処理を行う5つの関数があります。

条件

If() 関数は、条件（condition）が真（0でない非欠測値）であると評価された場合に、結果（result）のステートメントを戻します。真でない場合には、先に進み、次に条件が真と評価されたときの結果を戻します。

構文は次のとおりです。

```
If (condition, result1, result2)
```

たとえば、次のスクリプトは年齢が12より小さい場合、"若い"を戻します。そうでない場合は、"気持ちが若い"を戻します。

```
If ( 年齢 < 12,
   " 若い ",
   " 気持ちが若い "
);
```

複数の条件と結果をつなげることもできます。構文は次のとおりです。

```
If (condition1, result1,
   condition2, result2,
   ...,
   resultElse);
```

この例では、条件1（condition1）が真でない場合、関数は真の条件を見つけるまで評価を続けます。そして、真である条件の結果を戻します。

すべての条件が偽のときは、最後の結果を戻します。値がない場合は欠測値を戻します。そのため、式の最後にデフォルトの結果を含めておくことが大切です。次に、「Big Class.jmp」の性別の略称をコード変更する例を見てみましょう。

```
For Each Row( 性別 =
   If(
      性別 == "F", " 女性 ",
      性別 == "M", " 男性 ",
      " 不明 ");
);
```

このスクリプトは次のように機能します。

`For Each Row( 性別 =`	テーブルの各行で**「性別」**の列の値を変更するよう指定する。
`If(`	`If()` ループを開始する。
`性別 == "F", " 女性 ",`	**「性別」**の値が「F」であれば、その値を「女性」に変更する。
`性別 == "M", " 男性 ",`	**「性別」**の値が「M」であれば、その値を「男性」に変更する。
`" 不明 ");`	どちらの条件も真でない場合は値を「不明」に変更する。この部分が指定されず、**「性別」**が欠測値になっていた場合は、欠測値を戻します。
`);`	ループを終了する。

結果の式にアクションや割り当てを挿入することもできます。次のスクリプトは、最初の条件（`y < 20`）が偽なので、`x` に20を割り当てます。

```
y = 25;
z = If( y < 20, x = y, x = 20 );
```

注：等しいことを調べる場合には、`=` ではなく、`==` を使います。`If` 関数に *name=value* といった引数が付いている場合は、値を調べるのではなく割り当てます。

Match

Match() 関数を使うと、比較対象となる値をすべて手で入力しなくても、等しいかどうかの比較を繰り返すことができます。構文は次のとおりです。

```
Match(x, value1, result1, value2, result2, ..., resultElse)
```

たとえば次のスクリプトは、「Big Class.jmp」の性別の略称をコード変更します。

```
For Each Row ( 性別 =
   Match(
      性別 ,
      "F", " 女性 ",
      "M", " 男性 ",
      " 不明 ");
   );
```

このスクリプトは次のように機能します。

For Each Row(性別 =	テーブルの各行で**「性別」**の列の値を変更するよう指定する。
Match(	Match() ループを開始する。
性別 ,	**「性別」**を Match の引数に指定する。
"F", " 女性 ",	値が「F」であれば、「女性」に置き換える。
"M", " 男性 ",	値が「M」であれば、「男性」に置き換える。
" 不明 ");	F と M のどちらでもない場合は、「不明」に置き換える。
);	ループを終了する。

この Match() の例は、「条件」（104 ページ）の例を単純にしたものです。Match() の利点は、比較の変数を条件式ごとに繰り返す必要がなく、一度定義するだけでよいことです。欠点は、If とは異なり、演算子のある式が使えないことです。Match() 式で性別 == "F" という引数を使うと、エラーになります。

条件と結果のグループが多いほど、Match() が本領を発揮します。次のようなスクリプトに If() を使ったとすると、もっと多くの行が必要になります。

```
dt=open("$SAMPLE_DATA¥Travel Costs.jmp");
For Each Row( 予約した曜日 =
   Match( 予約した曜日 , "Sunday", "SUN", "Monday", "MON", "Tuesday", "TUE",
   "Wednesday", "WED", "Thursday", "THU", "Friday", "FRI", "Saturday", "SAT", "Not
   Specified");
);
```

条件と結果のデータタイプに気をつけてください。前述の例では、条件も結果も文字データです。データタイプが一致しないと、列のデータタイプが自動的に変更されてしまうため、予期しない結果になります。

次のスクリプトでは、列のデータタイプが、最初のセルのデータタイプに基づいて数値から文字に変更されます。最初の値「12」は「Twelve」に置き換わり、残りのセルには「Other」が挿入されます。

```
dt=open("$SAMPLE_DATA¥Big Class.jmp");
For Each Row( 年齢 =
   Match( 年齢 , 12, "Twelve", 13, "Thirteen", 14, "Fourteen", 15, "Fifteen", 16,
   "Sixteen", "Other");
);
```

比較する値が1、2、3・・・という整数値の場合には、Choose() を使用するとさらに入力する引数を少なくできます。詳細については、「Choose」（107 ページ）を参照してください。

Choose

Choose() 関数は、引数を整数と比較する場合に Match() よりも短く記述することができます。構文は次のとおりです。

```
Choose(expr, result1, result2, result3, ..., resultElse)
```

データテーブルの列に、1から7までの数値が入っているとします。次のスクリプトは、最初のセルに数字の1があるときに x="低" を戻します。

```
x = (Choose(group,
   "低",
   "中",
   "高",
   "不明"
   );
);
Show(x);
```

このスクリプトは次のように機能します。

x =	x 変数を作成する。
Choose(	Choose() ループを開始する。
group,	group の値を評価する。
"低",	group の値が1の場合、"低" を戻す。
"中",	group の値が2の場合、"中" を戻す。
"高",	group の値が3の場合、"高" を戻す。
"不明"	それ以外の場合は、"不明" を戻す。
);	ループを終了する。

);	x変数を閉じる。
Show(x);	xの値を戻す。

式が範囲外の整数（たとえば、置換用の値が4つしかないのに7が入力されたとき）を評価した場合は、最後の結果が戻されます。前述の例では、"不明"が戻されます。

If()およびMatch()では、Choose関数と同じ結果を得るためにより多くのコードが必要になります。

```
if(group==1, "低", group==2, "中", group==3, "高", "不明");
match(group, 1, "低", 2, "中", 3, "高", "不明");
```

注：式内のデータタイプが一致しないときは、列のデータタイプが自動的に変更されます。

Interpolate

Interpolate()関数は、(x1, y1)と(x2, y2)の2点間で、与えられたx値に対応するy値を戻します。値は線形補間されます。Interpolate()は、データ点間の欠測値の計算に使用できます。

各データ点は、次のように列挙することにより指定できます。

```
Interpolate(x, x1, y1, x2, y2, ...)
```

また、x値とy値が含まれた行列として指定することもできます。

```
Interpolate(x, xmatrix, ymatrix)
```

20～25歳の人の身長を記録したデータセットがあるとします。ただし、23歳の人のデータがありません。23歳の人の身長を推定するには、補間を使用します。次の例は、評価したい値（23歳）、年齢（20～25歳）の行列、そして身長（インチ; 59～75）の行列を示しています。

```
Interpolate(23, [20 21 22 24 25], [59 62 56 69 75]);
```

この式は、次の値を戻します。

```
62.5
```

62.5という値は、23がx値の22と24の真ん中であるように、y値の56と69の真ん中です。

メモ:

- 各リストまたは行列内のデータ点は、正の傾きをなすように並べる必要があります。たとえば、Interpolate(2,1,1,3,3)は2を戻しますが、Interpolate(2,3,3,1,1)は欠測値 (.) を戻します。
- Interpolateは連続量のデータに最適で、Step()は離散値に適しています。詳細は、「Step」(109ページ)の節を参照してください。

Step

Step() は、2 点間の線形補間ではなく、階段関数 (ステップ関数) により x に対応する y を求めることを除けば、Interpolate() 関数と同じです。離散型の y 値には Step() を使用してください（この場合、y 値に対応する x 値は y_1 または y_2 のみとなります）。ただし、y 値に対応する x 値が y_1 と y_2 の間である場合は、Interpolate() を使用します。

Interpolate と同様、データ点はリストで指定します。

```
Step(x, x1, y1, x2, y2, ...)
```

また、x 値と y 値が含まれた行列として指定することもできます。

```
Step(x, xmatrix, ymatrix)
```

たとえば、$25、$50、$75、$100 の購入額に対する割引率をまとめたデータテーブルがあるとします。購入額 $35 のときの割引額を示すグラフを作成したいのですが、データテーブルには記入されていません。次の例は、まず評価したい値である 35 を示し、次に購入額 $25 ～ $100 の行列を示しています。

```
Step(35, [25 50 75 100], [5 10 15 25]);
```

この式は、次の値を戻します。

```
5
```

割引率がスライド制であるなら（この場合は、5 と 10 の間）、Interpolate() を使用します。

```
Interpolate(35, [25 50 75 100], [5 10 15 25]);
```

この式は、次の値を戻します。

```
7
```

Interpolate() と同じく、各点は正の傾きをなすように並べる必要があります。

不完全または一致しないデータの比較

欠測値を含んだデータを比較する場合、常に真であるような条件を指定するか、IsMissing() や ZeroOrMissing() などの関数を使用しない限り、間違った結果が戻される可能性があります。また、（数値と文字など）タイプが異なるデータや行列内のデータを比較する場合も、混乱を招くことがあります。

表 5.5 は、そのような比較や行列の例と結果を示しています。比較で使用する演算子については、「演算子」（86 ページ）を参照してください。比較演算子と論理演算子については、表の後の節で詳しく説明しています。

注：ここでの行列は、行数と列数が同じものでなければなりません。

表5.5 特殊なケースの比較テスト（一部）

テスト	結果	説明
m=.; m==1	.	等しいかどうかのテストに欠測値を指定した場合は、欠測値を戻す。
m=.; m!=1	.	等しくないかどうかのテストに欠測値を指定した場合は、欠測値を戻す。
m=.; m<1; m>1; など	.	欠測値を使った比較はどれも欠測値を戻す（真である可能性がある場合は、次を参照）。
m=.; 1<m<0	0	（論理演算子と同様）2つ以上のオペランドを取る比較では、偽は欠測値に優先するため、真である可能性のない欠測値が含まれた比較は偽を戻す。
{a, b}==list(a, b)	1	リストの項目が等しいかどうかのテストは、結果として1つの値を戻す。
{a, b}<{a, c}	.	リストの項目の比較はできない。
1=="abc"	0	等しいかどうかのテストでデータのタイプが異なっている場合は、偽を戻す。
1<="abc"	.	データのタイプが異なっている比較は、欠測値を戻す。
[1 2 3]==[2 2 5]	[0 1 0]	行列が等しいかどうかのテストは、要素ごとの結果を行列で戻す。行列と行列を比較する場合、要素ごとに比較を行い、結果の1と0の行列を戻します。
[1 2 3]==2	[0 1 0]	行列とすべての要素が2である行列が等しいかどうかのテスト。行列を数値と比較する場合、数値はすべての要素がその値である行列として扱われます。
[1 2 3] < [2 2 5]	[1 0 1]	行列の比較は、要素ごとの比較の結果を行列で戻す。
[1 2 3] < 2	[1 0 0]	行列とすべての要素が2である行列との比較。
IsMissing(*m*)	1	欠測値の場合に1を戻し、そうでない場合は0を戻す。（文字タイプの欠測値には間に何も含まない複引用符を使用する。）
ZeroOrMissing(*m*)	1	値が0または欠測値の場合に1を戻す。引数は数値か行列でなければならず、文字列は不可。
All([2 2]==[1 2])	0	要素ごとの比較を要約する。**すべて**の比較が真の場合にのみ1を戻し、それ以外の場合は0を戻す。
Any([2 2]==[1 2])	1	要素ごとの比較を要約する。**いずれか**の比較が真の場合に1を戻し、そうでない場合は0を戻す。

数値の欠測値

欠測値を含む比較の多くは欠測値を戻し、真または偽は戻されません。そのため、スクリプトの処理が止まらないよう、常に真となる結果を含めることが大切な場合があります。データテーブル列に1、2、3という値が含まれており、列「**A**」に欠測値が1つあるとします。列「**B**」の計算式が比較を設定します。次のスクリプトを見てみましょう。

```
New Table( " 比較のテスト ",
   Add Rows( 4 ),
      New Column( "A",
      Numeric,
      Continuous,
      Format( "Best", 10 ),
      Set Values( [1, 2, 3, .] )
   ),
   New Column( "B", Character, Nominal,
                    Formula(
                     If(
                        :A, "true",
                        1, "false"
                  );
               );
            ) );
```

これは、次のような結果を戻します。

```
"true"
"true"
"true"
"false"
```

このスクリプトは次のように機能します。

`If(`	比較を開始する。
`:A, "true",`	「**A**」の値が欠測値でも0でもない場合、結果は"true"。最初の3行では、この比較の結果は真です。
`1, "false"`	1の値は常に真であるため、欠測値は"false"を戻します。
`);`	比較を閉じる。

欠測値を既知の値と比較する場合、例外が2つあります。

- 1つの値が真で、もう1つが欠測値の場合、`Or()` は真を戻します。(`Or()` テストでは、いずれかの値が真であれば、結果は真となります。)
- 1つの値が偽で、もう1つが欠測値の場合、`And()` は偽を戻します。(`And()` テストでは、両方の値が真でなければ結果は真となりません。)

いずれかの値が欠測値であるとわかっている場合は、Is Missing()を使った比較も可能です。前述の例は、次のように記述しなおすことができます。

```
If( :A, "true", Is Missing( :A ), "false", "false" )
```

Is Missing(:A)は、Aが欠測値のときは1、そうでなければ0を戻します。

欠測値が0の可能性がある場合は、代わりにZero Or Missing()関数を使用します。

```
Zero Or Missing(A);
```

この式は、Aが0または欠測値の場合に1を戻します。

ヒント：既知の値を、明示的な欠測値と比較することはできません。比較できるのは、欠測値を**含む**変数や行列などだけです。

文字の欠測値

文字を含まない文字列（name = ""など）は、欠測値ではなく、空の文字列（長さがゼロの文字列）とみなされます。空の文字列を含むデータを比較するときは、A==""やIsMissing(A)などのテストを使用します。

問い合わせ関数

問い合わせ関数を使うと、文字列、リスト、行列などの要素のタイプを調べることができます。要素のタイプがわかれば、それに合わせてスクリプトを記述することができます。

また、JMPは、問い合わせ関数を使ってディレクトリやファイルが書き込み可能かどうかを特定したり、コンピュータのオペレーティングシステムやJMPのバージョンを識別したりできます。

一般的な要素のタイプ

Type()関数は、結果の値のタイプを示す文字列を戻します。たとえば、次のような設定が行えます。

```
Show(Type(1), Type("hi"), Type({"a",2}), Type([10 24 325]));
```

の結果は、次のようになります。

```
Type(1) = "Integer"
Type("hi") = "String"
Type({"a", 2}) = "List"
Type([10 24 325]) = "Matrix";
```

特定の要素のタイプ

その他の問い合わせ関数（Is Matrix()、Is List()、Is Scriptable()など）は、特定のタイプのオブジェクトをテストします。次の例は、Is Matrix()が真の場合に特定の計算を実行します。

```
a = [2 3];
b = [1,1];
c = a * b;
if(Is Matrix(c),
   (c ^ a ) / (a * b),
   Print("c is not a matrix.") );
   [5 25]
```

Is Scriptable()は、オブジェクトがスクリプト可能なものの場合に1を戻します。次の例の4つの変数は、データテーブル、列、プラットフォーム、レポートを参照します。どのオブジェクトもスクリプト可能なので、Is Scriptable()はすべての例で1を戻します。

```
dt=Open( "$SAMPLE_DATA\Big Class.jmp" );
col=Column("体重(ポンド)");
plat=Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ) );
rep=Report(plat);
Show(Is Scriptable(dt));
   1
Show(Is Scriptable(col));
   1
Show(Is Scriptable(plat));
   1
Show(Is Scriptable(rep));
   1
```

Is Empty()は、変数に値、関数、式、またはオブジェクトへの参照があるかどうかを調べます。作成されていない変数や、まだ値が割り当てられていない変数を指定すると、エラーが戻されます。プログラマはこういった変数を**「初期化されていない変数」**と呼んでいます。

次の例は、データテーブルが開いているかどうか、つまりdt変数に割り当てられているかどうかを調べる例です。データテーブルが開いていない場合は、Open()関数がユーザにテーブルを開くよう促します。

```
If(Is Empty(dt = Current Data Table()),
   dt = Open()
);
```

Is Empty()関数はどの変数（グローバル変数、ローカル変数、列など）にも使用できます。

表5.6に、オブジェクトのタイプを識別する関数をまとめます。

表5.6 オブジェクトのタイプを識別する問い合わせ関数

構文	説明
Is Associative Array(*x*)	引数が連想配列のときは1、そうでなければ0を戻す。
Is Directory(x)	引数xがディレクトリのときは1、そうでなければ0を戻す。

表5.6 オブジェクトのタイプを識別する問い合わせ関数（続き）

構文	説明
Is Empty(*global*) Is Empty(*dt*) Is Empty(*col*)	グローバル変数、データテーブル、またはデータ列に値がない（初期化されていない）ときは1、あれば0を戻す。
Is Expr(*x*)	引数が式のときは1、そうでなければ0を戻す。
Is File(x)	引数xがファイルのときは1、そうでなければ0を戻す。
Is List(*x*)	引数がリストのときは1、そうでなければ0を戻す。
Is Matrix(*x*)	引数が行列のときは1、そうでなければ0を戻す。
Is Name(*x*)	引数が名前のときは1、そうでなければ0を戻す。詳細は、「結果ではなく保存された式を取り出す」（205ページ）の節を参照してください。
Is Namespace(*x*)	引数が名前空間のときは1、そうでなければ0を戻す。
Is Number(*x*)	引数が数値か欠測値のときは1、そうでなければ0を戻す。
Is Scriptable(*x*)	引数がスクリプト可能なオブジェクトのときは1、そうでなければ0を戻す。
Is String(*x*)	引数が文字列のときは1、そうでなければ0を戻す。
Type(*x*)	引数（x）のタイプを示す文字列を戻す。

オブジェクトの属性

JMPには、ファイルまたはディレクトリへの書き込みを行う前にそれが書き込み可能であるかどうかを特定するため、次のような関数が用意されています。これらの関数は、スクリプトの対象や属性を検証するIs Directory(path)およびIs File(path)と共に使用してください。その他の情報については、『スクリプト構文リファレンス』の「関数」の章を参照してください。

Is Directory Writable(path)関数は、引数pathで指定したディレクトリが書き込み可能であるときに1、そうでなければ0を戻します。

Is File Writable(path)関数は、引数pathで指定したファイルが書き込み可能であるときに1、そうでなければ0を戻します。

次の例は、パスがディレクトリかどうか、そして、そのディレクトリが書き込み可能かどうかを検証します。

```
If(
   Is Directory(
      "$SAMPLE_DATA\Loss Function Templates"
   ),
   If(
      Is Directory Writable(
         "$SAMPLE_DATA\Loss Function Templates"
      ),
```

```
            "Directory is writable.",
            "Directory is read only!"
         ),
         "Is a read only directory."
      );
```

ホスト情報

Host Is() 問い合わせ関数は、現在のオペレーティングシステムを識別します。その後、オペレーティングシステムに合ったアクションの実行が可能になります。

たとえば、オペレーティングシステムが Windows なら、次のスクリプトは Windows ダイナミックリンクライブラリ（DLL）をロードします。

```
If(Host is(Windows),
   dll_obj = Load DLL("C:¥Windows¥System32¥user32.dll")
);
```

Host Is() を使うと、レポート用に各オペレーティングシステムで異なるテキストサイズを指定することもできます。Windows 上で作成したスクリプトを Macintosh のユーザと共有すると、結果の表示が意図していたものと多少異なる場合があります。たとえば次の行を使用した場合、テキストは、Macintosh なら大きめ、Windows なら小さめに表示されます。

```
textsize = if(host is(Mac),12,10);
```

バージョン情報

JMP Version() 問い合わせ関数は、JMP バージョンを文字列で戻します。この関数を使って JMP のバージョンを確認した後、そのバージョンと互換性のあるスクリプトを実行できます。

```
JMP Version(); // JMP 11 では "11.x.0" を戻す
JMP Version(); // JMP 9 では " 9.0.x" を戻す
```

バージョン番号が 10.0.0 より小さい場合は、将来の 2 桁のバージョンと比較しやすいように 先頭にスペースが挿入されます。先頭のスペースがなければ、9.0.0 は 10.0.0 よりも大きな数（新しいバージョン）と解釈されてしまうためです。

第6章

データタイプ
数、文字列、日付、通貨、その他の使用

この章では、次のような基本的なデータタイプについて説明します。

- 数および文字列
- 特殊な文字列であるパス
- 特殊な数または特殊な文字列である日付および時間
- 通貨
- 16進数値およびBLOB

章の最後の2つの節では、文字列を使った高度な操作と正規表現を使ったパターンマッチの手法を紹介しています。

目次

数および文字列 119
Unicode 文字 119
パス変数 120
パス変数の作成とカスタマイズ 122
相対パス 122
ファイルパスの区切り 123
日付時間の関数と形式 123
日付時間値 123
日付時間関数を使用したプログラム 124
データテーブル内の日付時間値 130
通貨 133
16 進数の関数と BLOB 関数 134
文字関数の使用 137
Concat 137
Munger 138
Repeat 139
パターンマッチと正規表現の使用 140

数および文字列

数は、整数や小数、E の後ろに 10 のべき指数をつけた科学的表記、および日付時間値として表すことができます。ピリオド 1 つだけのものは欠測値です。

たとえば、次に挙げるものはすべて数です。

```
.   1   12   1.234  3E3  0.314159265E+1 1E-20
```

二重引用符に囲まれた 1 つまたは複数の文字は**文字列**となります。たとえば、次に挙げるものはすべて文字列です。

```
"Green" "Hello,\NWorld!" "54"
```

数が二重引用符に囲まれた場合は文字列で、数値ではないことに注意してください。数値を文字列に、または文字列を数値に変換するには、次の 2 つの関数を使用します。

- 文字列を数値に変換するには Num() を使用します。たとえば、次のような設定が行えます。

```
Num("54");
   54
```

注：Num() は数以外の文字を変換できないため、次のような場合には欠測値が戻されます。

```
Num("Hello");
   .
```

- 数値を文字列に変換するには Char() を使用します。たとえば、次のような設定が行えます。

```
Char(54);
   "54"
Char(3E3)
   "3000"
```

Num() または Char() の出力で、ロケール固有の形式を保持するには、次のように <<Use Locale(1) オプションを含めます。

```
Char( 42,5,2, <<Use Locale(1) );
   // フランスのロケールで実行した場合、"42,00" になる
```

Unicode 文字

JMP では、世界の言語のほとんどを、Unicode UTF-8 規格と UTF-16 規格でエンコーディングおよびテキスト表示できます。Unicode 規格のコード表と詳細については、ユニコードコンソーシアム（The Unicode Consortium）を参照してください。

JMP で Unicode 文字を表示するには、その文字の Unicode コードの前に「\!」をつけます。例:

- ギリシャ文字のシグマ（σ）の Unicode コードは U+03C3 なので、JMP では \!U03C3 と指定します。
- ギリシャ文字のミュー（μ）の Unicode コードは U+03BC なので、JMP では \!U03BC と指定します。

Unicodeを使って肩文字や添え字を表示する場合

- 上付きの1（$_1$）のUnicodeはU+2081なので、JMPでは \!U2081と指定します。
- 下付きの2（^{2}）のUnicode はU+00B2 なので、JMPでは \!U00B2と指定します。

x^2をUnicodeで表現する場合、JMPでは \!U0078\!U00B2と指定します。

パス変数

パス変数は、ディレクトリやファイルへのショートカットです。ディレクトリやファイルへのパス全体を入力する代わりに、スクリプト内でパス変数を使用できます。パス変数は特殊な文字列であり、常に二重引用符で囲んで使用します。

JMPで広く使用される定義済みのパス変数の1つに $SAMPLE_DATA があります。この変数はJMPインストールフォルダ内のサンプルデータのフォルダを指します。次の例は、「Big Class.jmp」サンプルデータテーブルを開きます。

```
Open("$SAMPLE_DATA¥Big Class.jmp")
```

JMPではいくつかのパス変数が定義済みです。次の表に、JMP 11のパス変数を示します。旧バージョンのJMPの変数はこれとは異なる場合があります。

表6.1 パス変数の定義

変数	パス
ALL_HOME	• Windows: "C:¥ProgramData¥SAS¥JMP¥<バージョン番号>¥" • Macintosh: "/Library/Application Support/JMP/<バージョン番号>/"
DESKTOP	• Windows: "C:¥Users¥<ユーザ名>¥Desktop¥" • Macintosh "/Users/<ユーザ名>/Desktop/"
DOCUMENTS	• Windows: "C:¥Users¥<ユーザ名>¥Documents¥" • Macintosh: "/Users/<ユーザ名>/Documents/"
ENGLISH_SAMPLE_DATA	この変数は廃止されました。代わりに、“$SAMPLE_DATA”を使用してください。
GENOMICS_HOME	"¥<JMP Genomicsのインストールディレクトリ>¥"
HOME	• Windows: "C:¥Users¥<ユーザ名>¥AppData¥Local¥SAS¥JMP¥<バージョン番号>¥" • Macintosh: "/Users/<ユーザ名>/"

表 6.1 パス変数の定義（続き）

変数	パス
SAMPLE_APPS	• Windows: C:¥<JMP インストールディレクトリ>¥Samples¥Apps¥" • Macintosh: "/Library/Application Support/<JMP インストールディレクトリ>/Samples/Apps/"
SAMPLE_DATA	• Windows: "C:¥<JMP インストールディレクトリ>¥Samples¥Data¥" • Macintosh: "/Library/Application Support/<JMP インストールディレクトリ>/Samples/Data/"
SAMPLE_IMAGES	• Windows: "C:¥<JMP インストールディレクトリ>¥Samples¥Images¥" • Macintosh: "/Library/Application Support/<JMP インストールディレクトリ>/Samples/Images/"
SAMPLE_IMPORT_DATA	• Windows: "C:¥<JMP インストールディレクトリ>¥Samples¥Import Data¥" • Macintosh: "/Library/Application Support/<JMP インストールディレクトリ>/Samples/Import Data/"
SAMPLE_SCRIPTS	• Windows: "C:¥<JMP インストールディレクトリ>¥Samples¥Scripts¥" • Macintosh: "/Library/Application Support/<*JMP インストールディレクトリ*>/Samples/Scripts/"
TEMP	• Windows: "C:¥Users¥<ユーザ名>¥AppData¥Local¥Temp¥" • Macintosh: "/private/var/folders/.../Temporary Items/"
USER_APPDATA JMP 環境設定やメニュー、ホームウィンドウに加えた変更、およびデバッガセッションの設定がここに保存されます。	• Windows: "C:¥Users¥<ユーザ名>¥AppData¥Local¥SAS¥JMP¥<*バージョン番号*>¥" • Macintosh: "/Users/<ユーザ名>/Library/Application Support/JMP/<バージョン番号>/"

パス変数の定義は、使用している JMP のバージョンに応じたものとなります。たとえば、JMP 9 で作成されたスクリプトであっても、JMP 11 で実行した場合は、JMP 11 におけるパス変数の定義が使用されます。

パス変数の定義を確認するには、`Get Path Variable` 関数を使用します。

```
Get Path Variable("HOME");
   "C:¥Users¥<ユーザ名>¥AppData¥Local¥SAS¥JMP¥11¥"
```

`Set Path Variable()` または `Get Path Variable()` にはドル記号は指定しません。ただし、スクリプト内で変数を使用する際には、ドル記号を指定する必要があります。

末尾のスラッシュ

パス変数の最後に必ずスラッシュ（または \）を付けてください。次の例では、dtName変数に "Big Class" というルート名が割り当てられています。Open式は、$SAMPLE_DATA と末尾のスラッシュを評価し、dtName の値とファイル拡張子の .jmp を付加します。

```
dtName = "Big Class";
dt = Open("$SAMPLE_DATA/"|| dtName ||".jmp");
```

このパスは、次のように解釈されます。

```
C:¥Program Files¥SAS¥JMP¥11¥Samples¥Data¥Big Class.jmp
```

$SAMPLE_DATA の後にスラッシュがない場合、パスは次のように解釈されます。

```
C:¥Program Files¥SAS¥JMP¥11¥Samples¥DataBig Class.jmp
```

パス変数の作成とカスタマイズ

Set Path Variable() を使って、独自のパス変数を作成したり、ビルトインのパス変数を上書きしたりできます。次の例で、パス変数は root です。この変数は c:¥ ディレクトリを指します。

```
Set Path Variable("root", "c:¥");
```

新しい変数の値を取得するには、Get Path Variable() を使用します。

```
Get Path Variable("root"); // "c:¥" を戻す
```

独自のパス変数も他の変数と同様に使用します。次の式は、c:¥ ディレクトリの myimportdata.txt ファイルを開きます。

```
Open("$root¥myimportdata.txt")
```

パス変数の取得の場合と同様、パス変数を設定する際はドル記号を指定しません。

相対パス

変数で相対パスを使用する予定がある場合は、デフォルトのディレクトリを設定しなければなりません。そうすることで、先頭にドライブ文字がないパスはすべて、デフォルトのディレクトリへの相対パスになります。以下はその例です。

```
Set Default Directory("c:¥users¥smith¥data");
```

デフォルトのディレクトリの値を戻すには、Get Default Directory() を使用します。

```
Get Default Directory(); // "c:¥users¥smith¥data" を戻す
```

そこで、次の式:

```
Open("cleansers.jmp");
```

は C:¥users¥smith¥data¥cleansers.jmp と解決されます。

ファイルパスの区切り

JMP では、通常、区切り記号にスラッシュ（/）を使用する Portable Operating System Interface（POSIX）（UNIX）形式のファイルパスを使用します。これにより、Windows と Macintosh で実行するスクリプト内で現在のオペレーティングシステムを識別する必要はありません。ただし、各ホストは、そのホストでのネイティブ形式のファイルパスも許容しています。

ファイルパス形式を Windows から POSIX（またはその逆）に変換するには `Convert File Path()` を使用します。パスをファイルや別のアプリケーションに出力する必要がある場合は、POSIX から Windows のパス形式に変換すると便利です。構文は次のとおりです。

```
Convert File Path (path, <absolute|relative>, <POSIX|windows>, <base(path)>);
```

たとえば、次のスクリプトは POSIX パスを Windows パスに変換します。

```
Convert File Path("/c:/users/smith", windows); // c:¥users¥smith を戻す
```

二重引用符内のパスをパス変数（$HOME など）に置き換えることもできます。

日付時間の関数と形式

日付時間値は、日付や時間の要素で構成されます。値は、秒（3388594698）、完全な日付（“2011 年 5 月 18 日”など）、日付と時間（“11/05/18 20:18:18”）、週番号（3）などです。

JMP では、日付時間値を共通の形式に変換し、算術演算を行ったり、さまざまな方法で処理したりできます。

ヒント：日付時間値のすべての関数と引数については、『スクリプト構文リファレンス』を参照してください。

日付時間値

日付時間値は、1904 年 1 月 1 日午前 0 時から数えた秒数として保存され、計算されます。たとえば、次のような設定が行えます。

```
Today(); // これは、2011 年 5 月 19 日午前 12:00:00 の場合、3388649872 を戻す
```

`Today()` と同様、`Date DMY()` 関数と `Date MDY()` 関数も、月、日、年の引数を秒数で戻します。たとえば、現在が 2011 年 5 月 19 日午前 12:00:00 だとしたら、次のすべてのステートメントは同じ値を戻します。

```
Date DMY(19,5,2011);
Date MDY(5,19,2011);
Today();
   3388608000
```

`As Date()` 関数は引数として指定された秒数を日付で戻します。

```
As Date(3388608000);
   19May2011
```

日付時間値は次の2つの形で使用できます。

- リテラル値。例: 19May2011:10:10
- 文字列。例: "2011年5月19日 木曜日"

日付時間のリテラルは、秒数を基本数とし、算術演算に使用できます。

```
As Date( 19May2011 + 1 );
   19May2011:00:00:01
```

日付時間関数を使用したプログラム

表6.2に、秒数を日付時間値に変換する関数と、日付時間値を秒数に変換する関数を示します。

表6.2 日付時間関数

関数	説明
Abbrev Date(*date*)	指定された日付（**date**）からOSで指定されているロケールの日付表示を戻す。形式はコンピュータの地域設定に基づきます。英語（米国）ロケールの場合、日付は "02/29/2004" のような形式になります。英語版のJMPを別のロケールで実行している場合も、ロケールの形式が適用されます。
As Date(*expression*)	指定された数値または式を日付形式で戻す。たとえば、次のような設定が行えます。 `x = As Date(8Dec2000 + inDays(2));` は、次のように表示されます。 *10Dec2000*
Date DMY(*day*, *month*, *year*)	指定された日付を、1904年1月1日午前0時からの秒数で表示する。たとえば、2004 年 2 月 29 日は DateDMY(29,2,2004) と指定します。これは3160857600を戻します。
Date MDY(*month*, *day*, *year*)	指定された日付を、1904年1月1日午前0時からの秒数で表示する。たとえば、2004 年 2 月 29 日は DateMDY(2,29,2004) と指定します。これは3160857600を戻します。
Day Of Week(*date*)	指定された日付（*date*）が何曜日であるかを表す整数値を戻す。この関数では、週は、日曜日から始まり、土曜日で終わるとみなします。
Day Of Year(*date*)	指定された日付（*date*）がその年の何日目であるかを表す整数値を戻す。
Day(*date*)	指定された日付（*date*）がその月の何日であるかを表す整数値を戻す。

表 6.2 日付時間関数（続き）

関数	説明
Format(*date*, *"format"*)	2 番目の引数で指定された形式（**format**）で*値*を戻す。秒数を日付時間値で表示するときに最もよく使われます。選択できる形式は、**「列情報」**ダイアログボックスに表示されています。表 6.3「JMP の 2 桁年の解釈方法」（129 ページ）も参照してください。
Hour(*datetime*)	指定された日付時間値（*date-time*）の時間を表す整数値を戻す。
In Days(*n*)	これらの関数はそれぞれ、*n*分、*n*時間、*n*日、*n*週間、*n*年を秒で表した値を戻す。秒数で表した時間間隔をこれらの関数で割ると、別の時間単位に変換できます。
In Hours(*n*)	
In Minutes(*n*)	
In Weeks(*n*)	
In Years(*n*)	
Long Date(*date*)	指定された日付（*date*）から OS で指定されているロケールの日付表示を戻す。形式はコンピュータの地域設定に基づきます。英語（米国）ロケールの場合、日付は "Sunday, February 29, 2004" のような形式になります。英語版の JMP を別のロケールで実行している場合も、ロケールの形式が適用されます。
MDYHMS(*date*)	指定された日付（*date*）から "2/29/04 00:02:20 AM" のような形式の日付表示を戻す。
Minute(*date-time*)	指定された日付時間値（*date-time*）の分を表す整数値を戻す。
Month(*date*)	指定した日付（*date*）の月を数値で戻す。
InFormat(*string*, *"format"*) Parse Date(*string*, *"format"*)	指定された形式（*format*）の文字列（*string*）を解析し、日付時間値を戻す。As Date() と同様に、日付を ddMonyyyy 形式で戻します。
Second(*date-time*)	指定された日付時間値（*date-time*）の秒を表す整数値を戻す。
Short Date(*date*)	指定された日付（*date*）をロケールに関係なく mm/dd/yyyy の形式で戻す（たとえば "02/29/2004"）。
Time Of Day(*date*)	指定された日時（*date-time*）がその日の何秒目かを整数値で戻す。
Today()	1904 年 1 月 1 日午前 0 時から現在の日時までの秒数を戻す。引数はとりませんが、括弧は必要です。

表 6.2 日付時間関数（続き）

関数	説明
Week Of Year(*date*, *<ルール番号>*)	日付がその年の何週目であるかを戻します。一年の第1週を定義するルールは3つあり、オプションとして指定できます。 • ルール1（デフォルト）では、週は日曜日から始まり、年の最初の日曜日が第2週となります。第1週は一部だけの週となるかまたは存在しません。 • ルール2では、最初の日曜日が第1週となり、その前にある日は第0週となります。 • ルール3は、ISO-8601方式の週番号を戻します。週は月曜日から始まり、その年に入ってからの4日間を含む最初の週が第1週となります。年の最初の3日間または最後の3日間が前年または翌年の週番号に属する場合があります。
Year(*date*)	指定された日付（*date*）の年を数値で戻す。

共通の日付時間関数の例

日付時間を戻す関数内では、秒数を戻すすべての関数を使用できます。

たとえば、今日が2011年5月19日の午前11:37:52だとすると、Today() は秒数を戻し、それ以下の関数は基本時間からの秒数を別の日付時間形式で戻します。

```
Today()
   3388649872

Short Date(Today());
   "05/19/2011"
Long Date(Today());
   "2011 年 5 月 19 日 "
Abbrev Date(Today());
   "2011/5/19"
MDYHMS(Today());
   "05/19/2011 11:37:52"
```

括弧内の日付の引数には、秒数（または秒数を戻す関数）または日付時間のリテラル値を指定できます。たとえば、次の2つの式は同じ値を戻します。

```
Long Date(3388649872);
Long Date(19May2011);
   "2011 年 5 月 19 日 "
```

注： Long Date() と Abbrev Date() の値は、お使いのコンピュータの地域設定に応じて異なります。

日付の一部の抽出

関数Month()、Day()、Year()、Day Of Week()、Day Of Year()、Week Of Year()、Time Of Day()、Hour()、Minute()、Second()を使って、日付値の一部を抽出できます。これらの関数は、すべて整数を戻します。現在日付が2011年5月24日の場合、以下の例はすべて年の144日目を表す144ををを戻します。

```
Day of Year(Today());
Day of Year(24May2011);
Day of Year(Date MDY(5,24,2011));
   144
```

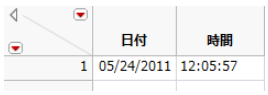
例

データテーブルの「**日付**」という名前の列に、"m/d/y"形式の日付時間値が含まれています。ここで、時間だけが含まれる列を作成するとします。次のスクリプトは、2列目の計算式が1列目の「**日付**」の値から時間を抽出します。

```
New Table( "Assembly Tests",
   Add Rows( 1 ),
   New Column( "日付",
      Numeric, Continuous,
      Format( "m/d/y" ),
      Set Values( [3389083557] )
   ),
   New Column( "時間",
      Numeric, Continuous,
      Formula(Format(Time Of Day( :日付 ), "h:m:s"))
   );
);
```

図6.1はこの結果です。時間は「**日付**」列には表示されないことに注意してください。これは、Format関数によって“m/d/y”形式を適用しているためです。

図6.1 時間の抽出の例



	日付	時間
1	05/24/2011	12:05:57

年の週を決定するルール

Week of Year()は、その年の何週目であるかを、日付時間値で戻します。一年の第1週を定義するルールは3つあり、オプションとして指定できます。

- ルール1（デフォルト）では、週は日曜日から始まり、年の最初の日曜日が第2週となります。第1週は一部だけの週となるかまたは存在しません。

```
Week Of Year(Date DMY(19,6,2013),1);
   25
```

- ルール 2 では、最初の日曜日が第 1 週となり、その前にある日は第 0 週となります。

```
Week Of Year(Date DMY(19,6,2013),2);
   24
```

- ルール 3 は、ISO-8601 方式の週番号を戻します。週は月曜日から始まり、その年に入ってからの 4 日間を含む最初の週が第 1 週となります。年の最初の 3 日間または最後の 3 日間が前年または翌年の週番号に属する場合があります。

```
Week Of Year( Date DMY(19,6,2013),3);
   25
```

日付の演算

日付時間データは、他の数値データと同じように、算術演算に使用できます。たとえば、日付時間値から数値を引くといった単純な計算ができます。

また、計算式を書いて実行することもできます。

例

データテーブルの「**日付**」列に、顧客がクレジットカードを使ってガソリンを購入した日付が入っています。顧客が何日おきにガソリンを購入しているかを知りたいとします。次のスクリプトは「**経過日数**」列を作成します。その列の計算式は、直前の行の日付値から現在の行の「**日付**」列の値を引きます。

```
New Table(" ガソリンの購入 ",
    Add Rows(3),
    New Column(" 日付 ",
        Numeric, Continuous,
        Format("m/d/y"),
        Set Values([3392323200 3393532800 3394828800])
    ),
    New Column(" 経過日数 ",
      Formula(
          If(row()==1, ., // 1 行目については欠測値を戻す
          (: 日付 [row()]-: 日付 [row() - 1])/in days())));
    );
```

図 6.2 はこの結果です。

図 6.2 日付時間値の計算例

	日付	経過日数
1	07/01/2011	•
2	07/15/2011	14
3	07/30/2011	15

時間の間隔

In Minutes、In Hours、In Days、In Weeks、In Yearsの各関数は、秒以外の単位で時間を表現するときに使用します。これらの関数はすべて、特定の時間間隔を秒数で戻します。たとえば、次の式は、現在から2012年7月4日までの週数を戻します。

```
(Date DMY(04,07,2012)-Today())/InWeeks();
   55.6559441137566
```

間隔関数の引数が空の場合、間隔は1と見なされます。これを変更するには別の数値を入力します。たとえば、In Years(10)は間隔を10年に変換します。たとえば、次の式は、現在から2037年12月31日までに10年間がいくつ含まれるかを戻します。

```
(Date DMY(31,12,2037)-Today())/InYears(10);
   2.65581440286967
```

2桁および4桁の年

JMPは、オペレーティングシステム固有の日付時間形式をサポートするのではなく、独自のアルゴリズムで日付時間値を解釈し、表示します。ただし、日付時間区切りは、「地域と言語」コントロールパネル（Windows）または「日付と時刻」の環境設定（Macintosh）で選択されているものが使用されます。

2桁年は、現在のシステムクロック年とJMPのルールに従って解釈されます。たとえば、スクリプト内の年が11で、1990年以降にそのスクリプトを実行した場合、年は2011と表示されます。

```
Long Date(25May11);
   "2011年5月25日"
```

曖昧さを回避するには、4桁の年を入力します。次の式は（2011年ではなく）1911年を戻します。

```
Long Date(25May1911);
   "1911年5月25日"
```

表6.3に、JMPの2桁年の解釈方法を示します。

表6.3 JMPの2桁年の解釈方法

2桁の年の値	評価する時	結果	例	結果
00–10	1989年以前（Windows）	19__	1979年に5と入力	1905
	1990年以前（Macintosh）			
	1990年以降（Windows）	20__	1991年に5と入力	2005
	1991年以降（Macintosh）			
11–89（Windows）	いつでも	現在の世紀	1988年に13と入力	1913
11–90（Macintosh）			2024年に13と入力	2013

表 6.3 JMP の 2 桁年の解釈方法（続き）

2 桁の年の値	評価する時	結果	例	結果
90–99（Windows）	2010 年以前	19__	1999 年に 99 と入力	1999
91–99（Macintosh）	2011 年以降	20__	2015 年に 99 と入力	2099

注：JMP では、「地域」の設定に関係なく、年が必ず 4 桁で表示されます。何らかの理由で、2 桁で年を表示する必要がある場合は、文字列関数を使ってください。詳細については、「データタイプ」（117 ページ）の章を参照してください。

データテーブル内の日付時間値

日付時間入力と表示形式の変更

データテーブルでは、日付時間値を 1 つの形式（**入力形式**）でのみ入力できます。入力した値は、基準日からの秒数として内部に保存され、別の日付時間形式で表示されます。`Informat()` 関数と `Format()` 関数を使って、これを制御できます。

- `In Format()` は、文字列の日付時間値と、その文字列に使用されている日付形式を指定すると、ddMonyyyy 形式で日付を戻します。

```
Informat("19May2011 11:37:52","ddMonyyyy h:m:s");
   19May2011:11:37:52
```

- `Format()` は、基準日からの秒数（または秒数を戻す日付時間関数）を取り、指定の形式で日付を戻します。

```
Format(3388649872,"ddMonyyyy h:m:s");
   "19May2011 11:37:52"
Format(Today(),"ddMonyyyy h:m:s");
   "19May2011 11:37:52"
```

列に `d/m/y h:m` 形式の日付を入力し、日付を `m/d/y` 形式で表示してみましょう。`Informat()` 関数は入力形式を定義し、`Format()` 関数は表示形式を定義します。次に例を挙げます。

```
New Table("製品の組立",
    Add Rows(1),
    New Column("日付",
        Numeric, Continuous,
        Format("m/d/y"),
        Input Format("d/m/y h:m"),
        Set Values([3126917100])
    );
);
```

図 6.3 日付時間の表示と入力値の例

`Format()` と `Informat()` の値は、データテーブルの列プロパティに表示されます（図 6.3）。セルをクリックして編集する際、日付時間値は入力時の形式で表示されます。値を変更したり、新しい値を入力したりすると、その列の［**表示形式**］で指定されている形式で値が表示されます。

特定の日付時間関数については、『スクリプト構文リファレンス』を参照してください。

表 6.4 に、日付時間関数の引数として、またはデータテーブル形式として使用できる形式を示します。これらの形式 は、データ列への `Format` メッセージの *format* 引数として使うこともできます。「データテーブル」の章の「形式の設定または取得」（305 ページ）を参照してください。使用するコンピュータによっては、日付区切り文字がスラッシュ（/）文字ではないことがあります。

注：時間値は、24 時間形式、または午前・午後という識別子を付けた形式で入力します。

表 6.4 日付時間形式

タイプ	形式引数	例
日付だけ	"m/d/y"	"01/02/1999"
	"mmddyyyy"	"01021999"
	"m/y"	"01/1999"
	"d/m/y"	"02/01/1999"
	"ddmmyyyy"	"02011999"
	"ddMonyyyy"	"02Jan1999"
	"Monddyyyy"	"Jan021999"
	"y/m/d"	"1999/01/02"
	"yyyymmdd"	"19990102"
	"yyyy-mm-dd"	“1999-01-02“
	"yyyyQq"	1999Q1

表 6.4 日付時間形式（続き）

タイプ	形式引数	例
日付と時間	"m/d/y h:m"	"01/02/1999 13:01" "01/02/1999 午後 1:01"
	"m/d/y h:m:s"	"01/02/1999 13:01:55" "01/02/1999 午後 1:01:55"
	"d/m/y h:m"	"02/01/1999 13:01" "02/01/1999 午後 1:01"
	"d/m/y h:m:s"	"02/01/1999 13:01:55" "02/01/1999 午後 1:01:55"
	"y/m/d h:m"	"1999/01/02 13:01" "1999/01/02 午後 1:01"
	"y/m/d h:m:s"	"1999/01/02 13:01:02" "1999/01/02 午後 1:01:02"
	"ddMonyyyy h:m"	"02Jan1999 13:01" "02Jan1999 午後 1:01"
	"ddMonyyyy h:m:s"	"02Jan1999 13:01:02" "02Jan1999 午後 1:01:02"
	"ddMonyyyy:h:m"	"02Jan1999:13:01" "02Jan1999: 午後 1:01"
	"ddMonyyyy:h:m:s"	"02Jan1999:13:01:02" "02Jan1999: 午後 1:01:02"
	"Monddyyyy h:m"	"Jan021999 13:01" "Jan021999 午後 13:01"
	"Monddyyyy h:m:s"	"Jan021999 13:01:02" "Jan021999 午後 13:01:02"
経過日数と時間	":day:hr:m"	"34700:13:01" ":33:001:01 PM"
	":day:hr:m:s"	"34700:13:01:02" ":33:001:01:02 PM"
	"h:m:s"	"13:01:02" " 午後 01:01:02"
	"h:m"	"13:01" " 午後 01:02"
	"yyyy-mm-ddThh:mm"	1999-01-02T13:01
	"yyyy-mm-ddThh:mm:ss"	1999-01-02T13:01:02

表 6.4 日付時間形式（続き）

タイプ	形式引数	例
時間の長さ	":day:hr:m"	“52:03:01” （52 日 3 時間 1 分）
	":day:hr:m:s"	“52:03:01:30” （52 日 3 時間 1 分 30 秒）
	"hr:m"	“17:37” （17 時間 37 分）
	"hr:m:s"	“17:37:04” （17 時間 37 分 4 秒）
	"min:s"	“37:04” （37 分 4 秒）
注：以下の形式は、日付時間をコンピュータの地域設定に従って表示します。これらの形式は、表示にのみ適用され、データテーブルへの日付の入力には使用できません。例は、米国のロケールを使用した場合です。		
短い日付	"Date Abbrev"	（表示のみ）"1999/01/02"
長い日付	"Date Long"	（表示のみ）"1999 年 1 月 2 日 "
ロケールの日付	“Locale Date”	（表示のみ）"1999/01/02"
ロケールの日時	“Locale Date Time h:m”	（表示のみ）"1999/1/2 13:01" または "1999/01/02 01:01 PM"
	“Locale Date Time h:m:s”	（表示のみ）"1999/01/02 13:01:02" または "1999/01/02 01:01:02 PM"

通貨

JMP は、Format() 関数を使って数値を通貨の形式で表示します。関数の構文は次のとおりです。

```
Format(x,"Currency", <"currency code">, <decimal>);
```

ここで

- *x* は列または数値
- *"currency code"* は国際標準化機構（ISO）4217 コード
- *decimal* は小数桁数

次はFormat関数の例です。

```
Format(12345.6, "Currency", "GBP", 3);
   "£12,345.600"
```

通貨コード（currency code）を指定しない場合、オペレーティングシステムのロケールに基づいた通貨記号が使用されます。たとえば、次のスクリプトを日本語のオペレーティングシステムで実行した場合、円記号が付きます。

```
Format(12345.6, "Currency", 3);
   "¥12,345.600"
```

JMPでサポートされていない通貨コードを指定すると、数値の前に通貨コードの文字列が付きます。

```
Format(12345.6, "Currency", "BBD", 3);
   "BBD 12,345.600"
```

表6.5に、JMPでサポートされている通貨を示します。

表6.5 JMPでサポートされている通貨

コード	通貨	コード	通貨	コード	通貨
AUD	オーストラリアドル	HKD	香港ドル	PHP	フィリピンペソ
BRL	ブラジルレアル	ILS	イスラエル新シェケル	PLN	ポーランドズウォティ
CAD	カナダドル	INR	インドルピー	RUB	ロシアルーブル
CHF	スイスフラン	JPY	日本円	SEK	スウェーデンクローネ
CNY	中国元	KRW	韓国ウォン	SGD	シンガポールドル
COP	コロンビアペソ	MXN	メキシコペソ	THB	タイバーツ
DKK	デンマーククローネ	MYR	マレーシアリンギット	TWD	ニュー台湾ドル
EUR	ユーロ	NOK	ノルウェークローナ	USD	米国ドル
GBP	英国ポンド	NZD	ニュージーランドドル	ZAR	南アフリカランド

16進数の関数とBLOB関数

JMPでは、BLOBと呼ばれるバイナリ大容量オブジェクト（Binary Large OBject）も扱うことができます。以下の関数は、16進数の値、数値、文字、BLOBの間での変換を行います。一部の関数については、表6.6の後に詳しい説明があります。

これらの関数は、『スクリプト構文リファレンス』にリストされています。

表6.6 16進数の関数とBLOB関数

構文	説明
`Hex("text")` `Hex(num)` `Hex(blob)`	テキスト（**text**）、数字（**num**）、またはBLOB（**blob**）を16進数表記に変換した文字列を戻す。 `Char To Hex`は別名です。
`Hex To Blob("hexstring")`	16進数表記の文字列を、BLOBに変換する。
`Hex To Char("hexstring", encoding)`	16進数表記の文字列を、指定されたエンコーディングに従い、文字列に変換する。 16進数のエンコーディングは、デフォルトでUTF-8に設定されています。UTF-8、UTF-16LE、UTF-16BE、US-ASCII、ISO-8859-1、ASCII~HEXのいずれかのエンコーディングを使用できます。
`Hex to Number`	16進数表記の文字列を、数値に変換する。
`Char To Blob(string)` `Char To Blob(string, encoding)`	文字列（**string**)を、バイナリデータ（BLOB）に変換する。 16進数のエンコーディングは、デフォルトでUTF-8に設定されています。UTF-8、UTF-16LE、UTF-16BE、US-ASCII、ISO-8859-1、およびASCII~HEXを使用できます。
`Blob To Char(blob)` `Blob To Char(blob, encoding)`	バイナリデータ（BLOB）を、文字列に変換する。 16進数のエンコーディングは、デフォルトでUTF-8に設定されています。UTF-8、UTF-16LE、UTF-16BE、US-ASCII、ISO-8859-1、およびASCII~HEXを使用できます。
`Blob Peek(blob, offset, length)`	元のBLOBから、オフセット*offset*から始まる長さ**length**バイトだけを抜き出し、新しいBLOBとして戻す。オフセットは0を基準にします。

`Hex (string)`は、引数の各文字に対応する16進数のコードを返します。例:

```
Hex("Abc")
```

は次を戻します。

```
"416263"
```

41、62、63はそれぞれ、A、b、cを表す16進数のコード（Ascii）です。

`Hex to Char (string)`は、16進数を文字に変換します。戻り値の文字が、有効な表示文字でない場合があります。すべての文字は、0-9、A-Z、またはa-zのいずれかの2文字で表されます。スペースまたはカンマは、無視されます。例:

```
Hex To Char ("4142")
```

は次を戻します。

```
"AB"
```

41と42は、それぞれAとBを表す16進数のコードです。

HexとHex To Charは、逆の処理を行うため、たとえば、

```
Hex To Char ( Hex("Abc") )
```

は次を戻します。

```
"Abc"
```

Hex To Blob(string)は、16進表記の文字列（string）をバイナリオブジェクトに変換します。

```
a = Hex To Blob("6A6B6C"); Show(a);
   a = Char To Blob("jkl", "ascii~hex")
```

Blob Peek(blob,offset,length)は、引数で指定されたバイトをblobから抽出します。

```
b = Blob Peek(a,1,2); Show(b);
   b = Char To Blob("kl", "ascii~hex")
b = Blob Peek(a,0,2); Show(b);
   b = Char To Blob("jk", "ascii~hex")
b = Blob Peek(a,2); Show(b);
   b = Char To Blob("l", "ascii~hex")
```

Hex(blob)は、blobを16進に変換します。

```
c = Hex(a); Show(c);
   c = "6A6B6C"
d = Hex To Char(c); Show(d);
   d = "jkl"
```

Concat(blob1,blob2)とblob1 || blob2は、2つのblobを連結します。

```
e = Hex To Blob("6D6E6F"); Show(e);
f = a||e; show(f);
   e = Char To Blob("mno", "ascii~hex")
   f = Char To Blob("jklmno", "ascii~hex")
```

length(blob)は、blobのバイト数を戻します。

```
g = length(f); show(g);
   g = 6
```

注:blob がログにリストされる場合、コンストラクタ関数の Char To Blob("...") とともに表示されます。

このとき、ASCII の範囲外は、ASCII 文字ではなく、波線のあとに 16 進数を示す方法で表記されます。なお、ASCII の範囲は、スペース（space）から、閉じ中括弧(}) までです（16 進数のコードで言うと、20 から 7D までです）。例:

```
h= Hex To Blob("19207D7E"); Show(h);
i = Hex(h); Show(i);
   h = Char To Blob("~19 }~7E", "ascii~hex")
   i = "19207D7E"
```

Char To Blob(string) は、文字列から BLOB を作成します。Blob To Char(blob) は、BLOB から文字列を作成します。これらにおいて、表示できないコードや ASCII 文字ではないコードは、波線と 16 進数（~XX）によって表わされます。

文字関数の使用

この節では、『スクリプト構文リファレンス』で説明している複雑な文字関数の一部について使用方法を示します。

Concat

Concat 関数は、文字列と文字列を連結します。なお、評価された時に名前を戻す式に対しては、その名前を文字列として扱います。次の例は、変数の名前を文字列として Concat 関数で扱うために、Name Expr と Char 関数を用いています。

```
n = { abc };
c=n[1] || "def";
show(c);
// 結果は "abcdef"

m=expr(mno);
c = m || "xyz";
show(c);
// 結果は、mno が解決されていないというエラーメッセージ

m = expr(mno);
c = Name Expr(m) || "xyz";
show(c);
// 結果は "mnoxyz"

m=char(expr(mno));
c=m || "xyz";
show(c);
// 結果は "mnoxyz"
```

Concat Items()は、文字列のリストを、1つの文字列に変換します。各文字列は、区切り文字で区切られます。区切り文字を指定しなかった場合は、スペースで区切られます。構文は、次のとおりです。

```
resultString = Concat Items ({list of strings}, <“delimiter string”>);
```

例:

```
a = {“ABC”, “DEF”, “HIJ”};
result = Concat Items(a, “/”);
```

は次を戻します。

```
“ABC/DEF/HIJ”
```

または、

```
result = Concat Items(a);
```

は次を戻します。

```
“ABC DEF HIJ”
```

Munger

Mungerは、引数として何が指定されているかによって異なる動作をします。

次の表では、Munger(text, *offset*, *find* | *length*, <*replace*>) を使って説明しています。

表 6.7 さまざまな種類の引数をとったときのMungerの動作

引数 find、length、replace	例
*find*に文字列を指定し、*replace*に文字列を指定しなかった場合、最初に *find*文字列が見つかった位置（offsetの位置から数えた文字数）を戻す。	Munger("the quick brown fox", 1, "quick"); *5*
*length*に正の整数を指定し、*replace*に文字列を指定しなかった場合、offset番目から（*offset*+*length*）番目までの文字列を戻す。	Munger("the quick brown fox",1,5); *"the q"*
findに文字列を指定し、*replace*に文字列を指定した場合、*text*内のoffsetの後ろで最初に一致した*find*の文字列を *replace* の文字列に置き換える。	Munger("the quick brown fox", 1, "quick", "fast"); *"the fast brown fox"*
*length*に正の整数を指定し、*replace*に文字列を指定した場合、offset 番目から（*offset* + *length*）番目までの文字列を *replace*の文字列に置き換える。	Munger("the quick brown fox", 1, 5, "fast"); *"fastuick brown fox"*

表 6.7 さまざまな種類の引数をとったときの Munger の動作（続き）

引数 find、length、replace	例
length に正の整数を指定し、*offset* + *length* が *text* の文字数以上の場合、*text* の *offset* 番目から最後の文字までの文字列を戻す。*replace* に文字列が指定されている場合は、*text* のその部分を replace の文字列に置き換える。	`Munger("the quick brown fox",5,25);` `"quick brown fox"` `Munger("the quick brown fox",5,25, "fast");` *`"the fast"`*
length に 0 を指定し、*replace* に文字列を指定しなかった場合、空の文字列を戻す。	`Munger("the quick brown fox", 1, 0);` *`""`*
length に 0 を指定し、*replace* に文字列を指定した場合、Munger は *offset* 番目の前に、その文字列を挿入する。	`Munger("the quick brown fox", 1, 0, "see ");` *`"see the quick brown fox"`*
length に負の整数を指定し、*replace* に文字列を指定しなかった場合、*text* の *offset* 番目から最後の文字までの文字列を戻す。	`Munger("the quick brown fox", 5, -5);` *`"quick brown fox"`*
length に負の整数を指定し、*replace* に文字列を指定した場合、Munger は、text の offset 番目から最後の文字までの文字列を、*replace* の文字列に置き換える。	`Munger("the quick brown fox", 5, -5, "fast");` *`"the fast"`*

Repeat

Repeat 関数は、第 1 引数のコピーを戻します。第 2（ときには第 3）引数は繰り返す回数です。これが 1 のとき、コピー回数は 1 回です。

第 1 引数が文字値またはリストのとき、結果はその文字のコピーになります。

```
repeat("abc",2)
   "abcabc"
repeat({"A"},2)
   {"A","A"}
repeat({1,2,3},2)
   {1,2,3,1,2,3}
```

第 1 引数が数値または行列のとき、結果は行列になります。第 2 引数は行の繰り返し回数で、第 3 引数で列の繰り返し回数を指定できます。指定されている引数が 2 つだけの場合、列の繰り返し回数は 1 になります。

```
repeat([1 2, 3 4],2,3)
   [  1 2 1 2 1 2,
      3 4 3 4 3 4,
      1 2 1 2 1 2,
      3 4 3 4 3 4]
repeat(9,2,3)
   [  9 9 9,
      9 9 9]
```

Repeat関数は、SAS/IML言語の同じ名前の関数と互換性がありますが、SASのDATAステップ関数とは互換性がありません。DATAステップ関数は、この関数より1回多く繰り返します。

パターンマッチと正規表現の使用

JSLのパターンマッチを使うと、文字列を柔軟に検索・処理することができます。

パターン変数は、JMPの他の変数と同様に定義・使用します。

```
i = 3; // 数値変数
a = "Ralph"; // 文字変数
t = textbox("Madge"); // ディスプレイボックス変数
p = ( "this" | "that" ) + patSpan(" ") + ( "car" | "bus" ); // パターン変数
```

上のステートメントを実行すると、pにパターン値が割り当てられます。パターン値は、他のパターンを作成したり、パターンマッチを実行したりするのに使用できます。patSpan関数は、引数で指定された文字列にマッチするパターンを戻します。たとえば、patSpan("0123456789")は0から9までの数字とマッチします。

```
p2 = "Take " + p + "."; // p を使って他のパターンを作成する
if( patMatch( "Take this bus.", p2 ), print("matches"), print("no match") ); //
   マッチングを実行する
```

上記のように、パターンがソーステキストとマッチすることだけを知りたい場合もあれば、マッチしたのがバスか車かなど、何がマッチしたのかを知りたい場合もあります。

```
p = ( "this" | "that" ) + patSpan(" ") + ( "car" | "bus" ) >?vehicleType; // パター
   ンがマッチした場合のみ、変数に値を割り当てる
if( patMatch( "Take this bus.", p ), show(vehicleType), print("no match") ); // パ
   ターンマッチしなかった場合は値が割り当てられないので、ELSE で vehicleType を使用しないこと
```

上のプログラムでは、ifにおいてパターンマッチの結果をチェックする必要がないので、変数(vehicleType)にデフォルト値をあらかじめ代入しておけばよいでしょう。条件割り当て演算子>?には2つのオペランドがあります。最初のオペランド（左辺）がパターンで、2番目（右辺）が変数名です。>?は、第1引数でパターンを作成し、全体のパターンマッチが成功したら、第1引数とのマッチングの結果を変数（第2引数）に保存します。>>も同様に変数に結果を保存しますが、全体のパターンマッチの成功を待ちません。>>に指定されたパターンがマッチするとそのたびに即座に割り当てが実行されます。

```
findDelimString = patLen(3)>>beginDelim + patArb()>?middlePart + expr(beginDelim);
testString = "SomeoneSawTheQuickBrownFoxJumpOverTheLazyDog'sBack";
rc = PatMatch( testString, findDelimString, "<<<" || middlePart || ">>>" );
show( rc, beginDelim, middlePart, testString );
```

上の例では、patMatch 関数の3番目の引数として置き換え文字列が使用されています。ここでは、3つの文字列を連結（|| 演算子）したものに置換されます。3つの文字列のうち、middlePartは、testStringから>?によって抽出されたものです。置き換えはパターンマッチが成功（rc == 1）しなければ実行されないため、このような指定は適切です。

findDelimStringに割り当てられたパターンを見てみましょう。3つのパターンの連結です。最初のPatLen(3)で任意の3文字とマッチさせ、その結果が>>演算子によりbeginDelimに割り当てられます。2番目は>?で任意の文字数の文字列とマッチさせ、>?演算子によりすべてのマッチングが成功した場合、結果をmiddlePartに割り当てます。最後は式で、パターンが作成されたときでは**なく**、パターンが実行されたときにbeginDelimに保持された文字列で構成されます。expr()と同様、引数の評価は実行時に行われます。それにより、パターンは、文中の2箇所に現れる同一の3文字の文字列を検索します。

適切なパターン関数を使うことにより、処理が速くなったり、プログラミングが簡単になったりする場合があります。たとえば、"a"|"b"|"c"はpatAny("abc")と等価です。patNotAny("abc")は"abc"を含まない文字列とマッチします。patBreak("0123456789")は、最初に出会った数字までの文字列(その数字は含まない)とマッチします。このようなパターンは、patNotAnyやPatBreak関数を用いずに記述するのは大変です。

次の例は、小数、指数、記号から構成される数値の文字列とマッチするパターンです。このパターンは、数字のない特殊なケースのマッチも行います。数字に割り当てられたパターンに特に注目してください。

```
digits = patSpan("0123456789") | "";

number = ( patAny("+-") | "" ) >?signPart +
         ( digits ) >?wholePart +
         ( "." + digits | "" ) >?fractionPart +
         ( patAny("eEdD") + ( patAny("+-") | "" ) + digits | "" ) >?exponentPart;

if( patMatch( "-123.456e-78", number ), show( signPart, wholePart, fractionPart,
  exponentPart ) );
```

データは固定フィールドで保存されている場合があります。patTab、patRTab、patLen、patPos、patRPos関数を使用すれば、固定フィールドの文字列を簡単に抽出できます。patTabとpatRTabは、文字列の左右の端から数えて何番目なのかを引数に取ります。指定のタブ位置までの文字列とマッチします。たとえば、次のような設定が行えます。

```
p = patPos(10) + patTab(15);
```

PatPos(10)は、10の位置にあるヌル文字列とマッチします。そのためマッチしたときには、位置10までマッチングが勧められます。そして、次にpatTab(15)が現在の位置（10）から位置15までテキストをマッチします。このパターンは、patPos(10)+patLen(5)と等価です。次に、別の例を示します。:

```
p = patPos(0) + patRTab(0);
```

この例は開始の 0 文字から終了の 0 文字まで、文字列全体をマッチさせます。patRem() 関数は、patRTab(0) の省略形で「文字列の残り」を意味し、引数を取りません。パターンマッチングは、次のように文字列の最初にアンカーを設定することもできます。

```
patMatch( "now is the time", patLen(15) + patRPos(0), NULL, ANCHOR );
```

上の例では置換後の文字列ではなく NULL を使用し、オプションで ANCHOR を使用しています。両方とも大文字です。NULL は置き換えが行われないことを意味します。ANCHOR はパターンマッチの開始位置が文字列の最初に固定されることを意味します。デフォルト値は UNANCHORED です。

次に示すパターンは、再帰的では**ありません**。

```
p = "a" | "b"; // 1 文字をマッチさせます。
p = p + p; // 2 文字
p = p + p; // 4 文字
patMatch( "babb", patPos(0) + p + patRPos(0) );
```

再帰的なパターンは expr() を使って現在の定義を参照します。

```
p = "<" + expr(p) + "*" + expr(p) + ">" | "x";
patMatch( "<<x*<x*x>>*x>", patPos(0) + p + patRPos(0) );
```

expr() はいわゆる「遅延」演算子であり、1 行目で p にパターンを割り当てる際には、expr(p) の（p）は評価されません。その次のステートメントでは patMatch によりパターンマッチが実行され、この時、expr() に遭遇するたびに引数の現在の値を参照します。この例では、マッチングの間の値は変化しません。p が自身によって定義されているなら、どのように処理されるのでしょうか。

p は 2 つの選択肢によって構成されます。右側の選択肢は簡単です。"x" 一文字です。左側は少し難解で、<p*p> というパターンです。そして、<p*p> の各 p は、"x" 一文字か、もしくは、<p*p> になっています。最後のいつくかの例では、パターンマッチがソーステキスト全体に対して行われるように、patPos(0) + ... + patRPos(0) を使用しています。テキスト全体にマッチングが必要な場合と、サブテキストだけにマッチングが必要な場合があります。ソーステキストを変更しながらこれらの例を試す場合は、テキスト全体へのマッチングを行った方が、何がマッチしたかを簡単に確認できるでしょう。patMatch の結果は 0 または 1 です。

次の例では「左の」再帰を使用しています。

```
x =  expr(x) + "a" | "b"; // + は | よりも優先される
```

patMatch 関数を FULLSCAN モードで使用した場合、マッチングが進められていく際にメモリが使い果たされたり、再帰の階層が膨れ上がってしまう可能性があります。ただし、デフォルトでは patMatch 関数は FULLSCAN を使用せず、ある仮定に基づいて再帰を停止させ、マッチングを成功させます。FULLSCAN モードで実行するには、patMatch 関数の第 4 以降の引数に FULLSCAN と指定します。先ほどの例のパターンは、「b」一文字か、「b」の後続がすべて「a」になっている文字列にマッチします。

```
rc = patMatch( "baaaaa", x );
```

パターンと大文字／小文字

正規表現と違って、パターンマッチでは大文字と小文字の区別がありません。大文字と小文字を区別させたい場合は、Pat Matchまたは Regex Matchに名前付き引数MATCHCASEを追加します。たとえば、次のような設定が行えます。

```
string = "abcABC";

result = Regex Match( string, Pat Regex( "[aBc]+" ) );
Show( string, result );
   string = "abcABC"
   result = {"abcABC"}

result = Regex Match( string, Pat Regex( "[cba]+" ), NULL, MATCHCASE );
Show( string, result );
   string = "abcABC"
   result = {"abc"}
```

正規表現

Regex()関数を使えば、標準的な正規表現をJMPでも使用することができます。

```
Regex(source, regular expression, <format>, <IGNORECASE>);
```

Sourceには、正規表現（regular expression）の適用先である文字列を指定します。両引数は、引用符付き文字列または引用符付き文字列への参照でなければなりません。

例:

```
Regex(
   " Are you there, Alice?, asked Jerry.", // ソース
   " (here|there).+ (\w+).+(said|asked) (\w+)\." );

   " there, Alice?, asked Jerry."
```

形式が指定されていないため、正規表現とマッチするテキストすべてが戻されます。オプションで形式（format）を指定すれば、ソースと正規表現を使用して新しい文字列を作成できます。

```
Regex(
   " Are you there, Alice?, asked Jerry.",
   " (here|there).+ (\w+).+(said|asked) (\w+)\.",
" I am \1, \4, replied \2." );

   " I am there, Jerry, replied Alice."
```

Formatのデフォルトは\0で、これはマッチした全体を意味します。\nは、正規表現によるn番目のマッチを指します。ソース内に正規表現とのマッチが見つからなかった場合、Regexは数値の欠測値を戻します。

正規表現には大文字と小文字の区別があります。区別させたくない場合は、オプションのスイッチ IGNORECASE を使用します。

ヒント：正規表現の詳細については、正規表現に関する文献を参照してください。

第 7 章

データ構造
さまざまなデータの処理

JSL では、次のようなデータ構造を使って 1 つの変数にさまざまなデータを入れることができます。

- リストは、入れ子構造のリストや式も含め、多数の値を保持できます。
- 行列は、行と列から成る数値のテーブルです。
- 連想配列は、値にキーをマップします。連想配列以外のほとんどのタイプのデータが使えます。

目次

リスト 147
リストの評価 147
リストを使った割り当て 148
リスト内の処理の実行 148
リスト内の項目の数を求める 148
添え字 148
リスト内で項目を検索する 149
リスト演算子 151
リスト内での反復 153
リストの連結 154
行列 154
行列の作成 155
添え字 156
問い合わせ関数 159
比較演算子、範囲チェック演算子、論理演算子 160
数値演算子 160
結合 163
Transpose（転置） 164
行列とデータテーブル 164
行列とレポート 167
Loc関数 167
順位づけと並べ替え 169
特殊な行列 170
逆行列と連立一次方程式 174
分解と正規化 178
ユーザ定義の行列演算子の作成 182
統計処理の例 183
連想配列 187

リスト

リストには、次のような項目を格納できます。

- 数値
- 変数
- 文字列
- 式（割り当てや関数呼び出しなど）
- 行列
- 入れ子構造のリスト

リストは次のいずれかの方法で作成します。

- List関数を使用する
- { } 中括弧を使用する

例

数値や変数を含むリストを作成するには、List()関数または中括弧を使用します。

```
x = List(1, 2, b);
x = {1, 2, b};
```

リストには、テキスト文字列、他のリスト、および関数呼び出しを含めることができます。

```
{"Red", "Green", "Blue", {1, "true"}, sqrt(2)};
```

リストに変数を入れ、同時にその変数に値を割り当てることができます。

```
x = {a=1, b=2};
```

リストの評価

リストを含んだスクリプトを実行すると、リストが戻り、リストの中の項目は評価されません。

```
b = 7;
x = {1, 2, b, Sqrt(3)};
Show(x);
   x = {1, 2, b, Sqrt(3)};
```

リスト内の項目を評価するには、Eval List関数を使います。

```
b = 7;
x = {1, 2, b, Sqrt(3)};
c = Eval List(x);
   {1, 2, 7, 1.73205080756888}
```

リストを使った割り当て

リストを使って変数に値を割り当てることができます。

例

```
{a, b, c} = {1, 2, 3}; // a に 1、b に 2、c に 3 を割り当てる
{a, b, c}--; // a、b、c を 1 ずつ減らす
{{a}, {b, c}}++; // a、b、c を 1 ずつ増やす
mylist = {1, log(2), e()^pi(), height[40]}; // 式を格納する
```

リスト内の処理の実行

リスト内で処理を実行することができます。

```
a = {{1, 2}, 3, {4, 5}};
b = {{10, 20}, 30, {40, 50}};
c = a + b;
   c = {{11, 22}, 33, {44, 55}}
```

リスト内の項目の数を求める

リスト内の項目の数を求めるには、`N Items()` 関数を使用します。

```
x = {1, 2, y, Sqrt(3), {a, b, 3}};
N = N Items(x);
Show(n);
   n = 5;
```

添え字

リストの添え字は、リストから指定した項目を抽出します。リスト内の複数の項目を戻すには、リストを添え字として使用します。

注：JSL は 1 からカウントを始めるため、リストの最初の要素は [1] となることに注意してください。他の一部の言語のように [0] とはなりません。

例

リスト a には 4 つの項目が含まれます。

```
a = {"bob", 4, [1,2,3], {x,y,z}};
Show( a[1] );
   a[1] = "bob";
Show( a[{1,3}] );
   a[{1, 3}] = {"bob", [1, 2, 3]};
a[2] = 5; // リストの 2 番目の項目に 5 を割り当てる
```

添え字は、リスト内の項目の選択または変更にも使用できます。

```
Show( a );
   a = {"bob", 5, [1, 2, 3], {x, y, z}};
c = {1, 2, 3};
c[{1, 2}] = {4, 4};
// c[{1, 2}] はどちらも 4 となる
Show( c );
   c = {4, 4, 3};
```

割り当てのリストや関数のリストの場合は、添え字に引用符付きの名前を使用できます。

```
x={sqrt(4), log(3)};
xx={a=1, b=3, c=5};
x["sqrt"];
   4
xx["b"];
   3
```

変数名は引用符で囲む必要があります。そうでないと、JMP は指定された変数を評価して、その値を使ってしまいます。次の例では、リスト内の a の値ではなく、リスト内の 2 番目の項目の値が戻されます。

```
a = 2;
Show(xx[a]);
   xx[a] = b = 3;
```

次の点を念頭に置いてください。

- 次のような状況では、左側に複数の添え字（たとえば、a[i][j] = value、ここで a は添え字可能な項目のリストを含む）を置くことができます。
  - 一番外側のレベル以外はすべてリストでなければなりません。前述の例では、a がリストでなければならず、a[i] は添え字可能なものであれば何でもかまいません。
  - 最後の添え字を除いて、添え字はすべて数値でなければなりません。前述の例では、i が数値でなければならず、j は行列またはリストでもかまいません。
- 添え字は入れ子構造のどのレベルに対しても実行できます。例:

  ```
  a[i][j][k][l][m][n] = 99;
  ```

リスト内で項目を検索する

リスト内の値を探すには、Loc() 関数または Contains() 関数を使用します。

```
Loc(list, value)
Contains(list, value)
```

Loc() および Contains() は、値の場所を戻します。Loc() は結果を行列で戻し、Contains() は結果を数値で戻します。

次の点を念頭に置いてください。

- Loc 関数は、反復する値をそのつど戻しますが、Contains() は、反復する値は最初の 1 度だけ戻します。
- 値が見つからなかった場合、Loc 関数は空の行列を戻し、Contains() はゼロを戻します。
- ある項目がリストに含まれているかどうかを評価するには、Loc() および Contains() を >0 とともに使用します。項目がリスト内にないときの戻り値はゼロです。項目がリスト内に最低 1 つは存在する場合は、1 が戻されます。

注：行列の処理に関する詳細や、行列を戻す Loc() コマンドについては、「行列」（154 ページ）を参照してください。

例

```
nameList = {“Katie”, “Louise”, “Jane”, “Jane”};
numList = {2, 4, 6, 8, 8};
```

nameList から "Katie" を検索します。

```
Loc(nameList, “Katie”);
   [1]
Contains(nameList, "Katie");
   1
```

nameList から "Erin" を検索します。

```
Loc(nameList, “Erin”);
   []
Contains(nameList, "Erin");
   0
```

nameList から数値の 8 を検索します。

```
Loc(numList, 8);
   [4, 5]
Contains(numList, 8);
   4
```

numList に数値の 5 があるかどうかを調べます。

```
NRow(Loc(numList, 5)) >0;
   0
Contains(numList, 5) >0;
   0
```

リスト演算子

表7.1は、リスト演算子とその構文を示しています。

表 7.1 リスト演算子

演算子および等価の関数		構文	説明
	As List()	As List(*matrix*)	行列をリストに変換して戻す。行列に複数の列がある場合は、各行を1つのリストとしたリストのリストを戻します。
= += -= *= /= ++ --	Assign() Add To() SubtractTo() MultiplyTo() DivideTo() Post Increment() Post Decrement()	{*list*} = {*list*} {*list*} += *value* {*list*} -= {*list*} ...	割り当て演算子の対象がリストで、割り当てられる値がリストの場合は、個々の項目について割り当てが行われる。左側にあるリストの最終的な値は、L-value（左辺に指定できるもの）、つまり値を割り当てることができる名前でなければなりません。 **注:** リストが等しいことを調べるときは、=ではなく、==を使います。
||	Concat()	Concat(*list1*, *list2*, ...);	最初のリストのコピーを戻します。その後に追加のリストが挿入されている場合はそのコピーも戻します。
	Eval List()	Eval List(*list*)	リスト（list）内の式を評価した後のリストを戻す。「リストの評価」（147ページ）を参照してください。
	Insert Into()	Insert Into(*list*, *x*, <*i*>)	リスト（*list*）の指定箇所（*i*）に新しい項目（*x*）を挿入する。*i*を指定しなかった場合、項目は最後尾に挿入されます。この関数は元のリストを変更します。
	Insert()	*list* = Insert(*list*, *x*, <*i*>)	リスト（*list*）の指定箇所（*i*）に新しい項目（*x*）を挿入した*list*を戻す。*i*を指定しなかった場合、項目は最後尾に挿入されます。この関数は元のリストを変更しません。

表 7.1 リスト演算子（続き）

演算子および等価の関数	構文	説明
Is List()	Is List(*arg*)	*arg*がList(items)または{items}で作成されたリストである場合に真（1）を戻し、そうでない場合に偽（0）を戻す。空のリストであってもリストなので、IsList({ })は真を戻します。miss=.（missという変数の値が欠測値である）の場合、IsList(miss)は欠測値ではなく偽を戻します。
{ } List	List(*a*, *b*, *c*) {*a*, *b*, *c*}	一連の項目を持つリストを作成する。項目は、他のリストを含め、どのような式でもかまいません。項目はカンマで区切る必要があります。テキストは二重引用符（" "）で囲むか、変数に保存して変数として呼び出す必要があります。
N Items	N Items(*list*)	指定されたリスト（*list*）内の項目数を戻す。変数に割り当てることも可能です。
Remove From()	Remove From(*list*, <*i*>, <*n*>)	リスト（*list*）の*i*番目から*n*個の項目を削除する。*n*を指定しなかった場合、*i*にある1項目だけが削除されます。*n*と*i*を指定しなかった場合、最後の1項目だけが削除されます。この関数は元のリストを変更します。
Remove()	Remove(*list*, <*i*>, <*n*>)	リスト（*list*）の*i*番目から*n*個の項目を削除した結果を戻す。*n*を指定しなかった場合、*i*にある1項目だけが削除されます。*n*と*i*を指定しなかった場合、最後の1項目だけが削除されます。この関数は元のリストを変更しません。
Reverse Into()	Reverse Into(*list*)	リスト（*list*）の項目の順序を逆にする。この関数は元のリストを変更します。
Reverse()	Reverse(*list*)	リスト（*list*）の項目の順序を逆にした結果を戻す。この関数は元のリストを変更しません。

表 7.1 リスト演算子（続き）

演算子および等価の関数	構文	説明
Shift Into()	Shift Into(*list*, <*n*>)	リスト（*list*）の最初の *n* 個の項目をリスト（*list*）の末尾に移動する。*n* を指定しなかった場合、最初の 1 項目だけを末尾に移動します。この関数は元のリストを変更します。
Shift()	Shift(*list*, <*n*>)	リスト（*list*）の最初の *n* 個の項目を末尾に移動した結果を戻す。*n* を指定しなかった場合、最初の 1 項目だけを末尾に移動します。この関数は元のリストを変更しません。
Sort Ascending()	Sort Ascending(*list*)	リスト（*list*）の項目を昇順に並べた結果を戻す。この関数は元のリストを変更しません。
Sort Descending()	Sort Descending(*list*)	リスト（*list*）の項目を降順に並べた結果を戻す。この関数は元のリストを変更しません。
Sort List Into()	Sort List Into(*list*)	リスト（*list*）の項目を昇順に並べる。この関数は元のリストを変更します。
Sort List()	Sort List(*list*)	リスト（*list*）の項目を昇順に並べた結果を戻す。この関数は元のリストを変更しません。
[ ] Subscript()	*list*[*i*] *x* = *list*[*i*] *list*[*i*] = *value* *a*[*b*, *c*] Subscript(*a*, *b*, *c*)	リスト（*list*）から *i* 番目の項目を抽出する。添え字は、リストまたは行列でもかまいません。
Substitute()	Substitute(*list*, *pattExpr1*, *replExpr1*, ...)	各パターン式のインスタンスに、対応する代替式を代入して、リストまたは式のコピーを戻す。
Substitute Into()	Substitute Into(*list*, *pattExpr1*, *replExpr1*, ...)	各パターン式のインスタンスに、対応する代替式を代入して、リストまたは式を変更する。**注:** リスト（*list*）または式は変数でなければなりません。

リスト内での反復

リスト内で反復処理を行い、各値を操作したり、特定の値を検索したりできます。次のスクリプトは、リスト内の各項目を調べ、項目が 10 以下の場合はその値の 2 乗を戻します。

```
x = {2, 12, 8, 5, 18, 25};
n = N Items (x);
for (i=1, i<=n, i++,
 if (x[i]<=10, x[i]=x[i]^2)
);
Show (x)
   x = {4, 12, 64, 25, 18, 25};
```

Loc() を使うと、新しいリストの中で25と等しい項目を検索できます。

```
Loc (x,25)
   [4, 6] // リストの 4 番目と 6 番目の項目が 25 と等しい
```

リストの連結

2つ以上のリストを1つのリストに連結するには、Concat() または || 演算子を使用します。

次の例は、Concat() を使ってリスト *a* とリスト *b* を連結しています。

```
a = {1, 2};
b = {7, 8, 9};
Concat( a, b );
   {1, 2, 7, 8, 9}
```

次の例は、同じ2つのリストを || 演算子を使って連結しています。

```
{1, 2} || {7, 8, 9}
   {1, 2, 7, 8, 9}
```

異なる種類（文字列の行列と数値の行列など）を連結することもできます。

```
d = {"apples", "bananas"};
e = {"oranges", "grapes"};
f = {1, 2, 3};
Concat( d, e, f);
   {"apples", "bananas", "oranges", "grapes", 1, 2, 3}
```

行列

行列とは、数値の行と列を長方形に配列したものを指します。行列に数値を格納すれば、その数値と行列代数を使った計算が実行できます。

この節では、次の点に注意してください。

- 行列は、太字で書かれた大文字の変数で表されます（例: **A**）。
- 行または列が1つしかない行列を**ベクトル**といいます（具体的には、それぞれ**行ベクトル**または**列ベクトル**）。

- わかりやすいように、ここではベクトルの行列を小文字の太字で表します（例: **x**）。
- **スカラー**は行列内にない数値です。

行列の作成

行列を作成するときは、次の点に注意してください。

- 行列の文字は角括弧で囲みます。`[...]`
- 行列内の数値として、小数部のある値や、負または正の値、また科学表記も使用できます。
- 列の項目は空白のスペースで区切ります。空白のスペースはいくつでも使用できます。
- 行はカンマで区切ります。

高度な行列を作成するには、「特殊な行列」（170 ページ）を参照してください。

例

3行、2列の行列 **A** を作成します。

```
A = [1 2, 3 4, 5 6];
```

R は行ベクトルで **C** は列ベクトルです。

```
R = [10 12 14];
C = [11, 13, 15];
```

B は 1x1 の行列、つまり、1行1列の行列です。

```
B = [20];
```

E は空の行列です。

```
E = [];
```

オプションで、空の行列内の行数と列数を指定できます。JMP は、必要に応じて行列を作成します。

スクリプトは空の行列を戻すことができます。「Big Class.jmp」において、次の式は、年齢が 8 歳に等しい行を探し、何も検出されなかったので空の行列を戻しています。

```
a = dt << Get Rows Where( 年齢 == 8 );
Show( a );
   a = [](0,1);
```

詳細については、「互換性に関するメモ」の章の「空の行列」（632 ページ）を参照してください。

リストから行列を作成する

リストを行列に変換するには、`Matrix()` 関数を使用します。リストが1つの場合は、1つの列ベクトルに変換されます。リストが複数ある場合は、行に変換されます。

次の例は、1つのリストから列ベクトルを作成します。

```
A = matrix({1,2,3});
   [1,2,3]
```

次の例は、リストのリストから行列を作成します。各リストが行列の各行に対応します。

```
A = matrix({{1,2,3}, {4,5,6}, {7,8,9}});
   [1 2 3,
   4 5 6,
   7 8 9]
```

式から行列を作成する

式から行列を作成するには、Matrix() を使用します。要素に指定した式は、評価すると数値になる必要があります。

```
A = matrix({4*5, 8^2, sqrt(9)});
   [20, 64, 3]
```

添え字

添え字演算子（[ ]）を使うと、行列から要素や部分行列を取り出せます。Subscript() 関数は通常、対象となる行列の後に、行と列を表す引数を大括弧で囲んで記述します。

単一要素の抽出

A[i,j] は、行列Aからi行j列の要素を取り出し、スカラーとして戻します。同じ機能を持つ関数は、Subscript(A,i,j) です。

```
P=[1 2 3, 4 5 6, 7 8 9];
P[2,3]; // 6を戻す
Subscript(P,2,3); // 6を戻す
```

変数testに、行列**A**の3行目の1列目の値（5）を割り当てます。

```
A = [1 2, 3 4, 5 6];
test = A[3,1];
Show(test);
   test = 5;
```

行列やリストによる添え字

部分行列を抽出するには、行列やリストによる添え字を使用します。選択された行と列の行列が戻されます。次の式では、2行目と3行目の2列目と1列目を抽出します。

```
P=[1 2 3, 4 5 6, 7 8 9];
P[[2 3],[1 3]]; // 添え字を行列で指定
P[{2,3},{1,3}]; // 添え字をリストで指定
```

この2つはどちらも次のような結果になります。

```
[4 6,
7 9]
```

1つしか引数をとらない添え字

添え字の引数を1つだけ指定した場合、その添え字は、すべての行を連結して1つの行にした形での通し番号になります。このため、引数を2つ指定する場合のA[i,j]は、引数を1つにした場合のA[(i-1)*ncol(A)+j]と同じになります。

例

```
Q = [2 4 6,8 10 12,14 16 18];
Q[8]; // Q[3,2] と同じ
   16
```

以下の例は、すべて列ベクトル[10, 14, 18]を戻します。

```
Q = [2 4 6, 8 10 12, 14 16 18];
Q[{5, 7, 9}];
Q[[5 7 9]];
ii = [5 7 9];
Q[ii];
ii = {5, 7, 9};
Q[ii];
Subscript( Q, ii );
```

以下のスクリプトでは、行列**P**の1～9番目までの値を順に戻します。

```
P = [1 2 3, 4 5 6, 7 8 9];
For( i = 1, i <= 3, i++,
   For( j = 1, j <= 3, j++,
      Show( P[i, j] )
   );
);
```

行および列の削除

行または列の削除は、空の行列をその行または列に割り当てることによって実行できます。

```
A[k, 0] = [];  // k 番目の行を削除する
A[0, k] = [];  // k 番目の列を削除する
```

行全体または列全体を選択する

添え字の引数を0にすると、すべての行または列を指定できます。

```
P = [1 2 3, 4 5 6, 7 8 9];
```

2列目を選択します。

```
P[0, 2];
   [2, 5, 8]
```

3列目と2列目を選択します。

```
P[0, [3, 2]];
   [3 2, 6 5, 9 8]
```

3行目を選択します。

```
P[3, 0];
   [7 8 9]
```

2行目と3行目を選択します。

```
P[[2, 3], 0];
   [4 5 6, 7 8 9]
```

すべての列と行を選択します（行列全体）。

```
P[0, 0];
   [1 2 3, 4 5 6, 7 8 9]
```

添え字を使った代入

添え字を使って、行列内の値を変更することができます。添え字として、スカラーや、行列またはリスト、またはすべての行や列を意味する数0を指定できます。挿入する行や列の次元は、挿入される要素の次元と一致しているか、または、要素は指定された箇所に繰り返し挿入される必要があります。

例

2行目の3列目の値を99に変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 3] = 99;
Show( P );
   P=[1 2 3, 4 5 99, 7 8 9]
```

4つの位置の値を変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[[1 2], [2 3]] = [66 77, 88 99];
Show( P );
   P=[1 66 77, 4 88 99, 7 8 9]
```

1つの列にある3つの値を変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[0, 2] = [11, 22, 33];
Show( P );
   P=[1 11 3, 4 22 6, 7 33 9]
```

1つの行にある3つの値を変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[3, 0] = [100 102 104];
Show( P );
   P=[1 2 3, 4 5 6, 100 102 104]
```

1行のすべての値を同じ値に変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 0] = 99;
Show( P );
   P=[1 2 3, 99 99 99, 7 8 9]
```

代入演算子

行列や行列の添え字に、代入演算子（+=など）を使用することができます。たとえば、次の例では、行列のi行j列の要素に1を加えます。

```
P=[1 2 3, 4 5 6, 7 8 9];
P[1,1]+=1;
Show(P);
   P=[2 2 3,
      4 5 6,
      7 8 9];

P[1,1]+=1;
Show(P);
   P=[3 2 3,
   4 5 6,
   7 8 9];
```

行と列の範囲指定

範囲を指定した行列を作成するには、Index() 関数 :: を使用します。

```
T1=1::3; // ベクトル [1 2 3] を作成する
T2=4::6; // ベクトル [4 5 6] を作成する
T3=7::9; // ベクトル [7 8 9] を作成する
T=T1|/T2|/T3; // ベクトルを 1 つの行列に連結する
T[1::3,2::3];// 1 行目から 3 行目の 2 列目から 14 列目を参照する
   [2 3, 5 6, 8 9]
T[index(1,3), index(2,3)]; // Index 関数と等価
   [2 3, 5 6, 8 9]
```

問い合わせ関数

NCol() 関数と NRow() 関数は、それぞれ行列（またはデータテーブル）の列数または行数を戻します。

```
NCol([1 2 3,4 5 6]); // 列が 3 つなので 3 を戻す
NRow([1 2 3,4 5 6]); // 行が 2 つなので 2 を戻す
```

値が行列であるかどうかを調べるには、Is Matrix() 関数を使います。引数が行列を参照している場合は 1 を戻します。

```
A = [20, 64, 3];
B = {20, 64, 3};
IsMatrix(A); // 真の場合は 1 を戻す
IsMatrix(B); // 偽の場合は 0 を戻す
```

比較演算子、範囲チェック演算子、論理演算子

JMP の比較演算子、範囲チェック演算子、および論理演算子は、行列に対しても使えます。要素にブール値を持つ行列が結果として戻されます。整合する行列を比較できます。

```
A<B; // より小さい
A<=B; // 以下
A>B; // より大きい
A>=B; // 以上
A==B; // 等しい
A!=B; // 等しくない
A<B<C; // 連続的な比較（範囲チェック）
A|B; // 論理和 OR
A&B; // 論理積 AND
```

Any() 演算子や All() 演算子を使うと、行列の比較結果を要約できます。Any() は、0 以外の要素が 1 つでもあれば 1 を戻します。All() は、要素がすべて 0 以外のときに 1 を戻します。

```
[2 2]==[1 2]; // 結果は [0 1] なので
All([2 2]==[1 2]); // 結果は 0
Any([2 2]==[1 2]); // 結果は 1
```

Min() と Max() は、引数として与えられた行列について、それぞれ、要素の最小値と最大値を戻します。

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
B=[0 1 2, 2 1 0, 0 1 1, 2 0 0];
Min(A); // 結果は 1
Max(A); // 結果は 12
Min(A,B); // 結果は 0
```

数値演算子

行列に対して、数値の演算（引き算、足し算、掛け算など）を実行できます。統計的手法の多くは、簡潔な行列表記で表現し、JSL に組み込むことができます。

たとえば次の式は、最小2乗回帰を行列の乗算と逆行列で表したものです。

$$b = (X'X)^{-1}X'y$$

この式を、次のJSL式に組み込みます。

```
b = Inv(X`*X)*X`*y;
```

数値計算の基本

行列に対して、次の基本的な算術を実行できます。

- 足し算
- 引き算
- 掛け算
- 割り算（逆数を掛ける）

注：標準的な乗算は、行列の掛け算であり、要素ごとの掛け算ではないことに注意してください。

行列の掛け算を実行するには、次のいずれかの方法を使用します。

- * 演算子
- `Multiply()` 関数
- `Matrix Mult()` 関数

行列の割り算を実行するには、次のいずれかの方法を使用します。

- / 演算子
- `Divide()` 関数

行列の掛け算および割り算では、次のことに注意してください。

- スカラーの掛け算または割り算は可換（ab = ba）ですが、行列の掛け算または割り算は**可換ではない**ことに注意してください。
- 2つの要素のうちの1つがスカラーの場合、要素ごとの掛け算または割り算が実行されます。
- 要素ごとの掛け算をする場合、`:*` または `EMult()` 関数を使用します。
- 要素ごとの割り算をする場合、`:/` か、または同じ作用を持つ `EDiv()` 関数を使用します。

例

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
B=[0 1 2, 2 1 0, 0 1 1, 2 0 0];
C=[1 2 3 4, 4 3 2 1, 0 1 0 1];
D=[0 1 2, 2 1 0, 1 2 0];
```

行列の足し算

```
R = A+B;
   [1 3 5,
   6 6 6,
   7 9 10,
   12 11 12]
```

行列の引き算

```
R = A-B;
   [1 1 1,
   2 4 6,
   7 7 8,
   8 11 12]
```

行列の掛け算（**A** の行と **C** の列の内積）

```
R = A*C;
   [9 11 7 9,
   24 29 22 27,
   39 47 37 45,
   54 65 52 63]
```

行列の割り算（`A*Inverse(D)` と等価）

```
R = A/D;[1.5 0.5 0,
   3 2 0,
   4.5 3.5 0,
   6 5 0]
```

行列の要素ごとの掛け算

```
R = A:*B;
   [0 2 6,
   8 5 0,
   0 8 9,
   20 0 0]
```

行列とスカラーの掛け算

```
R = C*2;
   [2 4 6 8,
   8 6 4 2,
   0 2 0 2]
```

行列のスカラーによる割り算

```
R = C/2;
   [0.5 1 1.5 2,
   2 1.5 1 0.5,
   0 0.5 0 0.5]
```

行列の要素ごとの割り算（ゼロで割った場合は欠測値を戻す）

```
R = A:/B;
   [.2 1.5,
   2 5 .,
   .8 9,
   5 ..]
```

行列に対する数値（スカラー）関数の使用

数値関数は、行列の各要素に対して作用します。純粋な数値関数の多くは行列にも適用でき、結果は行列として得られます。行列とスカラーを混在させてもかまいません。

数値関数の例には次のようなものがあります。

- Sqrt()、Root()、Log()、Exp()、^ Power()、Log10()
- Abs()、Mod()、Floor()、Ceiling()、Round()、Modulo()
- Sine、()Cosine()、Tangent()、ArcSine()、およびその他の三角関数
- Normal Distribution()、および他の確率関係の関数

例

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = Sqrt(A);    // 各要素ごとに平方根をとる
   [1 1.414213562373095 1.732050807568877,
      2 2.23606797749979 2.449489742783178,
      2.645751311064591 2.82842712474619 3,
      3.16227766016838 3.3166247903554 3.464101615137754]
```

結合

Concat() 関数は、2つの行列を横に結合し、1つの行列にします。2つの行列の行の数は一致している必要があります。縦の2重バー（||）は、Concatと同じ機能を持つ二項演算子です。

```
Identity(2)||J(2,3,4);
   [1 0 4 4 4,
   0 1 4 4 4]

B=[1,1]; B || Concat(Identity(2),j(2,3,4));
   [1 1 0 4 4 4,
   1 0 1 4 4 4]
```

VConcat() 関数は、2つの行列を縦に結合し、1つの行列にします。2つの行列の列の数は一致している必要があります。縦のバーとスラッシュの組み合わせ（|/）は、VConcatと同じ機能を持つ二項演算子です。

```
Identity(2) |/ J(3,2,1);  // または VConcat(Identity(2),J(3,2,1));
   [1 0,
   0 1,
   1 1,
   1 1,
   1 1]
```

Concat() と VConcat() はともに、空行列やスカラー、リストとの結合ができます。

```
a=[];
a || {1}; // 生成される行列は [1]
a || {2}; // 生成される行列は [2]
a || [3 4 5]; // 生成される行列は [3 4 5]
```

結合のための代入演算子として、||= と |/= があります。それぞれ、Concat To() 関数および V Concat To() 関数と等価です。

- a||=b は a=a||b と等価です。
- a|/=b は a=a|/b と等価です。

Transpose（転置）

Transpose() 関数は、行列の行と列を入れ換えます。逆引用符（`）は、Transpose() と同じ作用をする接尾演算子です。行列表記では、Transpose() は、一般的にプライム符号や肩文字の T を使って表現されるもの（A′ または A^T）と同じです。

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
A`;
   [1 4 7 10,
   2 5 8 11,
   3 6 9 12]
Transpose([1 2,3 4]);
   [1 3,
   2 4]
```

行列とデータテーブル

JMP のデータテーブルと行列との間で情報を移動させることができます。行列代数を使って、JMP データテーブルにある数値に対して独自の計算を実行し、その結果を JMP データテーブルに戻して保存することができます。

データテーブルから行列へデータを移動させる

ここでは、データテーブルのデータを行列に移動する方法について説明します。

すべての数値を移動させる

Get As Matrix() 関数は、データテーブルまたは列のすべての数値を含む行列を生成します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
A = dt<<GetAsMatrix;

dt=Open("$SAMPLE_DATA¥Big Class.jmp");
col=Column("身長 ( インチ )");
A = col<<GetAsMatrix;
```

すべての数値および文字列を移動させる

Get All Columns As Matrix() 関数は、文字列を含め、データテーブルのすべての列の値を行列で戻します。文字型の列には、名前の昇順に 1 から順に番号が付けられます。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
A = dt<<GetAllColumnsAsMatrix;
```

特定の列のみを移動させる

データテーブルの特定の列だけを取得するには、列リスト引数（名前または文字）を使用します。

```
dt=current data table();
x=dt<<Get As Matrix({"身長 ( インチ )", "体重 ( ポンド )"});
```

または

```
x=dt<<Get As Matrix({Name("身長 ( インチ )"), Name("体重 ( ポンド )")});
```

現在選択されている行

データテーブルで現在選択されている行の行列を求めるには:

```
dt<<Get Selected Rows
```

注： どの行も選択されていない場合は、空の行列が戻されます。

行の位置を検索する

式が真となる行番号の行列を求めるには:

```
dt<<Get Rows Where (expression)
```

たとえば、次のスクリプトは性別が男性（M）である行番号を戻します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
A = dt<<GetRowsWhere( 性別 =="M");
```

行列のデータをデータテーブルに移動させる

ここでは、行列のデータをデータテーブルに移動させる方法について説明します。

列ベクトルを移動させる

SetValues() 関数は、列ベクトルの値を既存のデータテーブル列にコピーします。

```
col<<SetValues(x);
```

col はデータテーブル列への参照で、**x** は列ベクトルです。

たとえば、次のスクリプトは、**test** という新しい列を作成し、ベクトル **x** の値を **test** 列にコピーします。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
dt<<New Column ("test");
col=Column("test");
x = 1::40;
col<<SetValues(x);
```

行列のすべての値を移動させる

Set Matrix() は、既存のデータテーブルに、行列の値を格納するのに必要なだけの新しい行と列を作成して、行列の値をコピーします。新しい列には、「**列 1**」、「**列 2**」という名前が付けられます。

```
dt=New Table("B");
dt<<Set Matrix([1 2 3 4 5, 6 7 8 9 10]);
```

このスクリプトは、2 つの行と 4 つの列を含んだ B という新しいデータテーブルを作成します。

行列引数から新しいデータテーブルを作成するには、As Table(matrix) コマンドを使用します。列には、「**列 1**」、「**列 2**」という名前が付きます。たとえば、次のスクリプトは、**A** の値を含んだ新しいデータテーブルを作成します。

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
dt=As Table(A);
```

列の要約

行列における各列の要約統計量を、行ベクトルで戻す関数がいくつか用意されています。

```
mymatrix = [11 22, 33 44, 55 66];
V Max(mymatrix) // 各列の最大値を戻す
   [55 66]
V Min(mymatrix) // 各列の最小値を戻す
   [11 22]
V Mean(mymatrix) // 各列の平均値を戻す
   [33 44]
V Sum(mymatrix) // 各列の合計を戻す
   [99 132]
V Std(mymatrix) // 各列の標準偏差を戻す
   [22 22]
```

行列とレポート

レポートから値を取り出して行列を作成できます。まず、取得する項目の位置を指定する必要があります。この情報はレポートのツリー構造の中です。

次のスクリプトを実行して、「二変量の関係」レポートにパラメータ推定値のテーブルを作成します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
biv = dt<<Bivariate(X(:Name(" 身長 ( インチ )")),Y(:Name(" 体重 ( ポンド )")),Fit Line);
```

次に、ツリー構造を開いて、パラメータ推定値を含んでいる項目を識別します。

- グレーの開閉アイコンを右クリックし、［**編集**］>［**ツリー構造の表示**］を選択します。

  パラメータ推定値は NumberColBox(13) に含まれています。次のようにスクリプトを続けます。

```
colBox=Report(biv) [NumberColBox(13)];
beta = colBox<<GetAsMatrix;
   [-127.145248610915,3.711354893859555]
```

次の点を念頭に置いてください。

- 変数にテーブルボックスへの参照が含まれている場合、Get As Matrix() は、テーブル内の全数値列の値を使って行列 **A** を作成します。

```
A = tableBox<<GetAsMatrix;
```

- 変数にレポートテーブルの数値列への参照が含まれている場合、Get As Matrix() は、列の値を使って列ベクトルの行列 **A** を作成します。

```
A = colBox<<GetAsMatrix;
```

Loc 関数

Loc()、Loc Nonmissing()、Loc Min()、Loc Max()、および Loc Sorted() 関数はすべて、行列内の特定の値の位置を示す行列を戻します。

Loc()

Loc() 関数は、行列 A において要素が「真」となっている位置を示す行列を作成します。つまり、行列 **A** において、「0 以外かつ欠測値以外」である要素の位置を示す行列を作成します。

```
A = [0 1 .3 4 0];
B = [2 2 0 0 5 6];
```

次の例は、**A** において、0 以外かつ欠測値以外である要素のインデックスを戻します。

```
I = Loc(A);
   [2, 4, 5]
```

次の例は、**B** の対応する要素よりも小さい **A** の要素のインデックスを戻します。2つの行列の行と列は同数でなければならないことに注意してください。

```
I = Loc(A<B);
   [1, 5, 6]
```

次のスクリプトは、**A** の4より小さい要素をすべて0に置き換えます。

```
A = [0 1 0 3 4 0];
A[Loc( A < 4 )] = 0;
Show( A );
   A = [0 0 0 0 4 0];
```

Loc Nonmissing()

Loc Nonmissing() 関数は、欠測値を含まない行列の行番号のベクトルを戻します。例:

```
A = [1 2 3, 4 .6, 7 8 ., 8 7 6];
Loc Nonmissing( A );
   [1, 4]
```

Loc Min() と Loc Max()

Loc Min() 関数と Loc Max() 関数は、それぞれ行列の最初の最小要素および最大要素の位置を戻します。行列の要素には、先頭行の最初の列から始め、左から右に向かって連続して番号が付けられます。

```
A = [1 2 2, 2 4 4, 1 1 1];
B = [6, 12, 9];
Show( Loc Max( A ) );
Show( Loc Min( B ) );
   Loc Max(A) = 5;
   Loc Min(B) = 1;
```

Loc Sorted()

Loc Sorted() 関数は、主に、ある数値がどのくらいの位置にあるかを調べるために使用します。この関数は、行列 **B** で与えられた値以下である値の中で最大のものが、**A** のどの位置にあるのかを戻します。結果のベクトルには、**B** の各要素に対応する項目が含まれます。

```
A = [10 20 30 40];
B = [35];
LocSorted(A,B);
   [3]

A = [10 20 40];
B = [35 5 45 20];
LocSorted(A,B);
   [2, 1, 3, 2]
```

次の点を念頭に置いてください。

- **A** は昇順に並んでいる必要があります。
- 戻り値は必ず 1 以上の整数です。**B** の要素が **A** のどの要素よりも小さい場合、関数は 1 の値を戻します。**B** の要素が **A** のどの要素よりも大きい場合、関数は *n* を戻します。*n* は **A** の要素の数です。

順位づけと並べ替え

Rank() 関数は、ベクトルまたはリスト内で昇順に並べた場合の各数値の位置を戻します。

```
E=[1 -2 3 -4 0 5 1 8 -7];
R=Rank(E);
   [9,4,2,5,7,1,3,6,8]
```

E を小さい値から順に並べ替えた場合、最初の数値は -7 です。**E** における -7 の位置は 9 番目です。

行列 **R** を **E** の添え字として利用すれば、元の行列 **E** を昇順に並べ替えることができます。

```
sortedE=E[R];
   [-7,-4,-2,0,1,1,3,5,8]
```

Ranking Tie() 関数は、ベクトルまたはリスト内での値の順位を戻します。同順位のデータに対しては平均順位が与えられます。同様に、Ranking() も、ベクトルまたはリスト内での値の順位を戻しますが、同順位のデータに対しては任意の順序が与えられます。

```
E=[1 -2 3 -4 0 5 1 8 -7];
Ranking Tie (E);
   [5.5, 3, 7, 2, 4, 8, 5.5, 9, 1]

E=[1 -2 3 -4 0 5 1 8 -7];
Ranking (E);
   [5, 3, 7, 2, 4, 8, 6, 9, 1]
```

Sort Ascending() 関数および Sort Descending() 関数は並べ替えベクトルです。

```
E=[1 -2 3 -4 0 5 1 8 -7];
Sort Ascending (E);
   [-7 -4 -2 0 1 1 3 5 8]

E=[1 -2 3 -4 0 5 1 8 -7];
Sort Descending (E);
   [8 5 3 1 1 0 -2 -4 -7]
```

引数がベクトルかリストではない場合、エラーメッセージが表示されます。

特殊な行列

単位行列の作成

Identity() 関数を使うと、任意の次元で単位行列を作成できます。単位行列とは、対角線上の要素が1で、それ以外の要素が0の正方行列です。引数には次元数のみを指定します。

```
Identity(3);
   [1 0 0,
   0 1 0,
   0 0 1]
```

指定の値を持つ行列の作成

関数 J() は、第1引数を行数、第2引数を列数、第3引数を全要素の値とする行列を作成します。

```
J(3,4,5);
   [5 5 5 5,
   5 5 5 5,
   5 5 5 5]
J(3,4,random normal());
   [0.407709113182904 1.67359154091978 1.00412665221308 0.240885679837327,
   -0.557848036549455 -0.620833861982722 0.877166783247633 1.50413740148892,
   -2.09920574748608 -0.154797501010655 0.0463943433032137 0.064041826393316]
```

対角行列の作成

Diag() 関数は、（同数の行と列を持つ）正方行列やベクトルから対角行列を作成します。対角行列とは、非対角要素がすべて0の正方行列です。

```
D=[ 1 -1 1];
Diag(D);
   [1 0 0,
   0 -1 0,
   0 0 1]
Diag([1,2,3,4]);
   [1 0 0 0,
   0 2 0 0,
   0 0 3 0,
   0 0 0 4]

A=[1 2,3 4];
f=[5];
D=Diag(A,f);
   [1 2 0
   3 4 0
   0 0 5]
```

三つ目の例では、一見すると、すべての非対角要素がゼロというわけではありません。行列表記を使用すると、次のようになります。

```
[A 0,
0` f]
```

ここで、Aとfは例の行列で、0はゼロの列ベクトルです。

対角要素からの列ベクトルの作成

VecDiag()関数は、行列の対角要素から成るベクトルを作成します。

```
v=vecdiag(
[1 0 0 1,
5 3 7 1,
9 9 8 8,
1 5 4 3]);
   [1,3,8,3]
```

2次形式の計算

VecQuadratic()関数は、回帰分析における予測の標準誤差や、外れ値に関するMahalanobis距離やT2乗統計量などを計算します。Vec Quadratic(Sym, X)は、VecDiag(X*Sym*X`)を計算するのと同じです。第1引数は対称行列でなければなりません。通常は、共分散行列の逆行列を指定します。第2引数は、列数が第1引数の行列と等しい矩形行列です。

対角要素の合計を戻す

Trace()関数は、正方行列の対角要素の和を戻します。

```
D=[0 1 2, 2 1 0, 1 2 0];
trace(D); // 1を戻す
```

整数値の行ベクトルの作成

Index()関数は、最初の引数から最後の引数までの整数値を要素とする行ベクトルを作成します。2重コロン::は等価の二項演算子です。

```
6::10;
   [6 7 8 9 10]
Index(1,5);
   [1 2 3 4 5]
```

オプションの第3引数を指定して、増分をデフォルト値の+1から変更することもできます。

```
Index(0.1, 0.4, 0.1);
   [0.1, 0.2, 0.3, 0.4]
```

増分は負の数でもかまいません。

```
Index(6, 0, -2);
   [6, 4, 2, 0]
```

デフォルトの増分は1、または、最初の引数が2番目の引数よりも大きい場合は-1です。

行列の再構築

Shape()関数は、既存の行列を指定の次元に作り直します。たとえば次のように指定すると、3x4の行列であるaが12x1の行列に変わります。

```
a=[1 1 1, 2 2 2, 3 3 3, 4 4 4];
shape(a, 12, 1)
   [1,1,1,2,2,2,3,3,3,4,4,4]
```

計画行列の作成

Design()関数は、ベクトルまたはリストの計画行列を作成します。引数の一意の値ごとに1つの列を作成します。計画行列の要素は0または1です。たとえば、下の**x**は、1、2、3の値から成っているため、計画行列には1の列、2の列、そして3の列があることになります。行列の各行には、行の値に対応する列に1が入ります。次の例では、第1行（1）に対しては、1の列（第1列）に1が入り、他の列には0が入ります。第2行（2）に対しては、2の列に1が入り、他の列には0が入ります。以下、同じような操作で行列が作成されます。

```
x=[1, 2, 3, 2, 1];
Design(x);
   [1 0 0,
   0 1 0,
   0 0 1,
   0 1 0,
   1 0 0]
```

この変形としてDesignNom()関数またはDesign F()関数があり、これは最後の列を削除して、それを他の列の値から引きます。したがって、DesignNom()またはDesign F()でできる行列の要素は、0、1、または–1となります。また、DesignNom()またはDesign F()でできた行列の列の数は、元のベクトルが持つ要素の種類の数より1つ少なくなります。この演算子は、効果のフルランクの計画行列を作成します。

```
x=[1, 2, 3, 2, 1];
DesignNom(x);
   [1 0,
   0 1,
   -1 -1,
   0 1,
   1 0]
```

DesignNom()については、「分散分析（ANOVA）の例」（184ページ）で詳しく説明しています。

順序尺度の因子を簡単にコーディングするには、DesignOrd() 関数を使います。この関数は、ベクトルまたはリストの一意の値の数より 1 つ少ない列を使ってフルランクのコーディングを作成します。低水準の行をすべてゼロにし、それに続く水準は計画行列の行にひとつずつ 1 を加えていきます。

```
x=[1, 2, 3, 4, 5, 6];
DesignOrd(x);
   [0 0 0 0 0,
   1 0 0 0 0,
   1 1 0 0 0,
   1 1 1 0 0,
   1 1 1 1 0,
   1 1 1 1 1]
```

Design()、DesignNom()、および DesignOrd() では、全水準の値とその順序を指定する第 2 引数をとることができます。この機能により、たった 1 行に対する計画行列も作成することができます。

- Design(values, levels) は計画行列を作成します。
- DesignNom(values, levels) はフルランクの計画行列を作成します。

次の点を念頭に置いてください。

- *values* 引数は、単一の要素でも、要素の行列またはリストでもかまいません。
- *levels* 引数は、計画行列作成の対象となる全水準を値として持つリストまたは行列です。
- 作成される行列は、*values* 引数として指定した値の要素数と同じ行数になります。
- 作成される行列は、*levels* 引数として指定した項目数と同じ列数になります。DesignNom() と DesignOrd() の場合は、*levels* 引数として指定した項目数より 1 つ少ない列数になります。
- 値が見つからない場合は、行全体がゼロになります。

例

```
Design(20, [10 20 30]);
   [0 1 0]
Design(30, [10 20 30]);
   [0 0 1]
DesignNom(20, [10 20 30]);
   [0 1]
DesignNom(30, [10 20 30]);
   [-1 -1]
DesignOrd(20, [10 20 30]);
   [1 0]
Design([20, 10, 30, 20], [10 20 30])
   [0 1 0,
   1 0 0,
   0 0 1,
   0 1 0]
```

```
Design({"b", "a", "c", "b"}, {"a", "b", "c"});
   [0 1 0,
   1 0 0,
   0 0 1,
   0 1 0]
```

直積を求める

Direct Product() 関数は、2つの行列の直積（Kronecker積）を求めます。行列と行列の直積は行列となり、その要素は、**A** と **B** の要素の積となります。

```
G=[1 2,
   3 5];
H=[2 3,
   5 7];
Direct product(G,H);
   [2 3 4 6,
   5 7 10 14,
   6 9 10 15,
   15 21 25 35]
```

H Direct Product() 関数は、行数の等しい2つの行列の行ごとの直積を求めます。

```
HDirectProduct(G,H);
   [2 3 4 6,
   15 21 25 35]
```

HDirect Product() は、計画行列における交互作用の列を作る際に便利です。

```
XA = Design Nom(A);
 XB = Design Nom(B);
 XAB = HDirect Product(XA,XB);
 X = J(NRow(A),1)||XA||XB||XAB;
```

逆行列と連立一次方程式

JMPには、Inverse()、GInverse()、およびSweep() という、逆行列を計算するための関数があります。Solve() 関数は、連立一次方程式の式を解くのに使用します。

Inverse または Inv

Inverse() 関数は、正則な正方行列の逆行列を戻します。Inverse() は、省略して Inv と書くこともできます。行列**A**において、**A** とInverse(**A**)の積(よくA(A^{-1})と記載される)は、結果として単位行列を戻します。

```
A=[5 6,7 8];
AInv=Inv(A);
A*AInv;
   [1 0,0 1]
```

```
A=[1 2,3 4];
AInv=Inverse(A);
A*AInv;
   [1 1.110223025e-16,0 1]
```

注：2 番目の例にあるように、浮動小数点の精度の限界が原因で、値がわずかに異なる場合があります。

GInverse

行列 **A** の（Moore-Penrose 型の）一般逆行列とは、次のような行列 **G** です。

AGA=A

GAG=G

(**AG**)´ = **AG**

(**GA**)´ = **GA**

`GInverse()` 関数は、非正方行列を含めた任意の行列を引数にとり、特異値分解を使って Moore-Penrose 型の一般逆行列を求めます。この関数は、フルランクでない行列の逆行列を求めるときに便利です。次の方程式系を見てください。

$$\begin{aligned} x + 2y + 2z &= 6 \\ 2x + 4y + 4z &= 12 \\ x + y + z &= 1 \end{aligned}$$

この方程式系の解を求めるには、次のスクリプトを使用します。

```
A=[1 2 2, 2 4 4, 1 1 1];
B=[6,12,1];
Show(GInverse(A)*B);
   G Inverse(A)*B=[-4,2.5,2.5]
```

Solve

`Solve()` 関数は、連立一次方程式の解を求めます。`Solve()` は、$x = A^{-1}b$ となるベクトル **x** を求めます（ここで、**A** は正方行列、**b** はベクトル）。この行列 **A** とベクトル **b** の行数は同じである必要があります。`Solve(A,b)` は `Inverse(A)*b` と同じです。

```
A=[1 -4 2, 3 3 2, 0 4 -1];
b=[1, 2, 1];
x=solve(A,b);
   [-16.9999999999999, 4.99999999999998, 18.9999999999999]
A*x;
   [1, 2, 0.999999999999997]
```

注：例にあるように、浮動小数点の精度の限界が原因で、値がわずかに異なる場合があります。

Sweep

Sweep() 関数は、正方行列の部分（**ピボット**）の逆行列を求めます。これをすべてのピボットに対して行うと、最終的には逆行列が求められます。通常、この行列は対角軸が 0 にならないよう、正定値行列（または負定値行列）である必要があります。Sweep() は、与えられた行列が正定値行列であるかどうかはチェックしません。正定値行列で**ない**場合も、掃き出し法において、対角軸に 0 以外の値が現れる限り演算は続けられます。0（または 0 に近い値）が対角軸に現れた場合、その行と列を 0 にします。結果として g2 型の一般逆行列が求められます。

Sweep 関数について

行列 **E** が、次に示すように、さらに小さな行列 **A**、**B**、**C**、**D** に分割されているとします。

$$E = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Sweep() の構文は次のようになります。

```
Sweep(E, [...]); // ここで [...] は行列 A の部分を示す
```

これにより、次のような結果の行列が作成されます。

$$\begin{bmatrix} A^{-1} & A^{-1}B \\ -CA^{-1} & D - CA^{-1}B \end{bmatrix}$$

次の点を念頭に置いてください。

- **A** の位置の部分行列は逆行列になります。
- 部分行列 **B** の位置は $Ax = B$ の解になります。
- 部分行列 **C** の位置は $xA = C$ の解になります。

Sweep 関数の使用法

Sweep() は逐次的で可逆的です。

- A=Sweep(A,{i,j}) は A=Sweep(Sweep(A,i),j) と同じで、逐次的です。
- A=Sweep(Sweep(A,i),i) は **A** を元の値に復元します。これは可逆的です。

次のような交差積の部分から成る行列があるとします。

$$C = \begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}$$

$X'X$ を引数とした Sweep 演算の結果は、次のようになります。

$$\begin{bmatrix} (X'X)^{-1} & (X'X)^{-1}X'y \\ -yX(X'X)^{-1} & y'y - y'X(X'X)^{-1}X'y \end{bmatrix}$$

統計的な観点からは、次のように見ることができます。

- 右上部はモデル Y = Xb + e の最小2乗推定値
- 右下部は誤差の平方和
- 左上部は推定値の共分散に比例する行列

Sweep 関数は、ステップワイズ回帰で必要な部分的な解を求める際に有効です。

2つ目の引数は、掃き出しを行う行（列）を列挙したベクトルです。たとえば、**E** が 4x4 の行列で、4 行すべてに掃き出し法を適用して、$\mathbf{E}^{-1}$ を得るには、以下のように指定します。

```
E=[ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
sweep(E,[1,2,3,4]);
   [0.56 -0.44 -0.02 -0.02,
   -0.44 0.56 -0.02 -0.02,
   -0.02 -0.02 0.34 -0.16,
   -0.02 -0.02 -0.16 0.34]
inverse(E); // これらの結果は同じ
   [0.56 -0.44 -0.02 -0.02,
   -0.44 0.56 -0.02 -0.02,
   -0.02 -0.02 0.34 -0.16,
   -0.02 -0.02 -0.16 0.34]
```

注： Sweep() の詳細と、逆行列を作成する Gauss-Jordan 法との関連については、Goodnight, J.H. (1979) "A tutorial on the SWEEP operator." *The American Statistician*, Augus 1979, Vol. 33, No. 3. pp. 149-58 を参照してください。

Sweep() については、「分散分析（ANOVA）の例」（184 ページ）でさらに説明しています。

行列式（Determinant）

Det() 関数は、正方行列の行列式を戻します。2 x 2 行列の行列式は、下に示すように、対角線上の要素の積の差です。*n* x *n* 行列の行列式は、(n - 1) × (n - 1) の行列式の重み和として再帰的に定義されます。逆行列を作成するには、行列の行列式が 0 以外でなければなりません。

$$\begin{vmatrix} 1 & 2 \\ 3 & 5 \end{vmatrix} = (1 \cdot 5) - (3 \cdot 2) = -1$$

```
F=[1 2,3 5];
Det(F);
   -1
```

分解と正規化

この節では、固有値および固有ベクトルを計算する関数と、行列を分解する関数について説明します。

固有値

Eigen() 関数は、対称行列の固有値分解を実行します。固有値分解は、主成分分析と正準相関分析をはじめとするさまざまな統計的手法で用いられています。主成分分析と正準相関分析は、最大の固有値に対応した変換が分散を最大化する変換となっています。

Eigen() は行列のリストを戻します。戻されたリストの最初の行列は固有値の列ベクトルで、2 番目の行列には列として固有ベクトルが含まれています。

```
A=[ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
Eigen(A);
   {[10, 5, 2, 1]
   [0.632455532033676 - 0.316227766016838 - 2.77555756156289e-16
      -0.707106781186547, 0.632455532033676 - 0.316227766016837
      - 1.66533453693773e-16 0.707106781186547, 0.316227766016838
      0.632455532033676 0.707106781186548 0, 0.316227766016837 0.632455532033676 -
      0.707106781186547 0]}
```

この関数は行列のリストを結果として戻します。結果を受け取るには、一方には固有値のベクトル、もう一方には固有ベクトルの行列が代入されるように、2 つのグローバル変数のリストに代入してください。

```
{evals,evecs}=Eigen(A);
```

固有値分解をすることで、*n* x *n* の行列 **A** に対する方程式 Ax = λx がゼロベクトルでない解 **x** を持つときの、すべての λ（ラムダ）とベクトル **x** が求められます。この λ を固有値、**x** ベクトルを固有ベクトルと呼びます。これは (A - λI)**x** = 0 を解くことと同じです。下のようなスクリプトで、固有値と固有ベクトルから **A** を再構成できます。

```
newA=evecs*diag(evals)*evecs`;
```

固有値および固有ベクトルに関しては、次のことに注意してください。

- 固有ベクトルの行列は直交行列で、転置行列が逆行列になります（E′E = EE′ = I）。
- 固有値が一意であるときのみ、固有ベクトルも一意に求められます。
- 固有値がゼロの行列は、特異行列です。
- 固有値の逆行列と固有ベクトルにより、逆行列も求められます。Moore-Penrose の一般化逆行列は、ゼロ以外の固有値の逆行列から求められます （「GInverse」（175 ページ）を参照）。

注：非常に小さい固有値を 0 とみなすかどうかを判断する必要があります。

- 固有値分解をすることで、正方行列の掛け算を次のように考えることができます。
  - 回転（直交行列を掛ける）
  - スケーリング（対角行列を掛ける）

– 逆回転（直交行列の逆行列、つまり転置行列を掛ける）

A*x = E`*diag(M)*E*x;//E で回転、diag(M) でスケーリング、E` で逆回転

Cholesky 分解

Cholesky() 関数は Cholesky 分解を行います。半正定値行列 **A** を、非特異下三角行列 **L** とその転置行列の積の形、**L*L'** = **A** に分解します。

```
E=[ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
L=Cholesky(E);
   [2.23606797749979 0 0 0,
   1.788854381999832 1.341640786499874 0 0,
   0.447213595499958 0.149071198499986 1.9436506316151 0,
   0.447213595499958 0.149071198499986 0.914659120760047 1.71498585142509]
```

結果を確認するには、次のように入力します。

```
L*L`;
   [5 4 1 1,
   4 5 1 1,
   1 1 4 2,
   1 1 2 4]
```

Cholesky 分解について

Cholesky() は、行列の式を扱いやすい形式に再構成するのに便利です。たとえば、JMP では固有値を求めるには、行列が対称行列である必要があります。したがって、対称行列の積 **AB** の固有値は直接求められない場合があります （**A** と **B** が対称行列であっても、その積が対称行列であるとは限らないからです）。しかし、この問題は、**L'BL** の固有値の問題に置き換えることができます。ここで、**L** は **A** の Cholesky 根です。L`BL は AB と同じ固有値を持ちます。

Cholesky() の他の使用法としては、Trace() 式で行列の対角和を求めるときに、行列の順序を変えるということが考えられます。Trace(A*B*A`) などの式で、**A** の行数が多い場合、大きな計算量を要することになります。しかし、**B** が LL' と Cholesky 分解できるのであれば、元の式は Trace(A*L*L`*A`) と書き直せます。これは、Trace(L`A`*AL) と等しく、したがって **AL** の要素の平方和を求めるだけでよくなるので、演算量はかなり少なくなります。

Chol Update() 関数を使用すると、Cholesky 分解を効率的に更新できます。*n* x *n* 行列 **A** の Cholesky 根を **L** とした場合、cholUpdate(L, C, V) を呼び出すと、A+V*C*V' の Cholesky 根が算出されます。**C** は *m* x *m* の対称行列、**V** は *n* x *m* 行列です。

例

Cholesky 分解を手動で更新する手順は次のとおりです。

```
exS = [16 1 0 11 -1 12, 1 11 -1 1 -1 1, 0 -1 12 -1 1 0, 11 1 -1 11 -1 9, -1 -1 1 -1
   9 -1, 12 1 0 9 -1 12];
// Cholesky 分解を実行する
exAchol = Cholesky( exS);

// 計画行列に 2 つの列ベクトルを追加する
exV = [1 1, 0 0, 0 1, 0 0, 0 0, 0 1];

/* 最初の列ベクトルは、計画行列の行の 1 つに足される。
2 番目の列ベクトルは、計画行列の行の 1 つから引かれる */ */
exC = [1 0, 0 -1];

// Cholesky 分解を手動で更新する
exAnew = exS + exV * exC * exV`;
exAcholnew = Cholesky( exAnew);
```

手動で更新するのではなく、Chol Update() を使って Cholesky 分解をより効率的に更新する手順は、次のとおりです。

```
// Cholesky 分解をより効率的に更新する
exAcholnew_test =
Chol Update( exAchol, exV, exC );

// 結果は手動での手順の場合と同じ
Show( exAcholnew_test );
Show( exAcholnew );
```

特異値分解

SVD() 関数を使うと、行列の特異値分解が求められます。つまり、行列 **A** について、SVD() は U*diag(M)*V`=A を満たす 3 つの行列、**U**、**M**、**V** のリストを戻します。

次の点を念頭に置いてください。

- 結果の行列 **M** では、余計な 0 の対角要素は省かれています。
- 特異値分解によって、**A** は USV′ の形で表されます。ここで、
  - **U** と **V** は、互いに直交している列ベクトルを含んでいる行列です（「直交するベクトル」とは、「直角」もしくは「独立」なベクトルのことを指します）。
  - **S** は *n* x *n* 対角行列で、**A** の特異値、つまり **A′A** の固有値の負でない平方根を要素として持っています。
- 特異値分解は対応分析などで使われています。

例

```
A=[1 2 1 0,
   2 3 0 1,
   1 0 1 5,
   0 1 5 1];
{U,M,V}=svd(A);
newA=U*diag(M)*V`;
   [1 2 0.999999999999997 -2.99456640040496e-15,
   2 3 -1.17505831453979e-15 1,
   0.999999999999997 -2.16851276518826e-15 0.999999999999999 5,
   2.22586706011274e-15 1 5 0.999999999999997]
```

正規直交化

Ortho() 関数は、列の直交化を行い、さらにベクトルの大きさで割って正規化します。この関数は Gram-Schmidt法を使用します。直交行列の列ベクトルは、大きさが単位長で互いに直交（内積が0）します。

```
B= ortho([1 -1,1 0,0 1]);
   [0.408248290463863 -0.707106781186548, 0.408248290463863 0.707106781186548,
      -0.816496580927726 3.14018491736755e-16]
```

各列ベクトルが直交していることは、**B** と **B** の転置行列を掛け合わせると単位行列になることから確かめられます。

```
C=B`*B;
   [1 -3.119061760824e-16, -3.119061760824e-16 1]
```

特に指定がない限り、ベクトルは正規化されます。つまり、ベクトルの大きさでベクトルそのものを割るスケーリングで、大きさ1のベクトルを作ります。Scaled(0) を指定して、スケーリングをしない場合は、次のようになります。

```
ortho([1 -1,1 0,0 1], scaled(0));
   [0.408248290463863 -0.353553390593274, 0.408248290463863 0.353553390593274,
      -0.816496580927726 1.57009245868377e-16]
```

全要素の和が0になるベクトルを作るには、オプションの Centered(1) を指定します。このオプションは対比の行列を作る場合に便利です。

```
result=ortho([1 -1,1 0,0 1], centered(1));
   [0.408248290463863 -0.707106781186548, 0.408248290463863 0.707106781186548,
      -0.816496580927726 3.14018491736755e-16]
```

各列の要素の和が0になることは、要素がすべて1のベクトルを前から掛けて列の要素の和を求めることで確認できます。

```
J(1,3)*result;
   [1.11022302462516e-16 2.02996189274239e-16]
```

直交多項式

OrthoPoly() 関数は、引数で指定された次元までのベクトルの直交多項式 を戻します。直交多項式は、多項式モデルを普通にあてはめると回帰係数の間に強い相関が生じてしまう場合に有用です。

```
OrthoPoly([1 2 3],2);
   [-0.707106781186548 0.408248290463862, 0 -0.816496580927726, 0.707106781186548
      0.408248290463864]
```

次元はベクトルの次元より小さいものである必要があります。「正規直交化」（181 ページ）の項にあるように、Scaled(1) を指定することで単位長のベクトルを作れます。

QR 分解

QR() 分解は、数値的に安定した行列の処理に便利です。QR() は 2 つの行列のリストを戻します。一般的な使い方は次のとおりです。

```
{Q, R} = QR(X);
```

Q と R に結果が入ります。*m* x *n* 行列 X に対して、QR() は X = Q*R と分解します。ここで、Q は、*m* x *m* の直交行列、R は *m* x *n* の上三角行列です。

逆行列の更新

M`**M** 行列の逆行列に行を追加または削除するには、Inv Update(S, X, w) 関数を使用します。逆行列の更新は、1 行除去による影響診断や候補計画の評価に役立ちます。

次の点を念頭に置いてください。

- 第 1 引数 **S** は更新する行列です。
- 第 2 引数 **X** は、追加または削除する行を含む行列です。
- 第 3 引数 w には、行を追加する場合は 1、削除する場合は -1 を指定します。
- 複数の行を追加または削除できます。

Inv Update(S, X, w) 関数は、次の式を計算するのと同じです。

```
S-w*S*X`*Inv(I+w*X*S*X`)*X*S
```

ここで、**I** は単位行列、Inv(A) は **A** の逆行列です。

ユーザ定義の行列演算子の作成

ユーザ自身が定義した演算子をマクロに登録できます。「プログラミング手法」の章の「マクロ」（212 ページ）を参照してください。同様に、カスタム行列演算子も作成できます。たとえばベクトルの大きさを求める行列演算子 Mag() を定義するには、次のように記述します。

```
mag=function({x},sqrt(x`*x));
```

同様に、大きさでベクトルを割るNormalizeを定義するには、次のように記述します。

```
normalize=function({x},x/sqrt(x`*x));
```

統計処理の例

この節では、行列を使った統計処理の例を紹介します。

回帰の例

回帰分析の計算手法について、JMPに組み込まれている機能を使わず、ユーザ自身が独自の方法を実施したい場合を考えてみましょう。そのような場合、簡潔な行列表現を使って、ほんの数行で書くことができます。

```
Y = [ 98,112.5,84,102.5,102.5,50.5,90,77,112,150,128,133,85,112];
X = [65.3,69,56.5,62.8,63.5,51.3,64.3,56.3,66.5,72,64.8,67,57.5,66.5];
X = J(nrow(X),1) || X;    // 切片の列に 1 を入れる
beta = Inv(X`*X)*X`*Y;    // 最小 2 乗推定
resid = Y-X*beta;         // 残差、Y - 推定値
sse = resid`*resid;       // 誤差平方和
show(beta,sse);
```

これは、データテーブルからデータを取得し、関連する統計量を計算してレポートウィンドウに表示するスクリプトへと拡張できます。

```
// データテーブルを開く
bigClass = open("$SAMPLE_DATA¥Big Class.jmp");

// データを行列に移す
x = (Column("年齢")<<getValues) || (Column("身長(インチ)")<<getValues);
x = j(nrow(x),1,1)||x;
y = Column("体重(ポンド)")<<getValues;

// 回帰分析の計算
xpxi = Inv(x`*x);
beta = xpxi*x`*y;         // パラメータ推定値
resid = y-x*beta;         // 残差
sse = resid`*resid;       // 誤差平方和
dfe = nrow(x)-ncol(x);    // 自由度
mse = sse/dfe;            // 誤差の平均平方、誤差分散推定値

// 推定値の追加計算
stdb = sqrt(vecDiag(xpxi)*mse); // 推定値の標準誤差
alpha = .05;
qt = Students t Quantile(1-alpha/2,dfe);
betau95 = beta+qt*stdb;          // 上側 95%信頼限界
betal95 = beta-qt*stdb;          // 下側 95%信頼限界
tratio = beta:/stdb;             // Student の t 値
probt = (1-TDistribution(abs(tratio),dfe))*2; // p 値
```

```
// 結果の表示
newWindow(" ビッグ クラスの回帰分析 ",
   tableBox(
      StringColBox(" 項 ",{" 切片 "," 年齢 "," 身長 ( インチ )"}),
      NumberColBox(" 推定値 ",   beta),
      NumberColBox(" 標準誤差 ",  stdb),
      NumberColBox("t 値 ",     tratio),
      NumberColBox("p 値 ",     probt),
      NumberColBox(" 下側 95%", betal95),
      NumberColBox(" 上側 95%", betau95)));
```

分散分析（ANOVA）の例

行列演算を用いて一元配置分散分析も実行できます。この例では、3水準の要因（薬品の投与量で低用量、中用量、高用量）と 1 つの応答変数から成る問題を扱います。つまりこの例は、次のような一般線形モデルで表されます。

$$Y = \mathbf{a} + bX + \mathbf{e}$$

ここで

- **Y** は応答変数のベクトル
- *a* は切片項
- **b** は係数から成るベクトル
- **X** は要因の計画行列
- *e* は誤差項

```
factor=[1,2,3,1,2,3,1,2,3];
y=[1,2,3,4,3,2,5,4,3];
```

まず、要因の計画行列を作成する必要があります。

```
designNom(factor);
   [1 0,
   0 1,
   -1 -1,
   1 0,
   0 1,
   -1 -1,
   1 0,
   0 1,
   -1 -1]
```

次に、切片項として、計画行列に1だけから成る列を加えます。これを行うには、単にJとDesign Nom()を結合するだけです。

```
x = J(9,1,1) || designNom(factor);
   [1 1 0,
   1 0 1,
   1 -1 -1,
   1 1 0,
   1 0 1,
   1 -1 -1,
   1 1 0,
   1 0 1,
   1 -1 -1]
```

ここで、一般的な方程式を解くために、次のような行列**M**を作成する必要があります。

$$\begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}$$

行列**M**は、次のような結合をすれば1ステップで作成できます。

```
M=(x`*x || x`*y)
        |/
  (y`*x || y`*y);
   [  9 0 0 27,
      0 6 3 2,
      0 3 6 1,
      27 2 1 93]
```

モデル全体の推定結果は、行列**M**において、$\boldsymbol{X'X}$の部分をすべて掃き出せば得られます。また、最初の列だけ掃き出せば切片のみのモデルの推定結果が得られます。

```
FullFit=sweep(M,[1,2,3]);     // フルモデルのあてはめ
InterceptOnly=sweep(M,[1]);   // 切片のみのモデル
```

分散分析の一部の結果は、これら2つのモデルを比較して計算されます。ここでは、フルモデル（切片と傾きを含んだモデル）の結果を見てみましょう。フルモデルに関して掃き出された行列は、次のような行列になります。

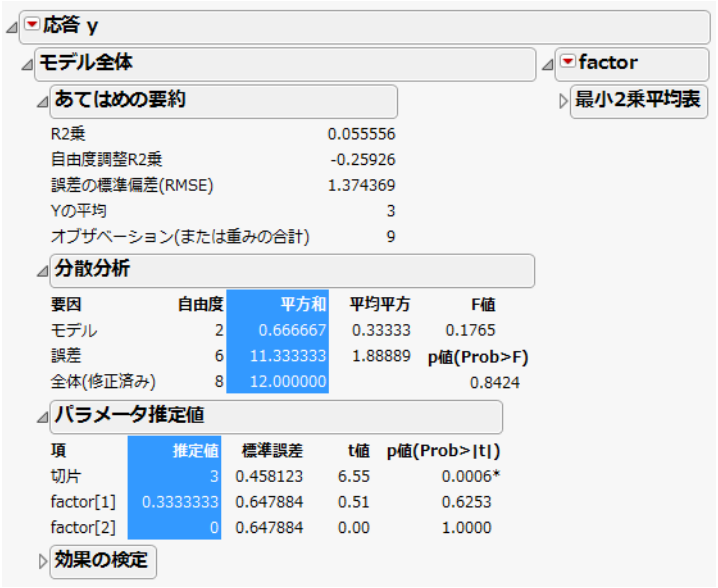
```
[0.111111111111111 0 0 3,
0 0.222222222222222 -0.11111111111111 0.333333333333333,
0 -0.11111111111111 0.222222222222222 0,
-3 -0.33333333333333 0 11.33333333333333]
```

行列右上部のモデル係数を見てください。左下部の係数も、符号が正負逆になっている以外は同じ、3, 0.333, 0 となっています。結果は次のように解釈できます。

- 切片項の係数は 3 です。
- 要因の 1 番目の水準の係数は 0.333 です。
- 2 番目の水準の係数は 0 です。
- `Design Nom()` を使っているので、3 番目の水準の係数は -0.333 です。
- 行列右下部には誤差平方和 11.333 が入ります。

このプログラム例では、特定の数値を指定していますが、それらを適切な引数に置き換えることにより、より汎用性があるスクリプトに変更できます。上の結果は、モデルのあてはめプラットフォームの結果と一致しています。図 7.1 を参照してください。

図 7.1 モデルのあてはめの ANOVA レポート



以下に、図 7.1 に示すレポートを作成する方法を示します。

1. 「データテーブル」（257 ページ）の章で説明している方法で、データテーブルを作成します。

```
factor = [1, 2, 3, 1, 2, 3, 1, 2, 3];
Y = [1, 2, 3, 4, 3, 2, 5, 4, 3];
dt = New Table( "foo" );
dt << New Column( "y", Set Values( ::y ) );
dt << New Column( "factor", Nominal, Values( ::factor ) );
```

2. 「プラットフォームのスクリプト」の章の「プラットフォームの起動」（361 ページ）で説明している方法で、モデルのあてはめを実行します。

```
Fit Model(
   Y( :y ),
   Effects( :factor ),
   Personality( Standard Least Squares ),
   Run Model(
      y << {Plot Actual by Predicted( 0 ), Plot Residual by Predicted( 0 ),
      Plot Effect Leverage( 0 )}
   );
);
```

3. 「表示ツリー」の章の「ディスプレイボックスオブジェクトの参照」（384 ページ）で説明している方法で、ディスプレイを操作します。

```
ranova = Report( Fit Least Squares[1] );
ranova[OutlineBox(5)] << Close(0);
ranova[OutlineBox(6)] << Close(1);
ranova[OutlineBox(8)] << Close(1);
ranova[OutlineBox(4), NumberColBox( 2 )] << select;
ranova[OutlineBox(5), NumberColBox( 1 )] << select;
```

連想配列

連想配列は、値に固有のキーを関連付けます（固有でない場合もあります）。連想配列は、辞書、マップ、ハッシュマップ、またはハッシュテーブルと呼ばれることもあります。キーは引用符で囲みます。キーに関連付けられる値は、数値、日付、行列、リストなどです。

注：行列もリストも、キーと値の両方に使用できますが、行列とリストを混在させることはできません。つまり、行列をキーとして使用し、リストを値として使用することはできません。その逆も同様です。

通常、連想配列には順序がありませんが、JMP の連想配列では、反復処理や逐次処理のプログラミングのために、キーをアルファベット順（文字コード順）で戻します。

大量のリストの場合、代わりに連想配列を使った方がより効率的で高速です。

連想配列の作成

空の連想配列を作成するには、`Associative Array()` 関数または `[=>]` を使用します。

```
cary = Associative Array();
cary = [=>];
   [=> ]
```

キーと値には、任意のJSLオブジェクトが使用できます。項目は添え字を使った指定により、追加・変更できます。

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
```

デフォルト値

連想配列にないキーの値は、デフォルト値によって決定します。連想配列にないキーを参照しようとすると、エラーが生じます。連想配列にデフォルト値を定義した場合は、存在しないキーを参照すると、次のように動作します。

- そのキーを連想配列に追加する
- デフォルト値をその新しいキーに割り当てる
- エラーではなく、新しいキーの値（デフォルト値）を戻す

キーに値を割り当てないで文字列のリストから連想配列を作成した場合、キーには1という値が割り当てられます。この連想配列のデフォルト値は0に設定されます。

デフォルト値を設定するには、次のようにします。

```
cary = Associative Array();
cary << Set Default Value("Cary, NC");
```

連想配列にデフォルト値が設定されているかどうかを確かめるには、<<Get Default Value メセージを使用します。

```
cary << Get Default Value
   "Cary, NC"
```

デフォルト値がない場合は、Empty() が戻されます。

Set Default Value メッセージ以外に、連想配列のリテラルを指定する際にキーなしの=>value を使用することによってもデフォルト値を設定できます。

```
counts = ["a"=>10, "b"=>3, =>0]; // デフォルト値は 0
counts["c"] += 1;
Show (counts);
   ["a" => 10, "b" => 3, "c" => 1, => 0]
```

1行目はデフォルト値を0に設定します。2行目はキー "c" が counts 内にないことを示します。結果として、デフォルト値0を持つキー "c" を作成し、1だけ増やします。

連想配列作成用関数

空の連想配列を作成します。

```
map = [=>];
map = Associative Array();
```

デフォルト値を持つ空の連想配列を作成します。

```
map = [=>0];
map = Associative Array(0);
```

値を指定して連想配列を作成します。

```
map = ["yes" => 0, "no" => 1];
```

各キーの値およびデフォルト値を指定して連想配列を作成します。

```
map = ["yes" => 0, "no" => 1, => 2];
```

キーと値のセットを表すリストから、連想配列を作成します。

```
map = Associative Array({{"yes", 0}, {"no", 1}});
```

キーと値のセットを表すリストにデフォルト値の指定も加えて、連想配列を作成します。

```
map = Associative Array({{"yes", 0}, {"no", 1}}, 2);
```

キーのリストと値のリストから、連想配列を作成します。

```
map = Associative Array({"yes", "no"}, {0, 1});
```

キーのリストと値のリストおよびデフォルト値を指定して連想配列を作成します。

```
map = Associative Array({"yes", "no"}, {0, 1}, 2);
```

2つの列参照から、連想配列を作成します。1番目の列がキー、2番目の列が値になります。

```
map = Associative Array(:name, :height);
```

2つの列参照およびデフォルト値を指定して連想配列を作成します。

```
map = Associative Array(:name, :height, .);
```

キーの1つのリストまたはキーの1つの列参照から、連想配列を作成します。この場合、デフォルト値は0になります。

```
set = Associative Array({"yes", "no"});
set = Associative Array(:name);
```

連想配列の使用

キー数の検出

連想配列に含まれているキー数を調べるには、N Items() 関数を使います。

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
N Items(cary);
   4
```

キーおよび値の追加と削除

連想配列へのキーと値のペアの追加や削除には、次の関数を使用します。

- Insert()
- Insert Into()
- Remove()
- Remove From()

次の点を念頭に置いてください。

- Insert() と Remove() は、キーと値のペアを追加または削除した連想配列のコピーを戻します。
- Insert Into() と Remove From() は、指定の連想配列に対して、直接キーと値のペアの追加または削除を行います。
- Insert() と Insert Into() は、連想配列、キー、および値の3つの引数を取ります。
- Remove() と Remove From() は、連想配列とキーの2つの引数を取ります。
- 値を指定しないでキーを挿入した場合は、キーには値1が割り当てられます。

例

次の例は、Insert() と Insert Into() を説明しています。

```
newcary = Insert(cary, "time zone", "Eastern");
show(cary, newcary);
   cary = Associative Array({
   {"high schools",{"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}
   });
   newcary = Associative Array({
```

```
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"time zone", "Eastern"},
   {"weather", "sunny"}
   });

Insert Into(cary, "county", "Wake");
show(cary);
   cary = Associative Array({
   {"county", "Wake"},
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}
   });
```

aa << Insertは連想配列に送られるメッセージで、Insert Into()関数と同じ処理を実行します。たとえば、次の2つのステートメントは同じ結果になります。

```
cary << Insert("county", "Wake");
Insert Into(cary, "county", "Wake");
```

次の例は、RemoveとRemove Fromを説明しています。

```
newcary = Remove(cary, "high schools");
show(cary, newcary);
   cary = Associative Array({
   {"county", "Wake"},
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}
   })
   newcary = ["county" => "Wake", "population" => 116244, "state" => "NC", "weather"
      => "sunny"];

Remove From(cary, "weather");
show(cary);
   cary = Associative Array({
   {"county", "Wake"},
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"}
   });
```

aa << Removeは連想配列に送られるメッセージで、Remove From()関数と同じ処理を実行します。たとえば、次の2つのステートメントは同じ結果になります。

```
cary << Remove("weather");
Remove From(cary, "weather");
```

連想配列内のキーまたは値の検出

連想配列の中に特定のキーが含まれているかどうかを調べるには、Contains()を使用します。

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
Contains(cary, “high schools”);
   1
Contains(cary, “lakes”);
   0
```

連想配列に含まれるすべてのキーのリストを取得するには、<< Get Keysメッセージを使用します。

```
cary <<Get Keys;
   {"high schools", "population", "state", "weather"}
```

連想配列に含まれるすべての値のリストを取得するには、<< Get Valuesメッセージを使用します。

```
cary <<Get Values;
   {{"Cary", "Green Hope", "Panther Creek"}, 116244, "NC", "sunny"}
```

特定のキーの値だけを取得するには、キーを引数として指定します。その際、キーはリストで指定する必要があります。

```
cary <<Get Values({"state", "population"});
   {"NC", 116244}
```

1つのキーの値を取得するには、<<Get Valueメッセージを使用します。指定できるキーは1つだけで、リストを指定することはできません。

```
cary <<Get Value("weather");
   "sunny"
```

連想配列に含まれるすべてのキーと値のペアのリストを取得するには、<< Get Contentsメッセージを使用します。

```
cary <<Get Contents;
   {{"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}}
```

注：<<Get Contents メッセージを使用して取得したリストには、デフォルト値は含まれません。キーはアルファベット順（文字コード順）で表示されます。

連想配列内の反復

連想配列の中を反復させるには、<<First メッセージと <<Next メッセージを使用します。<<First は、連想配列の中の最初のキーを戻します。<<Next(key) は、引数として指定されたキー（*key*）の後のキーを戻します。

次のコマンドは、連想配列 cary からキーと値のペアをすべて削除し、空の連想配列を残します。

```
currentkey = cary <<First;
total = N Items(cary);
for (i = 1, i <= total, i++,
   nextkey = cary<<Next(currentkey);
   Remove From (cary, currentkey);
   currentkey = nextkey;
);
Show(cary);
   cary = [=>];
```

連想配列の応用

連想配列を使用すると、いろいろなタスクをすばやく効率的に実行することができます。

データテーブル列から一意の値を取得する

連想配列内では 1 つのキーを一度しか使用できないため、列の値を連想配列に入れれば自動的に一意の値となります。たとえば、「Big Class.jmp」サンプルデータテーブルには行が 40 個あります。「**身長 (インチ)**」列に一意の値がいくつあるかを調べるには、次のスクリプトを実行します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
unique heights = associative array(dt:Name("身長 ( インチ )"));
nitems(unique heights);
   17
```

「**身長 (インチ)**」の一意の値は 17 個しかありません。キーを取得することで、これらの一意の値を使用できます。

```
unique heights << get keys;
   {51, 52, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70}
```

注：これが可能なのは、JMP では、文字列だけではなくあらゆるデータタイプが連想配列のキーとして使用できるためです。

連想配列を使えば、列内の一意の値を簡単に効率的に見つけることができます。次のスクリプトは、100,000 行もあるデータテーブルを作成するので時間がかかります。しかし 39 個の一意の値を探す作業はほとんど時間がかかりません。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
nms = dt:name << getvalues;
dtbig = New Table( "BigBigClass",
   New Column( "名前",
      character,
      setvalues( nms[J( 100000, 1, Random Integer( N Items( nms ) ) )] )
   );
);
Wait( 0 );
t1 = Tick Seconds();
Write(
   "\!N# names from BigBigClass = ",
   N Items( Associative Array( dtbig:name ) ),
   ", elpased time=",
   Tick Seconds() - t1
);
```

列の値を文字コード順に並べ替える

キーは文字コード順に並べられるため、連想配列に値を入れれば、値が文字コード順に並べ替えられます。たとえば、<<Get Keys メッセージはキー（「**名前**」列の一意の値）を昇順で戻します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
unique names = associative array(dt:名前);
unique names << get keys;
   {"ALFRED", "ALICE", "AMY", "BARBARA", "CAROL", "CHRIS", "CLAY", "DANNY", "DAVID",
      "EDWARD", "ELIZABETH", "FREDERICK", "HENRY", "JACLYN", "JAMES", "JANE",
      "JEFFREY", "JOE", "JOHN", "JUDY", "KATIE", "KIRK", "LAWRENCE", "LESLIE",
      "LEWIS", "LILLIE", "LINDA", "LOUISE", "MARION", "MARK", "MARTHA", "MARY",
      "MICHAEL", "PATTY", "PHILLIP", "ROBERT", "SUSAN", "TIM", "WILLIAM"}
```

2 つの異なるデータテーブルの列を比較する

連想配列を使用すると、1 つの列のどの値が別の列にないか（またはどの値が両方の列にあるか）を簡単に調べることができます。たとえば、国に関するデータテーブルが 2 つあり、どの国が両方のデータテーブルに記載されているかを調べたいとします。

各データテーブルの国名が含まれている列から連想配列を作成します。

```
dt1 = Open( "$SAMPLE_DATA¥BirthDeathYear.jmp" );
dt2 = Open( "$SAMPLE_DATA¥World Demographics.jmp" );
aa1 = Associative Array( dt1:国 );
aa2 = Associative Array( dt2:国 );
```

N Items() を使用して、各データテーブルに出現する国の数を調べます。

```
N Items(aa1);
   23
N Items(aa2);
   238
```

<<Intersect メッセージを使用して、共通の値を調べます。

```
aa1 = Associative Array( dt1:国 );
aa1 << Intersect( aa2 );
```

結果を表示します。

```
Show(N Items(aa1), aa1<<get keys);
   N Items(aa1) = 21;
   aa1 << get keys = {"Australia", "Austria", "Belgium", "France", "Greece",
      "Ireland", "Israel", "Italy", "Japan", "Mauritius", "Netherlands", "New
      Zealand", "Norway", "Panama", "Poland", "Portugal", "Romania", "Switzerland",
      "Tunisia", "United Kingdom", "United States"};
```

この例では、交差と呼ばれるセット処理を使用しています。連想配列を使用したセット処理で値を比較する方法については、「集合演算における連想配列」（198 ページ）でさらに例が紹介されています。

グラフ理論における連想配列

連想配列は、以下の有向グラフの例のようなグラフ理論データ構造にも使用できます。

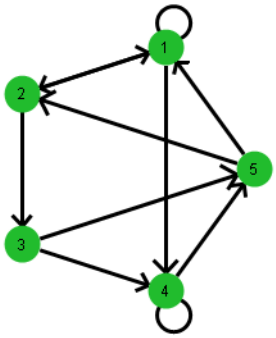
```
g = Associative Array();
g[1] = Associative Array({1, 2, 4});
g[2] = Associative Array({1, 3});
g[3] = Associative Array({4, 5});
g[4] = Associative Array({4, 5});
g[5] = Associative Array({1, 2});
```

ここでは連想配列が入れ子になっています。この連想配列 g には 5 つの連想配列（1、2、3、4、5）が含まれています。外側の配列 g では、キー（1 ～ 5）と値（マップを定義する配列）の両方が重要です。内側の連想配列では、値は関係なく、キーだけが意味を持っています。

連想配列は、図 7.2 のグラフを次のように表します。

- ノード 1 はノード 1、2、および 4 につながる
- ノード 2 はノード 1 および 3 につながる
- ノード 3 はノード 4 および 5 につながる
- ノード 4 はノード 4 および 5 につながる
- ノード 5 はノード 1 および 2 につながる

図 7.2 有向グラフの例



次に示す深さ優先探索を行う関数は、前述のような連想配列として表現された有向グラフを探索するのに使用できます。

```
dfs = Function( {ref, node, visited},
   Local( {chnode, tmp},
      Write( "Node: " || Char( node ) || ", " || Char( ref[node] << get contents )
   || "\!N" );
      visited[node] = 1;
      tmp = ref[node];
      chnode = tmp << first;
      While( !Is Missing( chnode ),
         If( !visited[chnode],
            visited = Recurse( ref, chnode, visited )
         );
         chnode = tmp << next( chnode );
      );
      visited;
   );
);
```

次の点を念頭に置いてください。

- 最初の引数はマップ構造を含んだ連想配列です。
- 2番目の引数は開始点として使用するノードです。
- 3番目の引数は、関数が、確認したノードの追跡に使用するベクトルです。

この関数の動作を確認するには、次を実行します。

```
dfs( g, 2, J( N Items( g << get keys ), 1, 0 ) );
```

出力は次のとおりです。

```
Node 2: {1, 3}
Node 1: {1, 2, 4}
Node 4: {4, 5}
Node 5: {1, 2}
Node 3: {4, 5}
[1, 1, 1, 1, 1]
```

出力の最初の5行は、ノード2から開始して、リストされた順番にその他のノードすべてに到達できることを示しています。各ノードは、そこからつながっているノード（キー）もリストします。各キーの値は1です。最後の行は、2から各ノードに到達できることを示す行列です。ノード2から到達できないノードがある場合、その値は0で表されます。

ノードのトラバーサルは次のように読みます。

1. ノード2から開始し、ノード1に行く
2. ノード1からノード4に行く
3. ノード4からノード5に行く
4. ノード5からノード2に戻り、その後、ノード3に行く

マップスクリプト

次は、図7.2のマップを生成するスクリプトです。

```
New Window( "Directed Graph",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( -1.5, 1.5 ),
      Y Scale( -1.5, 1.5 ),
      Local( {n = N Items( g ), k = 2 * Pi() / n, r, i, pt, from, to, edge, v, d},
         Fill Color( "green" );
         Pen Size( 3 );
         r = 1 / (n + 2);
         For( i = 1, i <= n, i++,
            pt = Eval List( {Cos( k * i ), Sin( k * i )} );
            edges = g[i];
            For( edge = edges << first, !Is Empty( edge ), edge = edges << Next(
   edge ),
               to = Eval List( {Cos( k * edge ), Sin( k * edge )} );
               If( i == edge,
                  Circle( Eval List( 1.2 * pt ), 0.9 * r ), // else
                  v = pt - to;
                  d = Sqrt( Sum( v * v ) );
                  {from, to} = Eval List(
                     {pt * (d - r) / d + to * r / d, pt * r / d + to * (d - r) / d}
                  );
```

```
                Arrow( from, to );
            );
        );
        Circle( pt, r, "fill" );
        Text( Center Justified, pt - {0, 0.05}, Char( i ) );
    );
  );
);
```

集合演算における連想配列

連想配列を使用して、集合演算を行うこともできます。次の例は、2つの集合の和集合、差集合および積集合を得る方法を示しています。

まず、3つのセットを作成し、ログで確認します。

```
set_y = Associative Array( {"椅子", "人", "リレー", "蛇", "三脚"} );
set_z = Associative Array( {"人", "蛇"} );
set_w = Associative Array( {"りんご", "みかん"} );
// セットをログに書き込む
Write(
   "\!N例:\!N\!tset_y = ",
   set_y << getkeys,
   "\!N\!tset_z = ",
   set_z << getkeys,
   "\!N\!tset_w = ",
   set_w << getkeys
);
   例:
      set_y = {"リレー", "椅子", "三脚", "蛇", "人"}
      set_z = {"蛇", "人"}
      set_w = {"みかん", "りんご"}
```

和集合

2つの集合の和集合を求めるには、1つの集合を他方に挿入します。

```
set_z << Insert( set_w );
Write( "\!N\!N和集合 (set_w, set_z):,\!N\!tset_z = ", set_z << getkeys );
   和集合 (set_w, set_z):,
      set_z = {"みかん", "りんご", "蛇", "人"}
```

差集合

2つの集合の差集合を求めるには、一方を他方から削除します。

```
set_y << Remove( set_z );
Write( "\!N\!N差集合 (set_z from set_y):\!N\!tset_y = ", set_y << getkeys );
```

```
差集合 (set_z from set_y):
   set_y = {"リレー", "椅子", "三脚"}
```

積集合

2つの集合の積集合を求めるには、aa << Intersectメッセージを使用します。

```
set_w << intersect( set_z );
Write( "\!N\!N積集合 (set_w, set_z):\!N\!tset_w = ", set_w << getkeys );
```

```
積集合 (set_w, set_z):
   set_w = {"みかん", "りんご"}
```

集合演算の例

名前のリストから、「Big Class.jmp」に含まれていない名前を探します。そのためには、名前を含む2つの集合から差集合を求めます。

1. 名前のリストを作成し、データテーブルを開きます。

```
names list = {"ROBERT", "JEFF", "CHRIS", "HARRY"};
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

2. 名前のリストを連想配列に入れます。

```
names = Associative Array( names list );
```

3. リストから列の値を削除することで、差集合を求めます。

```
names << Remove( Associative Array( dt:name ) );
```

4. 結果を確認します。

```
Write( "\!N{ROBERT, JEFF, CHRIS, HARRY}のうちでBig Classに含まれていないもの = ",
   names << getkeys );
```

```
"{ROBERT, JEFF, CHRIS, HARRY}のうちでBig Classに含まれていないもの =
```

第 8 章

プログラミング手法
複雑なスクリプト技術とその他の関数

この章では、例外のスローとキャッチ、スクリプトの暗号化、複雑な式の使用など、さらに高度な技術について説明します。

目次

リストと式 203
保存された式 203
マクロ 212
リストの操作 212
式の操作 215
高度な適用範囲指定と名前空間 218
Names Default To Here 219
適用範囲が指定された名前 221
名前空間 225
名前空間とスコープの参照 230
名前付き変数参照の解決 233
高度なスクリプトを作成する際のベストプラクティス 235
高度なプログラミングの概念 235
例外のスローとキャッチ 235
Function（関数） 237
Recurse（再帰） 238
Include 239
テキストファイルのロードと保存 239
BY グループを使ったスクリプト 240
ファイルをプロジェクトにまとめる 240
スクリプトの暗号化と暗号解読 243
その他の数値演算子 246
微分 246
代数的な処理 248
最大化と最小化 249
スクリプト実行のスケジューリング 250
メッセージを出力する関数 252
ログへの書き込みを行う 252
ユーザに情報を送る 253

リストと式

保存された式

式（expression）とは、評価を行うことができるものの総称です。「JSL の構成要素」の章の「名前解決のルール」（92 ページ）の最初の節では、JMP がどのように式を評価するか説明しました。ここでは、**いつ** JMP が式を評価するかについて学習します。

JMP は、できる限りすぐに式を評価し、その結果を戻します。式が割り当ての右辺にある場合、式は評価され、その結果が左辺に割り当てられます。多くの場合、このような処理で構わないでしょうが、時には評価を延期できるようにする必要が生じることもあります。

式のクォートと非クォート

いつ式を評価するかを制御する演算子は、`Expr` と `Eval` です。これらは、評価を遅延させる演算子と評価を促す演算子とみなすことができます。`Expr` は、引数を評価するのではなく、式としてそのままコピーします。`Eval` はその反対の処理をします。つまり、引数の値を求めてから、その結果を使って式全体を評価します。

`Expr` と `Eval` は、JMP に対していつ式として認識し、いつ式の評価結果を戻すかを指示する、クォート演算子と非クォート演算子とみなすことができます。

以下のすべての例では、次の 2 つが割り当てられているものとします。

```
x=1; y=20;
```

`Expr` を使って *x+y* を式として囲み、この式を `a` に割り当てた場合、*a* が評価されるときは、*x* と *y* の現在の値を使って式を計算し、結果を戻します。（例外として、`Show`、`Write`、および `Print` のユーティリティがあります。これらは、引数として名前だけを指定した場合、その名前に割り当てられている式を評価しません。）

```
a = expr(x+y);
a;
   21
```

評価された式の結果を求めるのではなく、変数に保存された式そのものが得たい場合は、`NameExpr` 関数を使います。「結果ではなく保存された式を取り出す」（205 ページ）を参照してください。

```
show(nameExpr(a));
   NameExpr(a) = x + y
```

式のクォートを入れ子にした場合、*a* が評価されるときに 1 階層だけが評価され、結果は式 `x+y` になります。

```
a = expr(expr(x+y));
show(a);
   a = Expr(x + y)
```

式の値を求めるときは、`Eval` を使ってすべての層の式を評価します。

```
show(eval(a));
   Eval(a) = 21
```

この操作は、次に示すようにどのレベルでも実行できます。

```
a=expr(expr(expr(expr(x+y))));
b=a;
   Expr(Expr(x + y))
c=eval(a);
   expr(x+y)
d=eval(eval(a));
   x+y
e=eval(eval(eval(a)));
   21
```

式を文字列としてクォート

JSL Quote() 関数は、式の内容を、引用符で囲んだ文字列として戻します。文字列内のコメントと空白はそのまま維持されます。ログウィンドウでも構文の色分けが適用されます。

次のスクリプトはその例です。

```
x = JSL Quote( /* クォートの開始 */
For (i = 1, i <= 5, i++,
   // i の値を出力
Print(i);
);
   // 式の終わり
);
Show(x);
```

出力では、JSL Quote 関数の内容が引用符で囲まれます。

```
x = " /* クォートの開始 */
For (i = 1, i <= 5, i++,
   // i の値を出力
Print(i);
);
   // 式の終わり
";
```

グローバル変数にスクリプトを格納する

Expr は、主にグローバル変数にスクリプト（マクロなど）を格納するときに使います。

```
dist = expr(Distribution(Column(:Name("身長 ( インチ )"))));
```

スクリプトを実行するときは、その名前を入力します。

```
dist;
```

ループの中に入れて、スクリプトを繰り返し実行することもできます。

```
for(i=0, i<10, i=i+1, dist);
```

Eval を使うと、式の評価を明示的に指定できます。

```
eval(dist);
```

ただし、列計算式における列に対しては、eval は思い通りに動作しないことに注意してください。たとえば、

```
Formula(log(eval(column name(i))))
```

はエラーになります。代わりに、次のように指定します。

```
Formula(Eval(Substitute( expr(log(xxx)),expr(xxx), column name(i))))
```

また、

```
Formula(eval(column name(i))+10)
```

でもエラーが生じます。計算式において列が eval されているためです。代わりに、次のように指定します。

```
Formula(Eval(Substitute(expr(xxx+10), expr(xxx), column name(i))))
```

結果ではなく保存された式を取り出す

グローバル変数を評価（実際にプラットフォームを起動）するのではなく、グローバル変数に保存されている式（上の例の dist に格納されている式 Distribution(Column(:Name("身長 (インチ)"))) など）が必要な場合は、どうするのでしょうか。そのときは、Name Expr 関数を使います。Name Expr は引数を評価しないで式として取り出します。ただし、引数がひとつの変数名だけの場合は、その変数に保存された式を、評価されていない状態で取り出します。

Expr が引数そのものを戻すのに対して、Name Expr は引数に保存されている式を検索します。Name Expr は式を得るために一番上の層だけを「開封」するのであり、すべての階層を開封するのではありません。

たとえば、変数に式を保存していて、その式を編集する必要がある場合などに、この関数を使うことになります。

```
popVar = Expr( Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row() ) );
   Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row() )

unbiasedPopVar = substitute( name expr( popVar ), expr( wild()/nrow() ), expr(
   (y[i] - Col Mean( y )) ^ 2 / ( N Row() - 1 ) ) );
   Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / (N Row() - 1) )
```

次のスクリプトを実行してから、x、Expr(x)、NameExpr(x)、および Eval(x) を比較してみます。

```
a=1; b=2; c=3;
x = Expr(a+b+c);
```

表 8.1 Eval、NameExpr、Expr の比較

コマンドと結果	説明
x; *6*	*x* を a+b+c と評価し、さらにその式を評価して結果の 6 を戻す（*外側の 2 つの層を開封する*）。

表 8.1 Eval、NameExpr、Exprの比較（続き）

コマンドと結果	説明
`Eval(x);` *6*	単に *x* を呼び出しているのと同じ。 *x* を a+b+c と評価し、さらにその式を評価して結果の 6 を戻す（*外側の2 つの*層を開封する）。
`NameExpr(x);` *a+b+c*	*x* に格納されていた式 a+b+c を戻す（*1 番外の*層を開封する)。
`Expr(x);` *x*	式 x を戻す（*1 つの*層に封入する)。

JSL には式にアクセスして式全体を調べる関数も用意されており、これらの関数はすべて、引数として名前または文字式のどちらかをとります。以下の例では、*expressionArg* は、変数の名前、または、変数と演算子などから構成される複合式のいずれかです。

NArg(*expressionArg*) は、*expressionArg* の中の引数の数を調べます。

expressionArg は、式を含む変数の名前、評価結果が式になるような式、または Expr() で囲まれた式のいずれかです。

NArg (name) は、name から式（評価されていない状態）を取り出し、引数の数を戻します。

NArg (*expression*) は、expression を評価し、引数の数を戻します。

NArg (Expr(expression)) は、式 expression の引数の数を戻します。

たとえば、aExpr = {a+b,c,d,e+f+g}; の場合

- NArg(aExpr) は 4
- NArg(Arg(aExpr,4)) は 3
- NArg(Expr({1,2,3,4})) は 4 となります。

Head(*expressionArg*) は、演算子を使わないで表した式の先頭の関数（引数なし）を戻します。式が特殊文字の二項演算子、接頭演算子、接尾演算子であるときは、それらと等価な関数が戻されます。

expressionArg は、式を含む変数の名前、評価結果が式になるような式、または Expr() で囲まれた式のいずれかです。

たとえば、aExpr = expr(a+b); の場合

- r = Head(aExpr) は Add()
- r = Head (Expr(sqrt(r))) は Sqrt()
- r = Head({1,2,3}) は {} となります。

Arg(*expressionArg*, *indexArg*) は、式の中から指定の引数を抽出し、式として戻します。

例:

Arg(*expressionArg*,i) は *expressionArg* の i 番目の引数を抽出します。

expressionArg は、式を含む変数の名前、評価結果が式になるような式、または Expr() で囲まれた式のいずれかです。

- Arg(*name*,i) は、*name* に含まれた式（評価されていない状態）を取得し、i 番目の引数を検索します。
- Arg(*expression*,i) は、*expression* を評価し、i 番目の引数を検索します。
- Arg(Expr(*expression*),i) は、expression の i 番目の引数を検索します。

別の例として aExpr = Expr(12+13*sqrt(14-15)); の場合を示します。

- Arg(aExpr,1) は 12
- Arg(aExpr,2) は 13*sqrt(14-15)
- Arg(Expr(12+13*sqrt(14-15)),2) は 13*sqrt(14-15)

式の中の引数の引数を抽出するには、Arg コマンドを入れ子にします。

- Arg(Arg(aExpr,2),1) は、aExpr の第 2 引数の中の最初の引数、つまり 13 を戻します。
- Arg(Arg(aExpr,2),2) は Sqrt(14 - 15)
- Arg(Arg(Arg(aExpr,2),2),1) は 14 - 15
- Arg(Arg(Arg(aExpr,2),2),3) は Empty()

以下に、最後の式がどのように開封されていくかを説明します。

1. 内側の Arg ステートメントが評価されます。

```
Arg(aExpr,2)
   13 * Sqrt( 14 - 15 )
```

2. 次に、内側から 2 番目の Arg が評価されます。

```
Arg(Arg(aExpr,2),2)
// これは Arg(Expr(13 * Sqrt( 14 - 15 ) ), 2) と等価
   Sqrt( 14 - 15 )
```

3. 最後に、外側の Arg が評価されます。

```
Arg(Arg(Arg(aExpr,2),2),3)
// これは Arg (Expr(Sqrt( 14 - 15 ) ), 3) と等価
   Empty()
```

Sqrt 式の要素は 1 つだけなので、第 3 引数への要求に対して Empty() が戻されます。Sqrt 式内の 2 つの引数にアクセスするには、次のようにします。

```
Arg(Arg(Arg(Arg(aExpr,2),2),1),2);
   15
```

HeadName(*expressionArg*) は、式の先頭の関数の名前を文字列として戻します。式が特殊文字の二項演算子、接頭演算子、接尾演算子であるときは、それらと等価な関数の名前が戻されます。

*expressionArg*は、式を含む変数の名前、評価結果が式になるような式、またはExpr()で囲まれた式のいずれかです。

たとえば、aExpr = expr(a+b);の場合

- r = HeadName (aExpr)は"Add"
- r = HeadName (Expr(sqrt(r)))は"Sqrt"
- r = HeadName ({1,2,3})は"List"となります。

JMPのこれまでのバージョンでは、Arg、Narg、Head、およびHeadNameがそれぞれArgExpr、NArgExpr、HeadExpr、およびHeadNameExprとして実装されていました。基本的に働きは同じでしたが、引数の評価を行わない点が異なります。旧バージョンの形式は、今後のマニュアルには記載されなくなる予定です。

文字列の部分置換

Eval Insertでは、特定の文字に囲まれた式を評価することによって、文字列の部分置換を行うことができます。[Perlでは、これは**補間**（interpolation）と呼ばれます。]

Eval Insertでは、式の前後の文字を指定すると、その間の式が評価、展開されます。

結果を返す関数と、値を置き換える関数の2通りがあります。

```
resultString = EvalInsert( 式が埋め込まれた文字列 , 開始区切り文字 , 終了区切り文字 )
EvalInsertInto( 式が埋め込まれた文字列を持つ変数 , 開始区切り文字 , 終了区切り文字 )
```

区切り文字の指定はオプションです。デフォルトの開始文字は"^"、デフォルトの終了文字は、開始文字と同じ文字です。

```
xstring = "def";
r = EvalInsert("abc^xstring^ghi");  // 戻り値が "abcdefghi"; になる

// 値を置き換え
r = "abc^xstring^ghi";
EvalInsertInto(r); // r が "abcdefghi"; になる

// 区切り文字の指定
r = EvalInsert("abc%xstring%ghi","%");  // 戻り値が "abcdefghi"; になる

// 開始値と終了値が異なる例
r = EvalInsert("abc[xstring]ghi","[","]");  // 戻り値が "abcdefghi"; になる
```

数値にロケール固有の表示形式が含まれている場合は、<<Use Locale(1)オプションを含めます。次の例は、小数点の代わりにカンマを使用するロケール環境で実行したものです。

```
EvalInsert( "^1.2^", <<Use Locale(1) );
   1,2
```

リストの中の式を評価する

Eval Listは、リスト内の式を評価し、その結果をリストにして戻します。

```
x = { 1+2, 3+4 };
y = evallist(x);   // yは{3,7}になる
```

Eval Listは、Column Dialogまたは引数Modalを使用したNew Windowから戻された、ユーザの選択によるリストをロードするのに便利です。

式の中の式を評価する

Eval Exprは、ある式の中に含まれている式を評価しますが、戻すのは評価の結果が含まれた式です。Evalはある式の中に含まれている式をすべて評価してから、その式全体を評価します。次は、内部の式を最初に評価する必要がある例です。

```
// X3という名前の列を持つデータテーブルがあるとする
x = expr( distribution(column( expr("X"||char(i)) )) );
i = 3;
y=Eval Expr(x);   // Distribution(Column("X3"))を戻す
```

ただし、Eval Exprは、内側の層における式を評価するだけで、結果は式として戻されます。結果を更に評価するには、その後のステップで結果を呼び出すか、EvalExpr()をEval()で囲む必要があります。

```
// 2つのステップを取る方法
y=Eval Expr(x);
y;

// 1つのステップで済む方法
eval(eval expr(x));
```

最初にEval Exprを実行しないで、直接xにEvalを使おうとした場合にどのようなことが起こるかについては、表8.3「計算を制御するすべての演算子の比較」（211ページ）を参照してください。

文字列を式に、式を文字列に解析する

構文解析とは、プログラム言語の式として文字列を取り込むことです。JSLの式を文字列として保持しており、その文字列から式を構築したいとします。Parse関数は式を戻します。Parse関数は式を戻すだけなので、式を評価するにはEval関数を使います。

```
x = parse("a=1") ;  // 現在xには式a=1が入っている
eval(parse("a=1")); // 現在aには値1が入っている
```

この逆をするには、式を文字列に変換するChar関数を使います。通常、Charは引数を評価してから文字列に変換するため、Char関数の引数をExpr関数（または変数のNameExpr関数）にします。

```
y = char(expr(a=1)); // yは「a=1」という文字値になる
z = char(42);
```

引数が数値の場合、Char関数では、フィールド幅と小数桁数の引数を指定できます。デフォルトでは、フィールド幅は18で、小数桁数は99です（［最適］の形式）。

```
char(42,5,2);
   // 文字値は "42,00" になる
```

数値にロケール固有の形式を維持するには、次の例のように、<<Use Locale(1)オプションを含めます。

```
char( 42,5,2, <<Use Locale(1) );
   // フランスのロケールで実行した場合、"42,00" になる
```

Charの逆は、それほど簡単ではありません。文字列を式に変換するときはParseを使いますが、文字列を数値に変換するときはNumを使います。

```
parse(y);
num(z);
```

表8.2 式の保存と計算をする関数

関数	構文	説明
Char	Char(Expr(*expression*)) Char(*name*)	*expression*（式）を文字列に変換する。式はExprで囲む必要があります。囲まない場合、その評価結果が文字列に変換されます。
	string = char(*number*, *width*, *decimal*)	*number*（数値）を文字値に変換する。*width*と*decimal*は、形式を指定するためのオプションの引数です。デフォルトでは、width（フィールド幅）は18で、decimal（小数桁数）は99です。
Eval	Eval(*x*)	*x*を評価し、次いで*x*の結果を評価する（非クォート）。
Eval Expr	Eval Expr(*x*)	*x*の中の式すべてを評価して、式を戻す。
Eval List	Eval List(*list*)	リスト（*list*）内の式を評価した後のリストを戻す。
Expr	Expr(*x*)	引数を評価しないまま戻す（式のクォート）。
NameExpr	NameExpr(*x*)	*x*に格納されている式を評価せずに戻す。NameExprは、*x*が1つの変数名の場合、名前xを評価せずに戻すのではなく、その変数*x*に保存されている式を評価せずに戻します。その点を除けば、NameExprはExprと同じです。
Num	Num("*string*")	文字列を数値に変換する。
Parse	Parse("*string*")	文字列をJSLの式に変換する。

まとめ

表8.3では、*x*を使った評価制御演算子のさまざまな使い方を比較しています。すでに、次のように*x*と*i*は割り当てられているものとします。

```
x = expr( distribution(column( expr("X"||char(i)) )) );
i = 3;
```

表8.3 計算を制御するすべての演算子の比較

コマンドと結果	説明
x; // または Eval(x); *見つかりません。'distribution' へのアクセスまたは評価 , 不正な引数 ({"X" || Char(i)}), distribution(Column(Expr("X" || Char(i))))*	Eval(x) と x の呼び出しは同じ。 式 distribution(column(expr("X" || Char(i)))) を評価します。結果はエラーとなります。列名は、Expr() 関数によって囲まれているため、"X"||Char(i) のように認識されます。
Expr(x); *x*	式 x を戻す（*追加の*層により x を包みこむ）。
Name Expr(x); *Distribution(Column(Expr("X" || Char(i))))*	*x* に保存されている式 Distribution(Column(Expr("X" || Char(i)))) をそのまま戻す。
y=Eval Expr(x); *Distribution(Column("X3"))*	内側の式を評価するが、外側の式は評価しないため、*y* は Distribution(Column("X3")) 。
y; // または Eval(Eval Expr(x)); *Distribution[]*	Eval(eval expr(x)) と y の呼び出しは同じ。 Distribution(Column("X3")) を評価し、プラットフォームを起動します。
z = Char(nameexpr(x)); *"Distribution(Column(Expr (¥!"X¥!" || Char(i))))"*	式全体を文字列としてクォートする。必要に応じて、¥!" というエスケープ文字を追加します。 Char(x) と指定すると、最初に *x* を評価しようとしてエラーが発生するため、欠測値を引用符で囲んだ "." が戻されることに注意してください。
Parse(z); *Distribution(Column(Expr("X" || Char(i))))*	文字列を解析（パース）して式を戻す。

表 8.3 計算を制御するすべての演算子の比較（続き）

コマンドと結果	説明
a = Parse(Char(NameExpr(x))); Eval(EvalExpr(a)); **Distribution[]**	とても極端な例。 この例は、少なくともこれを2つのステップに分解する必要があることに注意してください。これを1つの大きなステップに組み合わせると、Eval Expr関数が、Parse層以下の内容を評価せずにそのまま残してしまうため、別の結果となります。 Eval(EvalExpr(Parse(Char(NameExpr(x))))); *Distribution(Column(Expr("X" ||* *Char(i))))*

マクロ

保存された式（expression)は、マクロとして使用することも可能です。汎用的な処理を式として変数に保存しておくと、その変数を呼び出せばいつでも中の汎用的な処理を実行できます。この例では、Ifの引数として4つのマクロ（グローバル変数）を指定しています。この例では、Ifの引数として4つのマクロ（グローバル変数）を指定しています。

```
lastStdzdThickness=expr(
   (thickness[nrow()]-col mean(thickness)) / col std dev(thickness));
continue=expr(...<データを読み込むスクリプト>...);
log=expr(print(char( long date(today()))||" の分です。"));
limitvalue=1;

if(lastStdzdThickness<limitvalue,log;continue,break);
```

Exprを使って式を保存すると、その式はスクリプトそのもので、その時点では評価されません。式を保存した変数を実際に呼び出すまで、評価は行われません。式に含まれているすべての変数、データ、または式は、式の実行時に初めて評価されます。式を保存し、後で評価するときの詳細なルールについては、「保存された式」（203ページ）を参照してください。

リストの操作

次に挙げる演算子でリストを操作します。また、これらの演算子は、次の節、「式の操作」（215ページ）で説明するように、式の操作にも使うことができます。コマンドとその説明は、表8.4「リストと式を操作する関数」（217ページ）にまとめられています。

大部分の関数には、新しい値を作成する形と、その場で直接引数に作用する形の2種類があります。両方の例を挙げます。

```
A = Remove(A,3); // リスト A の 3 番目の項目を削除し、A に結果を保存する
Remove From(A,3); // その場でリスト A の 3 番目の項目を削除する

onetwo=Insert({1},2); // onetwo は {1,2} になる
InsertInto(A,{1,2},4); // 現在の 4 番目の項目の前に 1,2 を挿入する
```

注: Insert Into関数で挿入する場所を省略した場合、リストの最後尾に項目が挿入されます。

```
a=Shift({1,2,3,4},1); // a にリスト {2,3,4,1} を保存する
Shift Into(a,-1); // a は {1,2,3,4} になる

b=Reverse(a); // b は {4,3,2,1} になる
Reverse Into(a); // a も {4,3,2,1} になる

s=Sort List({1,4,2,5,-7.2,pi(),-11,cat, apple, cake});
   // s は並べ替えられたリストになる
c={5,pie,2,pi(),-2};
Sort List Into(c); // c は {-2,2,5,Pi(),pie} になる
```

インプレース演算子

インプレース演算子は、リストまたは式を直接操作する演算子です。この演算子は、たとえば、Remove From、Insert Intoなどのように、演算子名にFromまたはIntoが付いています。結果を戻さないため、結果を見るにはリストを表示させる必要があります。インプレース演算子の第1引数は、L-valueでなければなりません。**L-value**とは、値を代入できるグローバル変数などのエンティティです。

```
myList={a, b, c, d};
Insert Into(myList,2,3);
show(myList);
   myList:{a, b, 2, c, d}
```

以下の例では、入れ子になったリストを用いて、Insert IntoとRemove Fromの使い方を示しています。

```
a = {{1, 2, 3}, {"A", "B", "C"}};
Show( a );
   a = {{1, 2, 3}, {"A", "B", "C"}}

Insert Into( a[1], 99, 1 );
Show( a );
   a = {{99, 1, 2, 3}, {"A", "B", "C"}}

Remove From( a[1], 1 );
Show( a );
   a = {{1, 2, 3}, {"A", "B", "C"}}
```

インプレースでない演算子

インプレースでない演算子の場合には、リストを直接記述するか、評価結果がリストとなる変数の名前を指定する必要があります。このような演算子には、名前に From または Into は**付きません**。このタイプの演算子は、第1引数の元のリストや式には直接、変更を加えないで、変更した後のリストや式を**戻します**。

```
myNewList=Insert({a, b, c, d}, 2, 3);
   {a, b, 2, c, d}

oldList={a, b, c, d};
newList=Insert(oldList, 2, 3);
   {a, b, 2, c, d}
```

Substitute（置換）

Substitute と Substitute Into については、詳しく説明する必要があるでしょう。どちらも、リスト（または式）でパターンに一致するものをすべて検索し、それを別の式に置換します。パターン(pattern)は名前でなければなりません。引数は適用される前に評価されるため、ほとんどの場合、Expr 関数を使って引数をクォートする必要があります。

```
Substitute({a,b,c}, expr(a), 23); // {23,b,c} を作成する
Substitute(expr(sine(x)),expr(x),expr(y)); // sine(y) を作成する
```

引数の評価を延期するときは、Expr の代わりに Name Expr を使います。

```
a={quick,brown,fox,jumped,over,lazy,dogs};
b=Substitute(a,expr(dogs),expr(cat));
canine=expr(dogs);equine=expr(horse);
c=Substitute(a,nameexpr(canine),nameexpr(equine)); show(a,b,c);
   a = {quick,brown,fox,jumped,over,lazy,dogs}
   b = {quick,brown,fox,jumped,over,lazy,cat}
   c = {quick,brown,fox,jumped,over,lazy,horse}
```

Substitute Into は同じ操作をインプレースで実行します。

```
Substitute Into(a,expr(dogs),expr(horse));
```

引数に複数のパターンを列挙すると、一度のステップで2つ以上の置換を実行できます。

```
d=Substitute(a,
   nameexpr(quick),nameexpr(fast),
   nameexpr(brown),nameexpr(black),
   nameexpr(fox),nameexpr(wolf)
);
   {fast,black,wolf,jumped,over,lazy,dogs}
```

同じパターンが式中に複数ある場合には、すべてのパターンに対して代入が行われます。たとえば、次のような設定が行えます。

```
Substitute(expr(a+a), expr(a), expr(aaa));
```

の結果は、次のようになります。

```
aaa + aaa
```

式の操作

リストを操作する演算子は、ほとんどの式を処理することもできます。必ずExprを使って式をクォートしてください。たとえば、次のような設定が行えます。

```
Remove(Expr(A+B+C+D),2); // 式A+C+Dになる
b=Substitute(expr(log(2)^2/2), 2, 3); // 式Log(3)^3/3になる
```

リストの場合と同様、インプレース演算子の第1引数はL-valueでなければなりません。L-valueとは、値を代入できるグローバル変数などのエンティティです。インプレース演算子は、リストまたは式を直接操作する演算子です。この演算子は、たとえば、Remove From、Insert Intoなどのように、演算子名にFromまたはIntoが付いています。結果を戻さないため、結果を見るには式を表示する必要があります。

```
polynomial=expr(a*x^2 + b*x + c);
insertinto(polynomial,expr(d*x^3),1);
show(polynomial);
   polynomial = d * x ^ 3 + a * x ^ 2 + b * x + c
```

インプレース演算子でない場合には、式を直接記述するか、式を含む変数の名前をNameExprに指定する必要があります。このような演算子には、名前にFromまたはIntoは**付きません**。このタイプの演算子は、第1引数の元のリストや式には直接、変更を加えないで、変更した後のリストや式を**戻します**。

```
cubic=insert(expr(a*x^2 + b*x + c),expr(d*x^3),1);
   d * x ^ 3 + a * x ^ 2 + b * x + c

quadratic=expr(a*x^2 + b*x + c); cubic=insert(nameexpr(quadratic),expr(d*x^3),1);
   d * x ^ 3 + a * x ^ 2 + b * x + c
```

Substitute（置換）

置換はとても強力な機能です。すでに説明した「Substitute（置換）」（214ページ）を参照してください。ここでは、式で置換を行うときの注意点をいくつか説明します。

Substitute（pattern,name,replacement）は、式の名前を置き換えます。

NameExprは指定された変数に含まれている式を、評価せずに戻します。

```
a = expr(distribution(column(x), normal quantile plot)); show(NameExpr(a));
   NameExpr(a) = Distribution(Column(x), normal quantile plot)
```

Substituteでは引数がすべて評価されるため、引数を正しくクォートする必要があります。

```
b = substitute(NameExpr(a),expr(x),expr(:Name("体重(ポンド)")));
   show(NameExpr(b));
   NameExpr(b) = Distribution(Column(:Name("体重(ポンド)")), normal quantile plot)
```

SubstituteIntoの場合、第1引数はL-value（左辺に指定できるもの）です。NameExprを用いる必要はありません。

```
SubstituteInto(a,expr(x),expr(:Name("体重(ポンド)"))); show(NameExpr(a));
   NameExpr(a) = Distribution(Column(:Name("体重(ポンド)")), normal quantile plot)
```

Substituteは式の一部を変更するときに便利な関数です。次の例は、Is NumberやIs Matrix関数などの複数の関数を用いる例です。

```
data = {1, {1,2,3}, [1 2 3], "abc", x, x(y)};
ops = {is number, is list, is matrix, is string, is name, is expr};
m=J(n items(data),n items(ops),0);
test = expr(m[r,c] = _op(data[r]));
for (r=1,r<=n items(data),r++,
   for (c=1,c<=n items(ops),c++,
      eval(substitute(nameexpr(test), expr(_op),ops[c])))));
show(m);
   m =
   [1 0 0 0 0 0,
   0 1 0 0 0 1,
   0 0 1 0 0 0,
   0 0 0 1 0 0,
   0 0 0 0 1 1,
   0 0 0 0 0 1];
```

SubstituteIntoを使うと、JMPで2次方程式を求めることができます。次の例では、$4x^2 - 9 = 0$を求めています。

```
/* 次の方程式の解を求める */
/*        a*x^2 + b*x + c = 0       */
// 2 次の式は x=(-b +- sqrt(b^2 - 4ac))/2a
// リストを使って、+- 演算の + と - 両方の結果を保存する
x={expr((-b + sqrt(b^2 - 4*a*c))/(2*a)),
   expr ((-b - sqrt(b^2 - 4*a*c))/(2*a))};
// 次に、係数に挿入する
substitute into(x,expr(a),4, expr(b),0, expr(c),-9);
show(x);            // 置換結果を表示する
show(evalexpr(x)); // 解を表示する
   x = {Expr(-0+Sqrt(0^2-4*4*-9))/(2*4)),Expr((-0-Sqrt(0^2-4*4*-9))/(2*4))}
   EvalExpr(x) = {1.5,-1.5}
```

リストと式を操作する演算子については、この前の節、「リストの操作」（212ページ）で説明していますが、その概要を表8.4に示します。

表 8.4 リストと式を操作する関数

関数	構文	説明
Remove	*x* = Remove(*list*|*expr*) *x* = Remove(*list*|*expr*, *position*) *x* = Remove(*list*|*expr*, {*positions*}) *x* = Remove(*list*|*expr*, *position*, *n*)	指定の場所（*position*）の項目を削除したリスト（*list*）または式（*expr*）を戻す。場所（*position*）を省略したときは、最後の項目が削除されます。*position* をリストで指定することもできます。追加の引数 *n* を指定すると、1 つの項目ではなく *n* 個の項目が削除されます。
Remove From	Remove From(*list*|*expr*, *position*) Remove From(*list*|*expr*) Remove From(*list*|*expr*, *position*, *n*)	リストから項目を削除する。よって、戻り値を何かに割り当てることはできません。第 1 引数は L-value（左辺に指定できるもの）でなければなりません。
Insert	*x* = Insert(*list*|*expr*, *item*, *position*) *x* = Insert(*list*|*expr*, *item*)	リスト（*list*）または式（*expr*）の指定箇所 (*position*) に項目 (*item*) を挿入する。位置を指定しない場合、文字列は末尾に挿入されます。
Insert Into	Insert Into(*list*|*expr*, *item*, *position*) Insert Into(*list*|*expr*, *item*)	Insert 関数と同じだが、結果を元の変数に格納する。リスト（*list*）や式（*expr*）は L-value（左辺に指定できるもの）でなければなりません。
Shift	*x* = Shift(*list*|*expr*) *x* = Shift(*list*|*expr*, *n*)	1 つまたは *n* 個の項目を、リスト（*list*）または式（*expr*）の前から後ろへシフトする。*n* の値が負のときは、後ろから前へシフトします。
Shift Into	Shift Into(*list*|*expr*) Shift Into(*list*|*expr*, *n*)	項目をシフトして、その結果を元の変数に割り当てる。
Reverse	*x*=Reverse(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目の順序を逆にする。
Reverse Into	Reverse Into(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目の順序を逆にし、結果を元の変数に割り当てる。
Sort List	*x*=Sort List(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目を並べ替える。まず数値を昇順に並べ、その後に名前・文字列・演算子の内部コード順に並べます。たとえば、+（プラス記号）は -（マイナス記号）よりも、文字コードでは前に位置しているので、1+2 は 1-2 より小さいと判断されます。また、{1,2} は {1,3} より小さく、{1,3} は {1,3,0} より小さいと判断されます。{1000} は {"a"} より小さくなりますが、{a} と {"a"} は同じです。

表 8.4 リストと式を操作する関数（続き）

関数	構文	説明
Sort List Into	Sort List Into(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目を並べ替え、その結果を元の変数に割り当てる。
Sort Ascending	Sort Ascending(*list*|*matrix*)	リスト（*list*）または行列（*matrix*）の要素を昇順に並べたリストを戻す。
Sort Descending	Sort Descending(*list*|*matrix*)	リスト（*list*）または行列（*matrix*）の要素を降順に並べたリストを戻す。
Loc Sorted	Loc Sorted(*A*, *B*)	行列 **A** の値と行列 **B** の値が一致する位置を、添え字の行列で作成する。**A** は昇順に並べた行列でなければなりません。
Substitute	R = Substitute(*list*|*expr*, Expr(*pattern*), Expr(*replacement*), ...)	リスト（*list*）または式（*expr*）の中で、指定のパターン（pattern）に一致する部分を検索し、*replacement* で置き換える。パターン(pattern)は名前でなければなりません。第2引数と第3引数は適用される前に評価されるため、ほとんどの場合、Expr 関数を使って引数をクォートする必要があります。引数の評価を延期するときは、Expr の代わりに Name Expr を使います。複数の置換を実行する場合、1つのステートメントで pattern と replacement のペアを複数指定できます。
Substitute Into	Substitute Into(*list*|*expr*, Expr(*pattern*), Expr(*replacement*), ...)	Substitute 関数と同じだが、結果を元の変数に格納する。リスト（list）や式（expr）は L-value（左辺に指定できるもの）でなければなりません。

高度な適用範囲指定と名前空間

プロダクション環境で使用されるスクリプトには、スクリプト間の競合を回避する目的で、より高度な適用範囲の指定技術を使用する必要があります。JMP にはより高度な技術が3つ用意されています。

- Names Default To Here() 関数。簡単なスクリプトを作成するだけなら、このコマンドで十分です。「Names Default To Here」（219ページ）を参照してください。
- JMP で定義済みの適用範囲。「適用範囲が指定された名前」（221ページ）を参照してください。
- 独自のスクリプトに作成できる名前空間。「名前空間」（225ページ）を参照してください。

Names Default To Here

プロダクションで使用するスクリプトを作成する場合、そのスクリプトを現在のユーザ環境から分離する必要があります。そうでなければ、スクリプト内で使用している変数が、ユーザや別のスクリプトによって使用される他の変数の影響を受ける可能性があります。分離するためには、名前をローカル環境で使用します。それには、次のステートメントを使って実行モードを設定します。

```
Names Default To Here(1);
```

Names Default To Here モードがオンになっているスクリプトの中の非修飾の名前は、そのスクリプトだけに関連付けられます。ただし、名前はスクリプトが存続する限り、または、そのスクリプトによって作成されたオブジェクトやそのスクリプトを保持しているオブジェクトがアクティブである限り、存続します。特別な理由がない限り、プロダクション環境で使用するすべてのスクリプトは、`Names Default To Here(1)` で開始することをお勧めします。スクリプトがこのモードで非修飾の名前を使用する場合、名前はローカルの名前空間内で解決されます。

グローバル変数を参照するには、名前の適用範囲を明確にグローバル変数として指定します（たとえば `::global_name`）。データテーブル内の列を参照するには、名前の適用範囲を明確にデータテーブル列として指定します（たとえば `:column_name`）。

注： `Names Default To Here(1)` は、特定のスクリプトのモードを定義します。グローバルな定義ではありません。あるスクリプトでこのモードを有効にし、別のスクリプトでは無効にすることができます。デフォルトではオフに設定されています。

JMP 8以前のバージョンでは、スクリプト同士を分離する唯一の方法が、他のスクリプトで使用されていないような長い名前を使用することでした。`Names Default To Here(1)` を使用すると、この方法が必要でなくなります。

`Local()` は、スクリプト内の特定のコンテキストにだけローカルな適用範囲を作成し、相互に作用する関数のある長いスクリプトを含めることはできません。一方、`Names Default To Here(1)` は、スクリプト全体に対してローカルな適用範囲を作成できます。

簡単なスクリプトを作成するだけなら、`Names Default To Here(1)` で十分です。

非修飾の名前付き変数参照の操作

`Names Default To Here()` 関数は、**非修飾**の名前付き変数参照の解決方法を決定します。`here:var_name` を使って明示的に変数のスコープを指定すれば、`Names Default To Here()` のオン／オフに関わらず常に適用範囲で動作します。`here` およびその他の適用範囲については、「適用範囲が指定された名前」（221 ページ）を参照してください。

Names Default To Here モードを有効にすると、`Here` というスコープが実行スクリプトに関連付けられます。`Here` スコープには、作成された非修飾の名前付き変数のうち、割り当ての対象（L-value）であるものすべてが含まれます。JMP 8以前のバージョンでは、これらの変数は通常、グローバルスコープに置かれていました。`Here` スコープを使うと、複数の実行スクリプト内の変数がお互いに分離され、名前の競合が回避されるので、変数名の管理やスクリプト作成が簡単になります。`Global` スコープを使えば名前を共有できます。

Names Default To Here とグローバル関数

このスクリプト例を一度に 1 行ずつ実行し、Names Default To Here() 関数が変数名の解決にどのような変化をもたらすかを見てみましょう。

スクリプト例

```
a=1;
Names Default To Here(1);
a=5;
show(global:a, a, here:a);
   global:a = 1;
   a = 5;
   here:a = 5;
```

1. 1 行目を実行し、名前が **a**、値が 1 のグローバル変数を作成します。
2. 2 行目を実行し、**Names Default To Here** モードをオンにします。
3. 3 行目を実行し、名前が **a**、値が 5 のローカル空間を作成します。この行は、グローバル変数 **a** に割り当てられた値を**変更しません**。
4. 4 行目を実行し、範囲指定された変数と範囲指定されていない変数がそれぞれどのように解決されるのかを見ます。

非修飾の **a** は here:a と認識されます。Names Default To Here() がオンでない場合、**a** は **a** という名前のグローバル変数と認識されます。

ただし、Show() 関数内で global:a の代わりに ::a を使用した場合、出力は若干異なります。

```
show(::a, a, here:a);
   a = 1;
   a = 5;
   here:a = 5;
```

Names Default To Here() 関数の使用例

ここでは、次のような定義の 2 つのスクリプトがあり、どちらも Names Default To Here() が**オフ**（デフォルト設定）になっています。

注：この例では、2 つのスクリプトのスクリプトウィンドウが別々でなければなりません。

```
// スクリプト 1
a = 1;
show(a);

// スクリプト 2
a = 3;
show(a);
```

1. スクリプト 1 を実行します。結果は次のとおりです。

   a = 1

2. スクリプト 2 を実行します。結果は次のとおりです。

   a = 3

3. スクリプト 1 の `show(a);` 行のみを実行します。結果は次のとおりです。

   a = 3

変数 **a** はグローバルで、スクリプト 2 によって最後に変更されたため、ログには `a = 3` と表示されます。これは、JMP 9 以降ではデフォルトの動作ですが、JMP 8 以前のバージョンでは唯一可能な動作でした。

4. では、**両**スクリプトとも、`Names Default To Here()` をオンにしてみましょう。

   ```
   Names Default To Here(1);
   ```

注：`Names Default To Here()` は、特定のスクリプトに対してローカルです。グローバル設定では**ありません**。

---

5. スクリプト 1 を実行します。結果は次のとおりです。

   a = 1

6. スクリプト 2 を実行します。結果は次のとおりです。

   a = 3

7. スクリプト 1 の `show(a);` 行のみを実行します。結果は次のとおりです。

   a = 1

変数 **a** のコピーが各スクリプトに保持されるため、ログには `a = 1` と表示されます。

注：この関数を使用する際の問題は、通常、修飾名と非修飾名のグローバル変数への参照が混合することによって起こります。名前の適用範囲を常に明示的に指定することによって、意図しない変数への参照を防ぐことができます。

適用範囲が指定された名前

`scope:name` という形式のスコープを使用することにより、名前を解決する場所を指定できます。ここで、`scope` は名前の解決方法を示します。たとえば、`here:name` は、名前がローカルで解決されるべきであることを意味します。**Names Default To Here** モードを使用すると、`here:name` は `name` と等価です。スコープは名前の参照方法を指示します。

構文では、コロンを添えたスコープ演算子を使用します。

```
scope:name
```

スコープには、次のいくつかの種類があります。

- 解決ルールを示すもの。たとえば、here:x は、x がスクリプトにローカルな名前として解決されるべきであることを意味します。Global:xは、xがグローバル名として解決されるべきであることを意味します。
- 名前空間の参照変数。たとえば、ref:aは、aが、refが参照する名前空間で解決されるべきであることを意味します。
- 名前を列名として参照するデータテーブル参照。たとえば、dt:height は、height が、dt が参照するデータテーブルの列として解決されるべきであることを意味します。
- 独自に作成した名前空間の名前。たとえば、myNamespace:b。ここで、myNamespace は作成した名前空間。"myNamespace":bも等価です。「名前空間」（225ページ）を参照してください。

列計算式の適用範囲指定の例

以下の例は、計算式を含んだ列の適用範囲の指定方法を示しています。スクリプトの中で、*x*はグローバル変数、ローカル変数、列名として使用されています。

最初のスクリプトでは、列名 x は適用範囲が指定されていません。2 番目の列内の計算式は、列**x**の値を 100 倍するものです。この場合、列の値は 100、200、300 という結果になります。

```
::x=5;
New Table( "テスト",
   New Column( "x", Values( [1,2,3] ) ),
   New Column( "y", Formula( 100*x ) ),
);
```

次のスクリプトでは、列**y**の計算式はローカル変数**x**に 500 を割り当てた後、**x**に 50 を足します。列の各セルの値は 550 になります。

```
::x=5;
New Table( "テスト",
   New Column( "x", Values( [1,2,3] ) ),
   New Column( "y", Formula(Local( {x=500}, x+50 ) ) ),
);
```

定義済みスコープ

JMP には、削除や置換ができない定義済みスコープが用意されています。各スコープには、関連するオブジェクトに応じて、特定のルールがあります。

表 8.5 定義済みスコープ

スコープ	説明
Global	グローバル名は JMP 環境全体で共有される。
Here	実行スクリプトを範囲としたスコープ。

表 8.5 定義済みスコープ（続き）

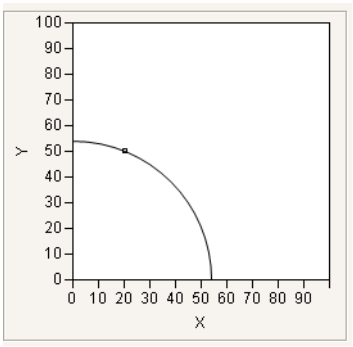
スコープ	説明
Builtin	JMP ビルトイン関数。たとえば、Builtin:Sqrt()。名前は、JMP 環境全体で共有されます。 JSL 関数をカスタム関数で上書きした場合でも、このスコープを使ってビルトインの JSL 関数にアクセスできます。
Local	ローカルスコープに似たスコープ。Function()、Local()、Parameter() の関数内で使用できます。
Local Here	Names Default to Here(1) 内に名前空間ブロックを提供する。ローカルブロックには寿命があるので Local({Default Local},) は常に機能するとは限りませんが、Local Here() は呼び出し全体にわたって維持されます。
Window	含まれているユーザ定義ウィンドウのスコープ。（まれ）
Platform	現在のプラットフォームを範囲としたスコープ。（まれ）
Box	ContextBox() を範囲としたスコープ。ContextBox はユーザ定義ウィンドウ内で使用されます。（まれ）

Window スコープの使用例

この例は、Window スコープを使って、実行中に情報を渡します。変数 x および y の適用範囲をウィンドウに指定することで、x と y はデータテーブルなどの他のコンテキストに適用されなくなります。変数 x および y は、指定の Window 環境内だけで作成および使用されます。Window スコープは、Local() を使用するのと似ていますが、それより便利です。なぜなら、Local() は使用できる範囲が限られているからです。

```
New Window( "例",
   window:gx = 20;
   window:gy = 50;
   Graph Box(
      Frame Size( 200, 200 ),
      Handle(
         window:gx,
         window:gy,
         Function( {x, y},
            window:gx = x;
            window:gy = y;
         );
      );
      Circle( {0, 0}, Sqrt( window:gx * window:gx + window:gy * window:gy ) );
   );
);
```

図 8.1 現在のウィンドウ名前空間の例



Here スコープの使用例

この例は、Here スコープを使い、同じスクリプトによって作成されたウィンドウ間で情報を渡します。Here: を使って変数の適用範囲を指定する際、Names Default To Here() がオンである必要はありません。Here: スコープは常に使用できます。

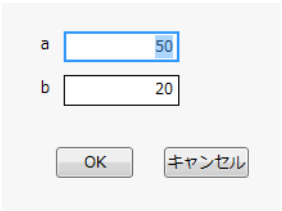
次のスクリプトは、2つのウィンドウを作成し、2つの異なるスコープを使用します。

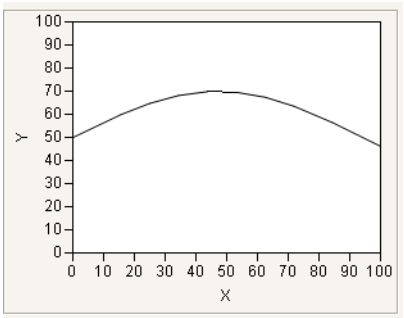
「起動」ウィンドウがユーザに2つの値を入力するよう求めます。その2つの値が「結果」ウィンドウに渡され、それらの値を使って関数がグラフ化されます。「起動」ウィンドウは、aBox と bBox の適用範囲をそのウィンドウに限定します。基本的に、それらの変数（Number Edit Boxes への参照）は「起動」ウィンドウ内にのみ存在し、「結果」ウィンドウでは使用できません。その後、2つのボックスの値が Here に範囲指定された変数内にコピーされ、このスクリプトによって作成された両方のウィンドウで使用可能となります。

```
launchWin = New Window( "起動",
   <<Modal,
   V List Box(
      Lineup Box(
         N Col( 2 ),
         Spacing( 10 ),
         Text Box( "a" ),
         window:aBox = Number Edit Box( 50 ),
         Text Box( "b" ),
         window:bBox = Number Edit Box( 20 ),

      ),
      Lineup Box(
         N Col( 2 ),
         Spacing( 20 ),
         Button Box( "OK",
            // ウィンドウを閉じる前に値をコピーする
            here:a = window:aBox << Get;
            here:b = window:bBox << Get;
```

```
            ),
            Button Box( "キャンセル",
               Throw( 1 );
            );
         ),

      );
   );

   New Window( "結果",
      Graph Box( Y Function( here:a + here:b * Sin( x / 30 ), x ) )
   );
```

図 8.2 起動と結果





名前空間

名前空間とは、一意の名前およびそれに対応する値の集まりです。名前空間への参照を変数に格納できます。JMP には名前空間マップが 1 つしかないので、名前空間の名前はグローバルです。名前空間参照は、オブジェクトを参照する他の変数と同様の変数です。それぞれの適用範囲または名前空間の中で一意でなければなりません。名前空間のメンバーは、: スコープ演算子を使って参照されます。たとえば my_namespace:x は、my_namespace という名前空間の中の x という名前のオブジェクトを参照します。独自の名前空間の作成方法および管理方法については、「ユーザ定義による名前空間の関数」（225 ページ）を参照してください。名前空間は、異なるスクリプト間での名前の競合を回避するのに特に便利です。

ユーザ定義による名前空間の関数

独自の名前空間を作成し、関連する変数および関数の定義セットを入れることができます。名前空間の管理に使用できる関数には、次のものがあります。

New Namespace

```
nsref = New Namespace( <"nsname">, <{ name = expr, ... }> );
```

これは、nsname という新しい名前空間を作成し、その名前空間への参照を戻します。引数はすべてオプションです。

nsname は、名前空間名の内部グローバルリストに格納される名前空間の名前です。nsname は、スコープ変数の接頭辞としても使用できます。この関数は名前空間への参照を戻します。また、スコープ変数参照の接頭辞としても使用できます。nsname を指定しなかった場合、名前空間が匿名となり、JMP によって作成された一意の名前が付けられます。Show Namespace() は、すべての名前空間とそれらの名前（指定名／匿名に関わらず）を表示します。

重要: nsname で指定した名前の名前空間がすでに存在する場合は、それが上書きされます。つまり、スクリプトの開発時に、名前空間をクリアまたは削除しなくても、スクリプトを変更して再実行できます。間違って上書きしてしまうのを避けるには、匿名の名前空間を使用するか、または、次のようにして、特定の名前空間がすでに存在しているかどうかを確認します。

```
If( !Namespace Exists( "nsname" ), New Namespace( "nsname" ) );
```

名前空間を作成する際の、名前付き式のリストはオプションです。name は、名前空間内でのみ存在する JMP 変数です。

注：名前付き式は、カンマで区切ったリストで指定する必要があります。セミコロンで区切ったリストは無視されます。

複数のユーザ定義の名前空間が使用される場合の競合を避けるため、名前空間には一意の名前を指定する必要があります。名前空間を匿名にすると、競合が回避されます。

名前空間

```
nsref = Namespace( "nsname" | nsref);
```

名前空間参照を戻します。引数は次のいずれかを取ります。

- 名前空間を含んだ引用符付き文字列
- 名前空間への参照

nsname がデータテーブルの場合、データテーブル列の名前空間が戻されます。

注：Namespace() はすでに存在する名前空間への参照を戻します。新しい名前空間は作成しません。

Is Namespace

```
b = Is Namespace( nsref );
```

nsref が名前空間の場合は 1（真）、そうでない場合は 0（偽）を戻します。

As Scoped

```
b = As Scoped( "nsname", var_name);
nsname:var_name;
```

As Scoped() はスコープ参照の関数形です。この関数は、指定の適用範囲にある指定の変数への参照を戻します。

Namespace Exists

```
b = Namespace Exists( "nsname" );
```

nsname がグローバル名前空間のリスト内に存在する場合は 1（真）、そうでない場合は 0（偽）を戻します。

Show Namespaces

```
Show Namespaces();
```

グローバル名前空間のリストに含まれるすべての名前空間の中身を表示します。名前空間は、New Namespace 関数または Namespace 関数のどちらかを使って参照されるまで表示されません。

名前空間へのメッセージ

名前空間は、管理のための関数だけでなく、その内容にアクセスして操作するための一連のメッセージにも対応します。

これらのメッセージは、他のすべてのメッセージと同様に、スクリプト可能なオブジェクトに送る必要があります。名前空間名は、定義済みのスクリプト可能なオブジェクトではないため、Send 操作で使用できません。ただし、名前空間の参照として使用することができます。たとえば、nsmane::var は nsref::var と等価です。

表 8.6 に、ユーザ定義の名前空間参照に対応するメッセージの定義を示します。

表 8.6 名前空間へのメッセージ

名前空間へのメッセージ	説明
`ns << Contains( "var_name" );`	*var_name* が名前空間内に存在すれば 1、そうでなければ 0 を戻す。
`ns << Delete;`	内部のグローバルリストからこの名前空間を削除する。 名前空間内の変数を削除するには、<<Remove を使用します。この表の <<Remove の項目を参照してください。
`ns << First;`	名前空間内で使用されている最初の変数名を引用符付き文字列として戻す。
`ns << Get Contents;`	名前と値のペアのリストを、それぞれ含んだリストとして戻す。名前は変数名を含んだ引用符付き文字列で、各値は変数に含まれている式（未評価）です。
`ns << Get Keys;`	名前空間内の変数名のリストを戻す。
`ns << Get Name;`	名前空間の名前を戻す。

表 8.6 名前空間へのメッセージ（続き）

名前空間へのメッセージ	説明
`ns << Get Value( "var_name" );`	名前空間内の *var_name* に含まれている式を、評価せずに戻す。
`ns << Get Values;`	名前空間内の各変数に含まれている式を、評価せずにリストとして戻す。
`ns << Get Values( { "var_name1", "var_name2", ... } );`	リスト引数で指定された、名前空間内の各変数に含まれている式を、評価せずにリストとして戻す。指定の変数名が見つからない場合、エラーを戻します。
`ns << Insert( "var_name", expr );`	指定の式（*expr*）を含む指定の名前の変数（*var_name*）を、名前空間に挿入する。
`ns << Lock;` `ns << Unlock;`	名前空間内のすべての変数をロックし、変数が追加または削除されるのを防ぐ。`<<Unlock` は、名前空間の変数すべてのロックを解除します。
`ns << Lock( <"var_name", ...> );` `ns << Unlock( <"var_name", ...> );`	名前空間内で指定の変数をロックする。変数を指定しなかった場合、すべての変数がロックまたはロック解除されます。
`n = ns << N Items;`	名前空間に含まれている変数の数を戻す。
`next k = ns << Next( "var_name" );`	指定に変数に続く変数の名前を戻す。
`ns << Remove( "var_name", ...);`	指定の変数または変数のリストを削除する。

名前空間参照の使用

次に示すものはすべて、`nsref` という参照を持つ `nsname` という名前の名前空間にある、`b` という名前の変数への参照です。

```
nsref:b
nsname:b
"nsname":b
nsref["b"]
nsref<<Get Value("b") // 右辺値として使用される
```

名前空間とインクルードされたスクリプト

インクルードされたスクリプトは、親スクリプトの名前空間内で実行されます。インクルードされたスクリプトに独自の名前空間が定義されている場合、次のいずれかを行う必要があります。

- 名前の競合を回避するため、名前空間の名前を管理する
- `New Namespace` 関数によって作成される匿名の名前を使用する

どちらの場合も、名前空間への変数参照を管理する必要があります。

Include関数には別のオプション（New Context）があります。これは名前空間を作成し、インクルードされたスクリプトをその中で実行します。この名前空間は匿名の名前空間で、親スクリプトの名前空間から独立しています。例:

```
Include("file.jsl", <<New Context);
```

この匿名の名前空間は、Hereを使って参照できます。

Include関数の詳細については、「Include」（239ページ）を参照してください。

ユーザ定義の名前空間の例

式を使った基本的な名前空間の作成および使用

ここでは、匿名の名前空間を作成し、その中で関数および変数を使用する方法を例示します。

```
new_emp = New Namespace(
   {name_string = "Hello, *NAME*!",

   print_greeting = Function( {a},
      Print( Substitute( new_emp:name_string, "*NAME*", Char( a ) ) )
   )}
);
```

名前空間内に定義する変数には、必ず完全修飾名を使用します。

```
new_emp:print_greeting( 6 );
   "Hello, 6!"
```

複素数の演算

この例では、まず複素数を表す2元リストをサポートするような関数を含む名前空間を作成し、次にその名前空間をロックします。

```
If( !Namespace Exists( "complex" ),
   New Namespace( "complex" );

   complex:make = Function( {a, b}, Eval List( {a, b} ) );
   complex:add = Function( {a, b}, a + b );
   complex:subtract = Function( {a, b}, a - b );
   complex:multiply = Function( {a, b}, Eval List( {a[1] :* b[1] - a[2] :* b[2],
   a[1] :* b[2] + a[2] :* b[1]} ) );
   complex:divide = Function( {a, b},
      d = b[1] ^ 2 + b[2] ^ 2;
      Eval List( {a[1] :* b[1] - a[2] :* b[2] / d, a[2] :* b[1] - a[1] :* b[2] / d}
   );
   );
   complex:char = Function( {a}, Char( a[1] ) || "+" || Char( a[2] ) || "i" );
```

```
);
Namespace( "complex" ) << Lock;
```

次に、このユーザ定義の名前空間に含まれる関数の使用例を示します。

```
c1 = complex:make( 3, 4 );
   {3, 4}

c2 = complex:make( 5, 6 );
   {5, 6}

cm1 = complex:make( [1, 2, 3], [4, 5, 6] );
   {[1, 2, 3], [4, 5, 6]}

cadd = complex:Add( c1, c2 );
   {8, 10}

csum = complex:Subtract( c1, c2 );
   {-2, -2}

cmul = complex:Multiply( c1, c2 );
   {-9, 38}

cdiv = complex:Divide( c1, c2 );
   {14.6065573770492, 19.7049180327869}

show( complex:char( c1 ) );
   complex:char(c1) = "3+4i";
```

名前空間とスコープの参照

名前付き変数参照の解決方法を左右する要因は数多くあります。表8.7に、名前付き変数参照が特定の状況でどのように解決されるかを示します。

表 8.7 名前空間参照[a]

形式	参照タイプ	参照ルール	作成ルール
a	非修飾	**Names Default To Here** モードがオンの場合、次の場所で変数を探します。 • Local 名前空間[b] • Here 名前空間 • 現在のデータテーブル **Names Default To Here** モードがオフの場合、次の場所で変数を探します。 • Local 名前空間[b] • Here 名前空間 • Global 名前空間 • 現在のデータテーブル	• **Names Default To Here** モードがオンの場合、Local 名前空間内[b]または Here 名前空間内に変数を作成する。 • **Names Default To Here** モードがオフの場合、Local 名前空間内[b]または Global 名前空間内に変数を作成する。
:a	現在のデータテーブル	現在のデータテーブル内で変数を探す。	（該当なし）
::a Global:a	グローバル	Global 名前空間内で変数を探す。	Global 名前空間内に変数を作成する。
ns:a dt:a Here:a "name":a expr:a	修飾	指定された名前空間内で変数を探す。変数が見つからない場合、エラーが戻されます。	指定された名前空間内に変数を作成する。ぞれ以前の値は上書きされます。
ns["a"] ns[expr]	添え字	指定された名前空間内で変数を探す。変数が見つからない場合、エラーが戻されます。	指定された名前空間内に変数を作成する。ぞれ以前の値は上書きされます。
Platform:a	修飾	プラットフォームに含まれている変数を探す。	プラットフォーム内に変数を作成する。
Local:a	修飾	ローカル関数内において、該当する変数を探す。なお、他の関数を呼び出している場合、呼び出しに使われている引数も、範囲に含みます。「Local:a の例」（233 ページ）を参照してください。	変数が指定されたローカル関数内に、変数を作成する。なお、他の関数を呼び出している場合、呼び出しに使われている引数も、範囲に含みます。

表 8.7 名前空間参照[a] （続き）

形式	参照タイプ	参照ルール	作成ルール
`Window:a`	修飾	`New Window` の window 名前空間内で、変数を探す。	`New Window` の window 名前空間内に変数を作成する。
`Box:a`	修飾	`New Window` 内に指定された `Context Box` の中で、変数を探す。	`New Window` 内に指定された `Context Box` の中に変数を作成する。

a. これらの形式は JMP 8 で使用されていました。a、:a、::a は、**Names Default To Here** モードをオフにした場合、JMP 9 以降でも同義です。

b. 現在の実行ポイントがユーザ定義の関数内、または Local 関数や Parameter 関数の本文にある場合、Local 名前空間が使用される。

Local:a の例

サンプルスクリプト	ログ出力
Delete Symbols(); Local({d1l1 = 12}, local:f1f1 = Function({fa1, fa2}, {fl1 = 99}, local:fa12 = fa1 + fa2; Local({d2l1 = 56}, local:l2l2 = 78; Show(fa12); Show(fl1); Try(Show(d1l1), Write("\!n\!n***Error=" || Char(exception_msg) || "\!n")); Show Symbols();); local:fa12;); f1f1(2, 3););	*fa12 = 5;* *fl1 = 99;* ****Error={"Name Unresolved: d1l1"(1,* *"d1l1", d1l1)}* *// Local* *d2l1 = 56;* *l2l2 = 78;* *// 2 Local* *// Local* *fa1 = 2;* *fa12 = 5;* *fa2 = 3;* *fl1 = 99;* *// 4 Local* *// Local* *d1l1 = 12;* *// 1 Local* *// Global* *exception_msg = {"Name Unresolved:* *d1l1"(1, "d1l1", d1l1)};* *// 1 Global* *5*

名前付き変数参照の解決

JMP スクリプト内で変数が参照されると、JMP は指定のルールセットを使用して変数の格納場所を解決します。変数が修飾名で参照された場合、その修飾の設定に基づいて解決されます。変数が非修飾名で参照された場合は、解決方法が少し複雑です。JMP は、実行スクリプトを使用して、実行ポイントを表す適用範囲の階層内を探します。ここでは、名前付き変数参照の解決に使用されるルールについて説明します。

JMP 9以降でも、変数名のデフォルトの解決方法がJMP 8以前と同様なので、既存のJSLスクリプトは同様に実行されます。JMP 9以降では、修飾のある名前参照とそうでない名前参照の違いを理解することが重要です。

修飾のある名前参照

修飾のある名前参照は、`:`演算子および`::`演算子を使って、参照する変数の場所やその作成場所に関する特定の情報を提供します。次に示すのは、修飾のある名前参照の例です。

```
:var
::globalvar
datatable:var
nsref:var
"nsname":var
```

修飾のない名前参照

修飾のない名前参照では、変数の場所や作成場所が明示的に特定されません。参照にはスコープ演算子（`:`または`::`）の指定がありません。修飾のない名前参照を解決する際のJMPの動作を変更するには、`Names Default To Here(1)`関数を使用します。変数名の解決に関する詳細は、「JSLの構成要素」の章の「名前解決のルール」（92ページ）を参照してください。

変数参照の解決のルール

JMPでは、変数参照の解決に、次のようなルールを（記載の順序で）使用します。

1. 変数の後ろに1組の丸括弧`( )`が付いている場合、関数とみなして探します。
2. 変数の接頭部に`:`スコープ演算子が付いている場合、または明示的なデータテーブルへの参照がある場合、データテーブル列またはテーブル変数とみなして探します。
3. 変数の接頭部に`::`スコープ演算子が付いている場合、グローバル変数とみなして探します。
4. 変数が明示的なスコープ参照（`group:vowel`など）の場合は、ユーザ定義の`group`名前空間内を探します。
5. 変数が`Local`関数または`Parameter`関数の中にある場合は、ローカル変数とみなして探します。変数に添え字がある場合（変数が入れ子になっている場合）には、展開できなくなるまで、最後まで展開します。
6. 変数がユーザ定義の関数の中にある場合は、関数の引数またはローカル変数とみなして探します。
7. 現在のスコープおよびその親スコープ内を探します。`Here`スコープに到達するまで繰り返します。
8. `Here`スコープ内の変数とみなして探します。
9. グローバル変数とみなして探します。
10. スクリプトの冒頭に`Names Default to Here(1)`がある場合は、探すのを中止します。そのスコープはローカルです。
11. データテーブル列またはテーブル変数とみなして探します。
12. 演算子またはプラットフォーム起動名（一変量の分布、二変量の関係、チャートなど）とみなして探します。

13. 名前が見つからなかった場合
    - 名前が参照されている場合、ログにエラーを出力します。
    - 名前が割り当ての対象（L-value）として使われている場合は、次を確認します。

      変数の接頭部に :: スコープ演算子が付いている場合、グローバル変数を作成して使用します。

      変数が明示的なスコープ参照の場合、指定された名前空間またはスコープの中に変数を作成して使用します。

      スクリプトの冒頭に Names Default to Here(0) がある場合は、グローバル変数を作成します。

      スクリプトの冒頭に Names Default to Here(1) がある場合は、Here 名前空間変数を作成します。

高度なスクリプトを作成する際のベストプラクティス

グローバル名前空間の汚染を最小化し、スクリプトの相互作用を回避する

スクリプトを常に次の行で開始します。

```
Names Default To Here(1);
```

スクリプト間で変数を共有する

名前付き名前空間を使用します。名前空間の名前はグローバルスコープに置かれます。

匿名の名前空間を使用する

匿名の名前空間への名前空間参照を使用すると、他の名前空間との競合を回避できます。

高度なプログラミングの概念

ここでは、複雑なスクリプトの作成に役立つ、より高度なプログラミング技術について説明します。

- 「例外のスローとキャッチ」（235 ページ）
- 「Function（関数）」（237 ページ）
- 「Recurse（再帰）」（238 ページ）
- 「Include」（239 ページ）
- 「テキストファイルのロードと保存」（239 ページ）

例外のスローとキャッチ

Throw( ) 関数を実行することにより、そのスクリプト自体を停止できます。スクリプトがエラー状態に入ってしまったときにその部分から抜け出すためには、Try( ) 関数でそのスクリプトを囲んでおきます。

Try 関数は、2つの式を引数としてとります。Try 関数はまず1つ目の式を評価します。1つ目の式がエラーや Throw() を戻したときは、次を実行します。

1. 評価をそこですぐに停止する。
2. 何も戻さない。
3. 2番目の式を評価する。

Throw には引数は必要ありませんが、オプションで文字列を引数として指定することができます。引数を指定した場合、Throw が実行されると、*exception_msg* という名前のグローバル変数にその文字列が格納されます。これについては、次の最初の例で示します。

例

たとえば、Try と Throw を使うと、For ループの入れ子の中から抜け出ることができます。

```
a = [1 2 3 , 4 5 ., 7 8 9];
b = a;
nr = nrow(a);
nc = ncol(a);
// a[2,3]=2; // "Missing b" という結果を表示させる場合は、この行のコメントを外す

try(
   sum = 0;
   for(i=1,i<=nr,i++,
      for(j=1,j<=nc,j++,
         za = a[i,j]; if(isMissing(za),throw("Missing a"));
         zb = b[j,i]; if(isMissing(zb),throw("Missing b"));
         sum += za*zb;
      );
   ),
   show(i,j,exception_msg); throw();
);
   i = 2;
   j = 3;
   exception_msg = "Missing a";
```

Try と Throw を使って、JMP そのものが出す例外を受け取ってスクリプトを停止させることもできます。

```
try(
   dt=open("My dataset.jmp"); // 開くことができないファイル
   summarize( a =by(:年齢),c=count,meanHt=mean(:Name("身長(インチ)")));
   show(a,c,meanHt),
   print("このスクリプトは、データセットがないと動作しません"); throw();
);
```

Throwを使うために、必ずしもTryを使う必要はありません。次の例では、TryによってThrowを受け取るのではなく、自らスクリプトを停止させます。

```
dt=new table(); // 空のデータテーブルを作成
if (nrow(CurrentDataTable())==0, throw("! 空のデータテーブル "));
```

Function（関数）

JSLでは、ローカル変数を引数とした関数を作成することもできます。関数は、Function関数によって作成されます。関数の作成は、マクロの概念を拡張したものと言えます。例として、平方根を求める関数を作成します。その関数では引数が負の場合はエラーではなくゼロを戻すようにします。まず、中括弧{}を使ってローカル引数のリストを指定してから、式を直接入力します。Functionで指定した式はその場では評価されず、式として保存されるため、Exprで式を囲む必要はありません。

```
myRoot = function({x},if(x>0,sqrt(x),0));
a = myRoot(4);  // a は 2 となる
b = myRoot(-1); // b は 0 となる
```

関数は値と同じように、変数に保存されます。このため、Rootという関数とRootという変数の両方を持つことはできません。また、関数それ自体の内部にいるときを除き、いつでも関数を再定義できることも意味しています。

関数は呼び出されると、引数を評価し、第1引数のリストで指定されているローカル変数にその引数を与えます。次に関数の本体、つまり第2引数を評価します。

引数の値は、関数の一時使用のためのものです。関数が終了すると、値は捨てられます。戻される唯一の値は、戻り値です。複数の値を戻したい場合は、1つの値ではなく、リストにして戻すようにしてください。

関数を定義した場合、関数内部の式そのものにアクセスすることはできません。これはName Exprコマンドを使用しても同様です。関数内部の式にアクセスするには、その関数全体をexpr()節の中に入れる必要があります。例:

```
makeFunction = expr(myRoot=function({x}, if (x>0, sqrt(x), 0)));
d=substitute(
   NameExpr(MakeFunction),
   expr(x), expr(y)
);
show(d);
makeFunction;
```

ローカルシンボル

変数が関数のローカルであることを宣言し、グローバルシンボルスペースに影響されないようにすることができます。これは特に再帰関数で便利です。再帰関数では、関数呼び出しの各レベルでローカル変数の値を分けておく必要があるためです。

前述したように、関数を定義するには次のように指定します。

```
functionName=Function({arg1, ...}, body);
```

次のように指定すると、適用範囲が明示されていない名前すべてをローカル変数として扱うようになります。

```
functionName=Function({arg1, ...}, {Default Local}, body);
```

Default Localは、以下のような名前をローカルにします。

- グローバルとして適用範囲が指定（たとえば::name）されていない名前
- データテーブルとして適用範囲が指定（たとえば:name）されていない名前
- それらの後に括弧なしで続く名前（たとえばname(...)という形式でないもの）

次の例では、3つの数字を合計する関数を定義しています。

```
add3 = Function({a, b, c}, {temp}, temp=a+b; temp+c);
X=add3(1, 5, 9);
```

次の関数も同じ処理をしますが、ローカル変数を自動的に設定します。

```
add3 = Function({a, b, c}, {Default Local}, temp=a+b; temp+c);
X=add3(1, 5, 9);
```

どちらの場合も、変数tempはグローバル変数ではなく、たとえ関数の外で同じ名前の変数がグローバル変数として設定されていても、そのグローバル変数は関数の実行に影響されません。

注：保存された式の中にローカルな変数を指定した場合、その変数名はコンテキストにより、グローバルな領域ではグローバル変数の名前として使用されます。一方、グローバル変数として明示的に指定した変数は、グローバル変数だけを指します。

ユーザ定義の関数内でDefault Localを使用した場合、コンテキストによって動作が異なるため、混乱を招く場合があります。つまり、同名の変数がスコープの中と外に存在する場合、同じ関数でも動作が異なる可能性があります。ユーザは、ローカルにしたい変数をすべて列挙する必要があります。そうすれば、スコープの外にある変数の値についての混乱や誤りを最小限に抑えることができます。

Recurse（再帰）

Recurse関数は、定義した関数を再帰的に呼び出します。たとえば、Recurse関数を使うと、階乗を計算する関数が作れます。階乗とは、ある数から、その数が1になるまで1ずつ引いていったときのすべての数の積を指します。

```
myfactorial=function({a},if (a==1, 1, a*recurse(a-1)));
myfactorial(5);
   120
```

Recurseを使わなくても、再帰的計算を定義することはできます。たとえば、Recurseをmyfactorialに置き換えても、このスクリプトは機能します。ただし、Recurseには次のような利点があります。

- 関数と同じ名前を持ったローカル変数がある場合の競合を回避できます。
- 関数自体に名前が付けられていない場合でも（たとえば、前述の *myfactorial* など、グローバル変数に割り当てられている場合）、再帰的計算ができます。

Include

Include関数は、スクリプトファイルを開き、その中のスクリプトを解析し、JSLを指定のファイル内で実行します。

```
include("pathname");
```

例:

```
include("$SAMPLE_SCRIPTS¥myStartupScript.jsl");
```

ファイルを評価するのではなく、ファイルから式だけを取得するオプションがあります。

```
include("pathname", <<Parse Only);
```

名前空間を作成し、インクルードされたスクリプトがその中で実行されるようにするための名前付きオプションもあります。この名前空間は匿名の名前空間で、親スクリプトの名前空間から独立しています。

```
Include("file.jsl", <<New Context);
```

スクリプトでの名前空間の使用方法については、「高度な適用範囲指定と名前空間」（218ページ）を参照してください。

Includeで読み込むファイルに関しては、次のことに注意してください。

- JSLファイル以外のJMPファイルは使用できません。
- その他、JMPで認識されるファイル（イメージファイル、SASデータセット、Microsoft Excelファイルなど）も使用できません。
- 認識できない種類のファイルは、JSLファイルとして扱われます。
- 拡張子が .txt のファイルは、JSL ファイルとして扱われます。データを含むテキストファイルは読み込まれますが、有効なJSLではないためエラーメッセージが表示されます。

テキストファイルのロードと保存

Load Text FileおよびSave Text Fileコマンドを使うと、JSLでテキストファイルを操作することができます。以下のコードで、pathは文字列を表します。

```
text = Load Text File( "path" );
Save Text File( "path", text );
```

Webサイトから、テキストファイルを読み込むこともできます。

```
Load Text File( "URL", <blob> );
```

上の指定において "URL" は、テキストファイルの URL を含む引用符付き文字列です。テキストファイルは文字列として戻されます。オプションの名前付き引数 *blob* を追加した場合は、BLOB（Binary Large OBject）が戻されます。

BY グループを使ったスクリプト

By グループ引数に対応する関数は、ColMean()、ColStdDev()、ColNumber()、ColNMissing()、ColMinimum()、ColMaximum() です。

BY 引数はいくつでも指定可能です。また、式を指定することもできます。BY 引数は、必ず列計算式または ForEachRow() のコンテキスト内で使用します。第 1 引数には、一般的な数値式を指定することもできます。

たとえば、次の場合が該当します。

```
New Column( "男女別の平均身長", Numeric, Formula( Col Mean( :Name("身長 ( インチ )"), :性別 ) ) );

New Column( "男女別、年齢別の最小身長", Numeric, Formula( Col Minimum( :Name("身長 ( インチ )"), :性別 , :年齢 ) ) );

Distribution( Continuous Distribution( Column( :Name("身長 ( インチ )") ) ), By( :性別 ) );

Tabulate(
   Show Control Panel( 0 ),
   Add Table(
      Column Table(
         Analysis Columns( :Name("身長 ( インチ )")),
         Statistics( Mean, N, Std Dev, Min, Max, N Missing )
      ),
      Row Table( Grouping Columns( :年齢 , :性別 ) )
   );
);
```

ファイルをプロジェクトにまとめる

JMP のプロジェクトは、すべての操作をスクリプトでも行うことができます。次のスクリプトは、新しいプロジェクトを作成し、それにグループおよびファイルを追加し、プロジェクトの名前を取得しています。

```
expj = New Project( "私のプロジェクト" );
exg1 = expj << Add Group( "データ" );
exg2 = expj << Add Group( "レポート" );
exdt = Open( "$SAMPLE_DATA¥Big Class.jmp" );
exg1 << Add Window( exdt );
```

```
exrp = Bivariate( X( :Name("身長(インチ)") ), Y( :Name("体重(ポンド)") ));
exg2 << Add Window( exrp );
Close( exdt, NoSave );
expj << getname();
```

保存されているプロジェクトを開くには、Open Project関数を使用します。例:

```
prj = Open Project("filepath");
```

JMP 9以降では、JMPプロジェクトのスクリプトが拡張されています。表8.8に変更点をまとめます。

表8.8 プロジェクトスクリプトの変更

項目	JMP 8	JMP 9以降
ログメッセージ	すべてのプロジェクト項目が戻される。 **例** `g1 = p << Find("グループ1");` *`Project: グループ1`*	プロジェクト内の項目がより適切に表示される。 **例** `g1 = p << Find("グループ1");` `/*:` *`Project Group: グループ1`*
グループ項目のプロパティ	<< Properties メッセージは、グループ項目に送られた場合、何も値を戻さない。	グループ項目の場合も、他のプロジェクト項目と同様に項目プロパティを戻す。
Empty()を戻すナビゲーション関数	プロジェクトナビゲーションのためのメッセージ（<< GetFirstChild、<< GetNextSibling、<< GetPreviousSibling、<< GetParentなど）は、該当する項目がない場合、エラーを表示する。	プロジェクトナビゲーションのメッセージは、Empty() を戻す。これは、IsEmpty() で判定できます。 そのため、JSLでプロジェクトツリー内をナビゲートする際にTry() ステートメントを使用する必要がありません。Try() を使用する既存のスクリプトは引き続き使用できます。
新しい<< Open メッセージ	プロジェクト項目を開くには、<< OpenXXXなどの特定のメッセージが必要。	特定のメッセージは引き続き使用可能。新しい、より一般的な<< Openメッセージが使用できるようになりました。項目はこれに対して適切に応答し、ウィンドウは、たとえば、<< Open Windowを送った場合と同様に開きます。

表 8.8 プロジェクトスクリプトの変更（続き）

項目	JMP 8	JMP 9 以降
<< Properties メッセージの出力	<< Properties メッセージの出力は、1 行の長い文字列。 **例** `projectName << Properties;` *"Name:* *Created: 17Apr2008 14:27:43* *Last Accessed: 02Dec2008 20:43:12* *Last Modified: 17Apr2008 14:27:44* *Size: 299 bytes* *Restore with project: No"*	連想配列プロパティが戻され、任意のプロパティ値に直接アクセスできるようになった。 **例** `projectName << Properties;` *["Created" =>* *17Apr2008:14:27:43,* *"Exists" => "Yes",* *"Location" =>* *"C:¥Users¥saspjs¥Documents¥JMP¥Projects¥TestJournal.jrn",* *"Modified" =>* *17Apr2008:14:27:44,* *"Name" => "TestJournal.jrn",* *"Restore with project" =>* *"No",* *"Size" => 299]*
SAS ストアドプロセス	ストアドプロセスを追加するための JSL メソッドはなし。	ストアドプロセスをプロジェクトに追加できるようになった。 `p << Add Stored Process(<metadatapath>);`
ドキュメントに対する `<< Properties` メッセージ	最後にアクセスした日付けが戻される。	最後にアクセスした日付けが戻されなくなった。
非表示オプション	開いたプロジェクトまたは作成したプロジェクトはすべて表示される。	`Open Project()` および `New Project()` で、オプションの第 2 引数（`invisible`）を指定できる。非表示のオプションはプロジェクトウィンドウを表示しません。 **注：**JMP を閉じるときに非表示のプロジェクトが開いていても、プロジェクトを保存するかどうかを確認するダイアログボックスは表示されません。スクリプトを作成する際は、非表示のプロジェクトを保存してから閉じてください。

表 8.8 プロジェクトスクリプトの変更（続き）

項目	JMP 8	JMP 9 以降
フォルダ項目	プロジェクト内のフォルダ項目に、子の項目のスクリプト可能なオブジェクトを作成するためのメッセージはない。	フォルダに対して `<< Open` メッセージを送った際に、子の項目のスクリプト可能なオブジェクトが作成される。スクリプト可能なオブジェクトは、パフォーマンスとメモリの理由により、子の項目に自動的には作成されません。
新しい `<< Add All Windows` メッセージ	JMP 8 では使用不可。	`projectName << Add All Windows()` は、開いたウィンドウをすべてプロジェクトに追加し、追加されたウィンドウ項目のリストを戻す。
`<< Set Restore All On Project Open` メッセージの変更	次の引数を取る。 • 0（どのウィンドウも復元しない） • 1（すべてのウィンドウを復元する）	次の引数を取る。 • 0（どのウィンドウも復元しない） • 1（すべてのウィンドウを復元する） • -1（指定したウィンドウを復元する）

スクリプトの暗号化と暗号解読

スクリプトに基本レベルの保護を付加するために、暗号化して、パスワードを知っている人だけが見たり実行したりできるようにできます。これは、スクリプトの共有を管理したい場合に便利です。

スクリプトを暗号化するには

1. 暗号化するスクリプトを開きます。
2. ［**編集**］>［**スクリプトの暗号化**］を選択します。
3. ユーザがスクリプトを見るのに必要となる、暗号解読のためのパスを入力します。
4. （オプション）実行パスワードを入力します。ユーザは、このパスワードを入力しなければ暗号化されたスクリプトを実行できません。

注：パスワードには 1 バイトの文字を使用しなければなりません。IME（Input Method Editor）を使ったテキストの入力は無効です。

5. ［**OK**］をクリックします。
6. 暗号解読パスワードのみを指定した場合、［**はい**］をクリックして、実行パスワードを指定しないことを確認します。

   暗号化されたスクリプトが、新しいウィンドウに表示されます。たとえば、次のような設定が行えます。

```
//-e6.0.2
S@FTQ;VGMUTF?J<;LS;B<=IRLXCU=BV;@NS<TW;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJ
   S>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<HNIZ;W
   DN?RMJ;FR>KYAXTEPPF?;XFJJOP=RQGBIAGXOYNNZ>PLIF>SW>L>ACL<KGP;=QQTCEG??U<PUXLV?TR
   BO?J>QGWTJCFJA@BNHWLVORNNGQYPIKL<IM<>JX>@G?LJ>=;RBODH@PTKK@SIUE;IJOR<TUTRMTGSYR
   SVGOR<XK<F=IWQYE=LVZFP;AUHA?YJLL;EIT?ZJZC;*
```

7. 暗号化されたスクリプトを保存します。

JSL スクリプトを暗号解読するには、次のようにします。

1. 暗号化されたスクリプトを JMP で開きます。
2. ［**編集**］>［**スクリプトの暗号解読**］を選択します。
3. 暗号解読パスワードを入力し、［**OK**］をクリックします。

   暗号解読されたスクリプトが新しいウィンドウに表示されます。

暗号化された JSL スクリプトを実行するには

注：暗号化されたスクリプトを実行する前に、スクリプトを実行する対象となるデータテーブルを把握しておく必要があります。データテーブルの名前がわからない場合は、実行する前にスクリプトを暗号解読して、データテーブルの名前を確認しておく必要があります。

1. 暗号化されたスクリプトを JMP で開きます。
2. ［**編集**］>［**スクリプトの実行**］を選択します。
3. 実行パスワードを入力し、［**OK**］をクリックします。

   スクリプトは次のように実行されます。

   – スクリプトがデータテーブルを参照している場合、データテーブルを開くように求められます。データテーブルを開くと、スクリプトが実行されます。

   – スクリプトが空のデータテーブルを必要とする場合は、まずデータテーブルを作成します。その後、暗号化されたスクリプトを実行します。

実行パスワードを入力した場合、スクリプトは実行されますが、表示はされません。スクリプトを表示するには、暗号解読パスワードを入力する必要があります。

暗号化とグローバル変数

暗号化だけでは、グローバル変数とそれらの値を隠すことはできません。`Show Globals()` コマンドを使えば、それらは通常どおり表示されます。暗号化スクリプト内のグローバル変数を隠す必要がある場合は、それらに特別な名前を付けます。

名前が 2 つの下線（__）で始まるグローバル変数は、すべて非表示となります。`Show Globals()` では名前も値も表示されません。たとえば、次のような設定が行えます。

```
myvar = 2;
__myvar = 5;
Show Symbols();
   //Global
   myvar = 2;
   // 2 Global (1 表示しない)
```

この方法は、スクリプトが暗号化されているかどうかに関わりなく有効です。

データテーブルのスクリプトの暗号化

データテーブルに保存されたスクリプトは、JSL Encrypted() 関数または Include(Char to Blob()) 関数を使って暗号化することもできます。

- JSL Encrypted() の方が、1つの関数を含むだけなので簡単です。暗号化スクリプトの中にコメントを含めることもできます。
- Include(Char to Blob()) では、コメントを含めることはできますが、スクリプトの中ではありません。

データテーブルのスクリプトを暗号化するには、次の手順に従います。

1. スクリプトをスクリプト編集ウィンドウに配置します。

   すでにデータテーブルに保存されているスクリプトを直接暗号化することはできません。
2. 「スクリプト」ウィンドウで、[**編集**] > [**スクリプトの暗号化**] を選択します。
3. 暗号解読パスワードを入力します。
4. （オプション）実行パスワードを入力します。ユーザは、このパスワードを入力しなければ暗号化されたスクリプトを実行できません。
5. 暗号解読パスワードのみを指定した場合、[**はい**] をクリックして、実行パスワードを指定しないことを確認します。

   暗号化されたスクリプトが、新しいスクリプトウィンドウに表示されます。
6. 暗号化されたスクリプト全体をコピーします。
7. 新しいデータテーブルスクリプトを作成するか、既存のスクリプトを開きます。
8. ウィンドウ内のスクリプトの部分に、次のいずれかの関数を入力します。

   ```
   JSL Encrypted( "" );
   ```

   ```
   Include( Char to Blob( "" ) );
   ```
9. 関数の引用符の中に暗号化スクリプトを貼り付けます。
10. [**OK**] をクリックします。

図 8.3 暗号化されたデータテーブルスクリプトの例

```
JSL Encrypted(
    "//-e6.0.2
WE@GSACGT<?CKEG=NG;B<=IRLXCU=BV;@NS<TW;LR<ZFOP=JJS>NNDA@T<V><DZA
>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA
>SU@MG;LR<ZFOP=JJS>NNDA@T<FFHG=R;GYNBNME>ZBMIB?O;G<>?;UTG@=HVYCC
@MFYCMVQEOTNWASNOEV=Y<TCELCQHW>AFY>;PGSSF<IBIU?H<YLQABWEGSQBMLHW
BJE?HUC=BTP?;?;LVTT>ZQI=IXTJ<P=IHYJIPAC?PGZH;OCCMXPOJHLIRHERSW;V
;@GZDHDNF><YCN@WLBTOFZRAILT?L;NT<OOCLLLWYAIVCSPBMVL???KK=TBZJ@KU
BD>=@;EOUFMUUFLSP<HMZWL?VLHXKK?S@WAFXC>EZODQOHREMNYKRBKAWS=PSXSQ
CRYFOPVJRBNHIVLE?BNPVNM=LO>BAT<?CQF?JNNQAUEHMM;BTU><=WFVOO<;WVKB
;RSBEXTS;QGBEKNDCANPFP;DYDHTSMQNAGVESB;TCWJKGWNBG>JBESVNKOVOBPBE
;?ZPYS;X?=VE=>YFHD;;;*"
)
```

その他の数値演算子

JSL では、計算式エディタではあまり意味をなさない、関数の行列計算や数値微分も実行できます。微分については、以下で詳しく説明します。

行列に対しても、足し算や引き算などの基本的な演算を、行列をオペランドとして実行することができます。また、行列には、要素ごとの掛け算、要素ごとの割り算、行列同士の連結、要素の指定といった行列固有の演算子もいくつかあります。詳細については、「データ構造」の章の「行列」（154 ページ）の章を参照してください。

微分

JSL には、微分を求めるための関数は 3 つあります。Derivative、NumDeriv、および NumDeriv2 です。これらの関数は、計算式エディタでは用意されていません。

`Derivative` 関数は、第 2 引数に指定した変数で、第 1 引数に指定された式を微分した結果を戻します。この第 2 引数には 1 つの変数を指定することも、リストにして複数の変数を指定する（つまり、中括弧 { } で複数の変数名を囲む）こともできます。

注： `Derivative` 関数と同様の機能は、計算式エディタではコマンドとして用意されています。このコマンドは、計算式エディタの上部中央のドロップダウンリストにあります（キーパッドの上）。このコマンドを使用するには、式の中の 1 つの変数を強調表示（どの変数についての導関数を求めるかを指定）してから、メニューから［**微分した式**］コマンドを選びます。計算式全体が、強調表示された変数で微分した導関数に置き換わります。

スクリプトでこの関数を最も簡単に使用する方法は、変数を 1 つ指定する方法です。この場合は、導関数のみが戻されます。

$f(x) = x^3$ の場合、第 1 次導関数は $f'(x)$ または $\frac{d}{dx}x^3 = 3x^2$ です。

```
result = derivative(x^3, x); show(result);
   result = 3 * x ^ 2
```

ある 1 つの式から複数の変数に対する導関数を効率的に求めたい場合は、リストに複数の変数を指定してください。**一時変数に分割された**元の式と、その導関数の式を含むリストが結果として戻されます。必要に応じて、導関数で用いる部分式が、**一時変数**に割り当てられます。

次の例は、3つの変数が含まれている式の例です。3つの変数をリストに入れると、それぞれの変数に対する第1次導関数を戻します。結果は、元の式と、要求した順に並べた導関数を含むリストになります。ただし、この例において、JMP が式 x^2 を格納するための一時的な変数 T#1 を作成し、この一時的な変数を使って計算を省略していることに注意してください。

```
result2=derivative(3*y*x^2+z^3, {x,y,z}); show(result2);
   result2 = {3 * y * (T#1 = x ^ 2) + z ^ 3, 6 * x * y, 3 * T#1, 3 * z ^ 2}
```

第2次導関数をとるには、第3引数としてもう1つ変数を指定します。第2引数と第3引数は、どちらもリストでなければなりません。JMP は、元の式、第1次導関数、そして第2次導関数の順番で結果を含んだリストを戻します。

```
second=derivative(3*y*x^2,{x},{x}); show(second);
   second = {3 * y * x ^ 2, 6 * x * y, 6 * y}

second=derivative(3*y*x^2,{y},{y}); show(second);
   second = {3 * y * (T#1 = x ^ 2), 3 * T#1, 0}

second=derivative(3*y*x^2,{y},{x}); show(second);
   second = {3 * y * (T#2 = x ^ 2), 3 * T#2, 6 * x}
```

NumDeriv は、数値微分を行うための関数です。第1引数と、それに小さい値（Δ）を足した値に対する関数値を計算し、その差をΔで割ることにより、1次微分の値を求めます。NumDeriv2 は、同様の方法で2次微分の値を数値的に求めます。これらの関数は、非線形モデルで内部的に使用されますが、JSL で直接利用することは少ないでしょう。これらの関数は変数では微分せず、関数に対する引数についてのみ微分することに注意してください。x について微分するためには、x を、式の内部に埋め込まれている記号ではなく、関数の中の引数の1つにする必要があります。

$x = 3$ で、$y = 3*x^2$ を微分するとします。このとき次のスクリプトを実行すると、**間違い**になります。

```
x=3;
n=NumDeriv(3*x^2);
```

x を関数の中の引数にするのが、**正しい**方法です。

```
x=3;
f=function({x}, 3*x^2);
n=NumDeriv(f(x), 1);
```

数式による表記と、JSL での表記方法を例を挙げて検討してみましょう。

$f(x) = x^2$ の場合、$\frac{d}{dx}x^2 = \frac{(x+\Delta)^2 - x^2}{\Delta}$ という計算になります。$x_0 = 3$ では、$\frac{d}{dx}x^2 = 6.00001$ です。

```
x=3; y=numderiv(x^2);  // または、y = numDeriv(3^2)
   6.0000099999513
```

もう少し例を挙げます。

```
x = numderiv(sqrt(7));
y = numDeriv(Normal Distribution(1));
z = Num deriv2(normal distribution(1));
```

表 8.9 微分を行う関数

関数	構文	説明
Derivative	Derivative(*expr*, {*name*, ...})	*name* に対する *expr* の導関数を戻す。第 2 引数は、中括弧 { } でリストにして指定するか、1 つだけの場合は単純に変数として指定できます。2 つの導関数をとるときは、2 つの名前リストを指定します。
NumDeriv	NumDeriv(*expr*)	式（*expr*）の最初の引数で、1 次微分した結果（1 階導関数の値）を数値的に求める。
NumDeriv2	NumDeriv2(*expr*)	式（*expr*）の最初の引数で、2 次微分した結果（2 階導関数の値）を数値的に求める。

代数的な処理

JSL には、逆関数を代数的に求める Invert Expr() 関数が用意されています。

```
Invert Expr(expression, name, y)
```

この式で、

- *expression* には逆関数を求めたい式、もしくは、その式を含むグローバル変数を指定します。
- *name* について解かれた逆関数が求められます。
- *y* は元の式の従属変数の変数名です。

例 :

```
Invert Expr(sqrt(log(x)),x,y)
```

この結果は、変数 x について解かれた次のような逆関数になります（なお、Invert Expr() 関数は、変数 x が、元の式に 1 箇所だけで使われている必要があります)。

```
exp(y^2)
```

これは、次のように手計算で代数的に求めるのとまったく同じように行われます。

$y = \mathrm{sqrt}(\log(x))$

$y^{\wedge 2} = \log(x)$

$\exp(y^{\wedge 2}) = x$

Invert Expr関数は、逆関数が存在する大部分の基本演算をサポートしています。また、必要に応じて、正の平方根の中だけで、三角関数の場合は可逆な定義域内だけで逆関数を求めます。

F、Beta、Chi Square、t、Gamma、およびWeibullの*Distribution*と*Quantile*関数がサポートされています。第1引数に関しての逆関数である分布関数と分位点関数が戻されます。変換できない式の場合は、Invert Expr()関数はEmpty()を戻します。

JSLには、整理されていない複雑な式を、さまざまな代数的規則を用いて簡略化する、Simplify Exprコマンドがあります。このコマンドの使用方法は、以下のとおりです。

```
result = Simplify Expr(expr(expression));
```

または

```
result = Simplify Expr(nameExpr(global));
```

例:

```
Simplify Expr (expr(2*3*a+b*(a+3-c)-a*b));
```

の結果は、次のようになります。

```
6*a + 3*b + -1*b*c
```

Simplify Exprはまた、入れ子構造のIf式を展開します。たとえば、次のような設定が行えます。

```
r = simplifyExpr(
  expr(If(cond1,result1,if(cond2,result2,if(cond3,result3,resultElse)))));
```

の結果は、次のようになります。

```
If(cond1, result1, cond2, result2, cond3, result3, resultElse);
```

最大化と最小化

Maximize関数およびMinimize関数は、式を最適化（最大化または最小化）する因子の値を求めます。式は、因子の値に対して連続関数でなければいけません。

この関数を呼び出す方法は、次のとおりです。

```
result = Maximize(objectiveExpression,{list of factor names}, <<option(value))
result = Minimize(objectiveExpression,{list of factor names}, <<option(value))
```

*objectiveExpression*は、最適化する対象の式です。式そのもの、または式が保存されているグローバル変数名のどちらかを指定します。

{*list of factor names*}には、*objectiveExpression*に含まれる因子名のリストを指定します。

因子名の後に、name(lowerBound,upperBound)のように、許容値の範囲を指定することもできます。

許容値の上限または下限だけを指定する場合は、以下の例のように片側を欠測値にします。

```
{beta} // 制限なし
{beta (0,1)} //0 以上 1 以下
{beta (.,1)} //1 以下
{beta (0,.)} or {beta (0)} // 0 以上
```

因子の値は、数値または行列のいずれかです。

また、以下のオプションも使用できます。デフォルト値は、以下のとおりです。

```
<< tolerance(.00000001) // 収束基準
<< maxIter( 250) // 最大反復数
<< limits() //
<<Method(NR|BFGS) // Newton-Raphson 法または BFGS 更新の準 Newton 法を指定する
```

関数を呼び出す前に、因子には初期値が割り当てられているものとします。

これらの関数は、複数の局所解を持つ関数において、大域的な最適解を見つけるとは限りません。指定された初期値から、ある局所的な最適解（問題によっては、大域的な最適解かもしれませんが）を求めることしかできません。

現在のところ、戻り値は目的関数の値です。

最小 2 乗法の例

Minimize を使用して、指数モデルにおいて最小 2 乗法の推定値を求める例を示します。サンプルデータのフォルダ内にある「Nonlinear Examples」の「US Population.jmp」データテーブルのデータを使用します。

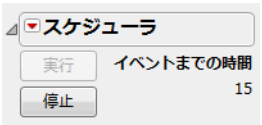
```
xx = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910,
  1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990];
yy = [3.929, 5.308, 7.239, 9.638, 12.866, 17.069, 23.191, 31.443, 39.818, 50.155,
  62.947, 75.994, 91.972, 105.71, 122.775, 131.669, 151.325, 179.323, 203.211,
  226.5, 248.7];
b0 = 3.9;
b1 = .022;
sseExpr = Expr( Sum( (yy - (b0 * Exp( b1 * (xx - 1790) ))) ^ 2 ) );
sse = Minimize( sseExpr, {b0, b1}, <<tolerance( .00001 ) );
Show( b0, b1, sse );
```

スクリプト実行のスケジューリング

Schedule 関数を使うと、指定した秒数後にスクリプトを実行できます。

```
schedule(15, print(" こんにちは "));
```

図 8.4 JMP スケジューラ



「スケジューラ」ウィンドウには、次のイベントまでの時間が表示されます。また、スケジュールを再開するためのボタン（［**実行**］）や停止するためのボタン（［**停止**］）があります。このウィンドウのポップアップメニューには［**スケジュールの表示**］がコマンドあり、現在のスケジュール内容をログに出力します。たとえば、上の「こんにちは」の例の実行中に何度かスケジュールを確認すると、次のような表示が見られます。

```
スケジュール時間 11.55000000000018 :Print(" こんにちは ")
スケジュール時間 4.716666666666697 :Print(" こんにちは ")
スケジュール時間 3.083333333333485 :Print(" こんにちは ")
```

このスクリプトは、格納された式を参照する名前でもかまいません。たとえば、次のようにそのスクリプト自体を呼び出すスクリプトを実行してみましょう。

```
quickieScript = expr( show(" やあ、どうも "); schedule(15, quickieScript); );
quickieScript;
```

このスクリプトは、15 秒後にログウィンドウに文字列「やあ、どうも」を表示し、そのさらに 15 秒後に同じスクリプトが実行されるように再スケジュールします。これは［**停止**］ボタンがクリックされるまで続きます。

また、プロダクションの設定として、次のようなスケジュールを設定することもあるでしょう。

```
FifteenMinuteCheck = expr(show(" データのチェック ");
   open("my file", options...);
   distribution(column(column1), capability(spec limits));
   schedule(15*60, FifteenMinuteCheck));
FifteenMinuteCheck;
```

Schedule はイベントキューを開始します。イベントがキューに置かれると、スクリプト内の次のステートメントの実行に移ります。たとえば、次の例のような結果になることもあります。

```
schedule(3, print(" いち "));
print(" に ");
   " に "
   " いち "
```

イベントが終了するまで、それ以降のスクリプトが実行されないように待機させる方法の 1 つに、Wait( ) を使って適当な停止時間を設定する方法があります。別の方法としては、以降の処理をスケジュールのキューに入れます。Schedule では、たくさんのイベントを連続的にキューに入れられるようになっています。複数のイベントをスケジュールに設定した場合、各イベントは、個別に呼び出されます。各イベントの実行時間はすべて、「今」から（または、［**実行**］がクリックされてから）の時間で指定してください。たとえば、以下の複数のイベントは、12 秒間ではなく 5 秒間ですべて終了します。

```
schedule(3, print(" こんにちは "));
schedule(4, print("、世界のみなさん "));
schedule(5, print("-- さようなら "));
```

スケジュールキュー上のイベントをすべてキャンセルするには Clear Schedule を使います。

```
scheduler[1]<<Clear Schedule( );
```

注: Schedule を使ってマルチスレッドで動作させることはできません。

表 8.10 スケジュールのコマンド

メッセージ	構文	説明
Schedule	*sc*=Schedule(*n*, *script*)	*n* 秒後に指定のスクリプト（*script*）を実行するというイベントをキューに入れる。
Clear Schedule	*sc*<<Clear Schedule( )	スケジュールのキューにあるイベントを、すべて停止する。

メッセージを出力する関数

Show、Print、および Write は、ログウインドウにメッセージを表示します。Speak、Caption、Beep、および StatusMsg は、ユーザに情報を与える方法を提供します。Mail では、プロセス管理者に警告の電子メールを送ることができます。

JMP のスクリプト言語を使うと、ユーザにデータ列の選択やその他の情報入力を促すためのダイアログボックスを作成できます。「表示ツリー」章の「モーダルウィンドウ」（435 ページ）を参照してください。

ヒント： Show、Print、Write の出力でロケール特有の数値の表示形式を使用するには、<<Use Locale(1) オプションを含めます。

ログへの書き込みを行う

Show

Show は、指定の項目をログに表示します。変数を表示させる場合、結果のメッセージは、変数名、コロン（:）、および変数の現在の値となります。

```
X=1;A="Hello, World";
show(X,A,"foo");
   x = 1
   A = "Hello, World";
   "foo"
```

Print

Printは、指定のメッセージをログに送ります。Printは、変数名とコロンなしで変数の値だけを表示する点を除けば、Showと同じです。

```
X=1;A="Hello, World";
print(X,A,"foo");
   1
   "Hello, World"
   "foo"
```

Write

Writeは、指定のメッセージをログに送ります。WriteとPrintの違いは、Writeでは文字列の両側の引用符が省略されることと、改行させるにはエスケープシーケンスの¥!Nを挿入する必要があるという点です。

```
write(" これがメッセージです。");
   これがメッセージです。

myText=" これがメッセージです。";
write(myText||" ミルクを買うのを忘れないで。");// 連結するには || を使用
write("\!N それからパンも。"); // 改行するには \!N を使用
   これがメッセージです。ミルクを買うのを忘れないで。
   それからパンも。
```

シーケンス\!Nは、ホスト環境に応じた復帰文字および改行文字を挿入します。3つの改行エスケープシーケンスについては、「JSLの構成要素」章の「二重引用符」（82ページ）を参照してください。

ユーザに情報を送る

Beep

Beepは、警告音を鳴らします。

Speak

Speakは、テキストを読み上げます。Macintoshでは、Speakにブール値（1または0）をとるオプションのWaitがあり、読み上げの終了まで次のステップへ進むのを待つかどうかを指定できます。デフォルトでは待たない設定になっているため、待つようにするには毎回Wait(1)を送る必要があります。たとえば、ここによく意味のわからない文を読み上げるスクリプトがあるとします。Wait(1)を使えば言葉は聞き取れるので、すぐに実行を中断したくなるはずです。しかし、Wait(1)ではなくWait(0)を使った場合、繰り返しの方が読み上げが完了するより早く処理されるため、結果として変な音が出力され、一体何が起こっているのかもわかりません。Windowsでは、Wait(n)コマンドが同じ機能を果たします。

```
for(i=99,i>0,i--,
   speak(wait(1),char(i)||" bottles of beer on the wall, "
      ||char(i)||" bottles of beer; "
      ||"if one of those bottles should happen to fall,"
      ||char(i-1)||"  bottles of beer on the wall."))
```

もっと実用的な例としては、JMPに60秒ごとに時間を読み上げさせるというものがあります。

```
// 時報
script = expr(
        tod = mod(today(),indays(1));
        hr  = floor(tod/inHours(1));
        min = floor(mod(tod,inHours(1))/60);
        timeText = "time, " || char(hr) || " " || char(min);
        text = Long Date(today()) || ", " ||timeText;
        speak(text);
        show(text);
        schedule(60,script);    // 次のスクリプトまでの秒数
);
script;
```

同じような方法を使って、プロセスが制御不可能になったことをJMPから警告させることもできます。

Caption

Captionは、小さなウィンドウを開き、ユーザへのメッセージを表示します。この機能により、結果のウィンドウに余分なオブジェクトを加えることなしに、デモに注釈をつけられます。第1引数の{x,y}はオプションで、スクリーン上の位置を左上からのピクセル数で指定するものです。第2引数は、ウィンドウに表示するテキストです。位置の引数が省略された場合、ウィンドウは左上隅に表示されます。

Spokenオプションを指定すると、注釈が読み上げられます（OSに音声出力システムが組み込まれている場合）。Spokenはブール値（1または0）の引数をとり、Spoken引数を持つCaptionコマンドによって改めて変更されるまでは現在の設定（オンまたはオフ）が有効になります。

読み上げを一定時間遅らせるには、名前付き引数Delayedと時間（秒）を指定します。これにより、この指定と同時に送られる注釈と、**それ以降のすべての**注釈には、指定された秒数分の遅れを生じさせることができます。これは、CaptionコマンドでDelayedの設定が更新されるまで続きます。遅れをすべてなくすには、Delayed(0)を使います。

下のスクリプトでは、読み上げをオンにし、最後の注釈までオンのままにしています。

```
caption({10,30},"JMP の紹介 ", spoken(1), Delayed(5));
caption(" データテーブルを開きます ");
   bigClass=open("$SAMPLE_DATA¥Big Class.JMP");
caption("JMP データテーブルはデータの行と列で構成されています ");
caption(" 行には番号、列には名前がついています ");
caption({250,50}," データそのものは右側のグリッドに表示されます ");
caption({5,30},spoken(0)," 左側のパネルには列その他の属性が表示されます ");
```

新しいCaptionを指定するたびに、以前の注釈が非表示になります。つまり、一度に表示できる注釈ウィンドウは1つだけです。新しい注釈を表示せずにウィンドウを閉じるには、名前付き引数Removeを使います。

```
caption(remove);
```

StatusMsg

このコマンドは、ステータスバーにメッセージを送信します。

```
StatusMsg("string")
```

Mail

（Windowsのみ）Mailは、JMPの状態についての警告メッセージを、電子メールでユーザに送ります。たとえば、プロセス制御管理者は、管理図にテストの警告スクリプトを含めておいて、警告の電子メールが管理者に届くように指定できます。

```
mail("JaneDoe@company.com", " 制御不能 ", Format(today(), "d/m/y h:m:s")||" プロセス
  12A が制御不能 ");
```

Mailは、電子メールの添付ファイルも送ることができます。オプションとして4つ目の引数に添付ファイルを指定します。添付ファイルは、ディスク上に存在することが確認されると、バイナリー形式で送信されます。たとえば、「Big Class.jmp」データテーブルを添付するには、次のスクリプトを実行します。

```
mail("JohnDoe@company.com", " 興味深いデータセット ", " このクラスのデータをご覧ください。",
  "C:¥myJMPData¥Big Class.jmp");
```

第 9 章

データテーブル
データテーブルオブジェクトの操作

データテーブルやデータテーブル内のオブジェクトを操作するには、まずデータテーブルを開き、そのオブジェクトに**参照**を割り当てる必要があります。このマニュアルでは、データテーブルオブジェクトの参照を**dt**と表記します。

JSL でオブジェクトを操作するには、そのオブジェクトを表す参照にメッセージを送り、タスクのどれかを実行するよう要求します。**メッセージ**とは、特定の状況のもとで、特定のタイプのオブジェクトだけが理解できるコマンド群のことです。たとえば、データテーブルオブジェクトへのメッセージには、Save、New Column、Sort などがあります。これらのメッセージのほとんどは、たとえばプラットフォームオブジェクトなど、他のオブザベーションに対しては何の意味も持ちません。

この章では、次のことを学びます。

- データテーブルに参照を割り当てる方法
- 参照にメッセージを送り、指定のアクションを実行させる方法
- データテーブルオブジェクトが理解できるメッセージの種類

この章の大部分は、データテーブルや列、行、値などのデータテーブルオブジェクトに送ることができる各種メッセージに焦点を当てています。

ヒント：データテーブルの操作に使用できる JSL コマンドの大半は説明してますが、すべてを網羅しているわけではありません。包括的なリストについては、JMP の［スクリプトの索引］を参照してください。[**ヘルプ**] > [**スクリプトの索引**] を選択します。メニューから [**オブジェクト**]、次に [**データテーブル**] を選択します。

目次

はじめに 259

データテーブルのスクリプトの基本 261

- データテーブルを開く 261
- 新しいデータテーブルの作成 263
- データの読み込み 264
- 現在のデータテーブルの設定 271
- データテーブルに名前をつける 271
- データテーブルの保存 272
- データテーブルを非表示にする 273

データテーブルの高度なスクリプト 277

列 294

行 312

データ値へのアクセス 337

データテーブルへのメタデータの追加 339

計算 343

はじめに

ヒント：ログウィンドウを開いたままにして、実行したスクリプトの出力を確認しましょう。[**表示**] > [**ログ**] を選択し、ログウィンドウを表示します。「スクリプト作成のツール」の章の「ログの使用」（62 ページ）を参照してください。

データテーブル内の値を処理する場合、次のような手順が一般的です。

1. 使用したい値が入っているデータテーブルを、現在のデータテーブルとして設定する。すでにデータテーブルの参照がある場合は、その参照を使用することができます。
2. 使用したい値が入っている行または列を指定し、使用したい値が含まれている列名を指定します。

次の例では、「Big Class.jmp」データテーブルを開き（それにより、このデータテーブルが「現在のデータテーブル」になります）、「体重」列の行 2 を指定します。ログを見ると、123 という値が戻されていることがわかります。これは、行 2 の「Louise」の体重です。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt:weight[2];
   123
```

データテーブルがすでに開いている場合は、次の例のいずれかを使用します。

```
dt = Data Table("My Table"); // 「My Table」という名前のすべてに開いているテーブル
dt = Current Data Table();  // アクティブなウィンドウにあるテーブル
dt = Data Table(3); // 3 番目に開かれたテーブル
```

参照のあるデータテーブルが開いたら、<<演算子か Send 関数を使ってデータテーブルにメッセージを送ります。次の例では、両方の使い方を示します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Save();
Send(dt, Save());
```

メッセージについて、次の点に留意してください。

- メッセージは、どのデータテーブルオブジェクトにも送ることができます（データテーブル、列、行など）。
- データテーブルオブジェクトに送られるメッセージは、常に次のようなパターンを持ちます。

  参照 << メッセージ（引数）;
- 通常は、参照を作成し、その参照にメッセージを送るのが最も簡単です。しかし、メッセージを 1 つしか送らない場合は、参照を使わずに直接送ることもできます。次の例は、現在のデータテーブルを保存します。

  ```
  Current Data Table() << Save();
  ```
- 一連のメッセージを 1 つのステートメントに連ねることもできます。コマンドは左から右へと評価され、各コマンドが、影響するオブジェクトの参照を戻します。次の例は、「My Table」という名前の新しいデータテーブルを作成し、そこに 2 つの列を追加した後、データテーブルを保存します。

```
dt = New Table("My Table");
dt << New Column("Column 1") << New Column("Column 2") << Save("");
```

この例では、各メッセージがデータテーブル（dt）の参照を戻します。

- メッセージの引数が足りないと、JMP は必要な情報の入力を求めるウィンドウを表示します。JMP は、スクリプトが不完全な場合によくウィンドウを表示します。JMP のこの動作は、ユーザによる入力が必要なスクリプトを作成するときに利用できます。
- 一部のメッセージは対になっており、一方は各属性を設定（set）または割り当て、もう一方は各属性の現在の設定を取得（get）または問い合わせます。

オブジェクトに送るコマンド（メッセージ）と単独で使うコマンドがある理由

New Table と Open は、まだ存在しないオブジェクトを作成するためのコマンドです。オブジェクトが作成できたら、そのオブジェクトに変更を求めるメッセージを送ります。オブジェクトは自身を削除できないので、オブジェクトを閉じるには、オブジェクトの**コンテナ**を閉じる必要があります。

次の例は、テーブルを作成し、**dt** 変数にデータテーブル参照を割り当てます。そのデータテーブルの参照に New Column メッセージを送ります。これらの列のいずれかを削除する場合は、列そのものではなく、データテーブルの参照に Delete Columns メッセージを送ります。

```
dt = New Table("Airline Data");
dt << New Column("Date");
dt << New Column("Airline");
dt << Delete Columns("Date");
```

データテーブルオブジェクトに送ることができるメッセージ

データテーブルオブジェクトに送ることができるメッセージは、次のような手順で、［スクリプトの索引］で確認できます。

1. ［**ヘルプ**］>［**スクリプトの索引**］を選択します。
2. 表示する種類のドロップダウンメニューから［**すべてのカテゴリ**］を選択します。
3. リストの中から［**Data Table**］を選択します。

Show Properties コマンドを使用することもできます。このコマンドは、メッセージのリストを印刷したい場合やコピーしたい場合に便利です。Show Properties コマンドは、データテーブルに送ることのできるメッセージを「ログ」ウィンドウに一覧表示します。Show Properties は、データテーブルや列など、スクリプト可能なオブジェクトを引数としてとるコマンドです。データテーブルのプロパティを表示するには：

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show Properties( dt );
```

結果のメッセージは、階層型のリストに表示されます。

[サブテーブル] は、JMP のメニューに表示されるコマンドセットを指します。たとえば、「Tables」というサブテーブルは、JMP の［テーブル］メニューを表します。このサブテーブルに、インデントされた形で表示されるメッセージは、［テーブル］メニューのコマンドに該当します。メッセージの多くに、短い説明がついています。

[アクション] のラベルがあるメッセージはすべて、何らかのアクションが実行されます。スクリプトでしか使用できないものには、[スクリプトの場合のみ] のラベルがつきます。ブール値の引数をとるメッセージには、[ブール値] のラベルがつきます。

［**分析**］メニューと［**グラフ**］メニューのプラットフォームがプロパティのリストに表示されているのは、プラットフォーム名をメッセージとしてデータテーブルに送れるからです。プラットフォーム名をメッセージとして送ると、そのデータテーブルに対して通常の起動ウィンドウが開きます。プラットフォームに関連するスクリプトの書き方については、「プラットフォームのスクリプト」（347 ページ）の章を参照してください。

```
dt << Distribution(Y(:Name(" 身長 ( インチ )")));
```

注：Show Properties は、データテーブルだけでなく、プラットフォームとディスプレイボックスにも使用できます。

これらのプロパティに関する詳細は、JMP の［スクリプトの索引］を参照してください。［スクリプトの索引］では、サンプルスクリプトの実行や変更もできます。［**ヘルプ**］>［**スクリプトの索引**］を選択し、［オブジェクト］リストの中でプロパティを探します。

データテーブルのスクリプトの基本

データテーブルオブジェクトを使用するには、まずデータテーブルを開くか、または新しく作成し、そのデータテーブルに参照を割り当てる必要があります。この節では、データテーブルに対して実行できる、命名、保存、サイズ変更などの基本的なアクションについて説明します。

データテーブルを開く

データテーブルを開くには、Open() 関数を使用します。

- データテーブルの参照を戻さずに、ただデータテーブルを開くには：

  ```
  Open("$SAMPLE_DATA¥big class.jmp"); // データテーブルを開くのみ
  ```

- データテーブルを開いてその参照を維持するには：

  ```
  dt = Open("$SAMPLE_DATA¥big class.jmp"); // 開いて参照を維持する
  ```

データテーブルのパスとしては、引用符で囲んだ文字リテラル（絶対パスまたは相対パス）か、パス名を戻す引用符のない式が使えます。相対パスは、.jsl ファイルの場所を基準として相対的に解釈されます（保存されたスクリプトの場合）。保存されていないスクリプトの場合、パスは、プライマリパーティション（Windows の場合）または *<ユーザ名>*/Documents フォルダ（Macintosh）との相対で解釈されます。

```
Open("../My Data/Repairs.jmp"); // Windows および Macintosh における相対パス
Open("../My Data/Repairs.jmp"); // Macintosh における相対パス
Open("C:/My Data/Repairs.jmp"); // 絶対パス
```

JMP には、ディレクトリやファイルにより簡単にアクセスする方法（**パス変数**）が用意されています。この方法では、ディレクトリやファイルの完全なパスを入力する代わりに、Open() 式にパス変数を含めます。たとえば、JMP のサンプルスクリプトで広く使われるパス変数の $SAMPLE_DATA は、Samples/Data フォルダにあるファイルを開きます。パス変数の詳細については、「データタイプ」の章の「パス変数」（120 ページ）を参照してください。

データテーブルを開くたびに完全パスを入力する手間を省くには、ファイルパスの文字列を定義して、それをファイル名に連結します。

```
myPath = "C:/My Data/Store25/Maintenance/Expenses/";
Open( myPath || "Repairs.jmp" );
```

この方法でデータテーブルを開くと、開いたデータテーブルがメモリに格納されます。つまり、スクリプトで開けようとしているデータテーブルがすでに開いている場合は、コンピュータに保存されているデータテーブルが開く代わりに、開いた状態でメモリに格納されたデータテーブルが表示されます。

開いたデータテーブルのテスト

開いたデータテーブルに依存するスクリプトでは、テーブルが開いているかどうかを Is Empty() または Is Scriptable() を使ってテストします。次の例では、スクリプトが「Big Class.jmp」で二変量の分析を行った後、データテーブルを閉じます。スクリプトで後述されている一元配置分析に進む前に、If Scriptable() によって、開いたデータテーブルがテストされます。1（真）が戻された場合、テーブルは開いており、スクリプトは続行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
obj = Bivariate( Y( :Name("身長 (インチ)") ), X( :年齢 ), Fit Line );
Close( dt );
If( Not( Is Scriptable( dt ) ),
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" ),
);
obj = Oneway( Y( :Name("身長 (インチ)") ), X( :年齢 ), Means( 1 ), Mean Diamonds( 1
  ) );
```

データテーブルを開くようユーザに促す

開いているデータテーブルがないときに、データテーブルを開くようユーザに促すには、If() 式を使用します。ユーザがテーブルを開かないときはスクリプトが終了するようにします。次のスクリプトは、その一例です。

```
dt = Current Data Table();
If( Is Empty( dt ),
   Try( dt = Open(), Throw( "データテーブルは開かれませんでした。" ) )
);
```

ユーザは、データテーブルを開くよう促されます。ユーザがデータテーブルを開かずに［**キャンセル**］をクリックした場合、ログにエラーが表示されます。

特定の列だけを表示する

データテーブルを開いて特定の列セットだけを表示するには、Open() 式の中でそれらの列を指定します。これは、大規模なデータテーブルのうち、一部の列だけが必要な場合に有効です。

次の例は「Big Class.jmp」を開き、「**年齢**」、「**身長 (インチ)**」、「**体重 (ポンド)**」の3列のみを含めます。

```
dt = Open( "$SAMPLE_DATA¥Big Class.jmp", Select Columns("年齢", "身長 (インチ)", "
  体重 (ポンド)") );
```

開いているデータテーブルのうち、指定の列だけを除外することもできます。

```
dt = Open( "$SAMPLE_DATA¥Big Class.jmp", Ignore Columns("名前", "性別") );
```

新しいデータテーブルの作成

新しいデータテーブルを作成したり、新しいデータテーブルを作成してその参照をグローバル変数に格納したりできます。どちらの場合も、テーブル名を引数として指定します。

```
New Table("My Table");
```

または

```
dt = New Table("My Table");
```

以下の節で、New Table() 関数のオプションの引数について説明します。

Invisible | Private

Invisibleは、新しいデータテーブルを画面からは隠し、ウィンドウリストには表示するためのオプションのキーワードです。Privateは、新しいデータテーブルを画面とウィンドウリストの両方で隠すためのオプションのキーワードです。

アクション

新しいデータテーブルの定義に使えるオプションの引数です。たとえば、次のスクリプトは、**Little Class** という名前の新しいデータテーブルを作成し、行を3つと、名前のついた列を2つ作成し、各セルに値を入力します。

```
dt = New Table( "Little Class",
   Add Rows(3),
   New Column( "名前",
      character,
      Nominal
      Set Values( {"KATIE", "LOUISE", "JANE"} )
   ),
   New Column( :Name("身長 (インチ)") ),
```

```
        Continuous,
        Set Values( [59, 61, 55] )
    );
);
```

データの読み込み

テキストファイルやMicrosoft Excelファイルは、JMPに読み込む際、データテーブル形式に変換されます。Windowsでは、3文字のファイル拡張子に従ってファイルの種類とファイル内容の解釈方法が特定されます。次に、JMPによって識別されるファイル拡張子の例を挙げます。

```
Open("My Data.xlsx"); // Microsoft Excel ファイル
Open("My Data.txt"); // テキストファイル
Open("$SAMPLE_IMPORT_DATA/Carpoll.xpt"); // SAS 移送ファイル
```

Macintosh上では、Macintoshのタイプおよびクリエータのコード（もしあれば）に基づいて解釈し、2次的に3文字のファイル名拡張子を使います。ファイルをMacintoshに読み込む場合は、あらかじめファイル拡張子を追加することを忘れないでください。

- タイプおよびクリエータコードは、Finderが作成元のアプリケーションに対応する正しいアイコンでファイルを表示できるようにするための情報です。
- 一般アイコン（他のシステムから取得したファイルに表示されているもの）で表示されるファイルにはファイル名拡張子が付いています。

  ```
  Open("My Data.txt", text);
  ```

注：Macintoshでテキストファイルの種類を指定しない場合、テキストファイルは、スクリプトエディタで開きます。

その他に使えるフォーマットには、.csv、.jsl、および.jrnがあります。

テキストファイルからのデータの読み込み

テキストファイルからデータを読み込む場合は、列名や、行の終わりを示す文字が正しく読み込まれるよう、引数を追加することができます。読み込みのオプションについての詳細は、『スクリプト構文リファレンス』を参照してください。

Open()関数の中で指定できる引数には、次のようなものがあります。

```
CharSet("option")
   // "Best Guess"、"utf-8"、"utf-16"、"us-ascii"、"windows-1252"、"x-max-roman"、
   "x-mac-japanese"、"shift-jis"、"euc-jp"、"utf-16be"、"gb2312"
Number of Columns(number)
Columns(colName=colType(colWidth),... )
   // colTypeは Character|Numeric
   // colWidthは列幅を指定する整数
End Of Field (Tab|Space|Comma|Semicolon|Other|None)
EOF Other ("char")
```

```
End Of Line (CRLF|CR|LF|Semicolon|Other)
EOL Other ("char")
Strip Quotes|Strip Enclosing Quotes (Boolean)
Labels | Table Contains Column Headers (Boolean)
Year Rule | Two digit year rule
   ("1900-1999"|"1910-2009"|"1920-2019"|"1930-2029"|"1940-2039"|
   "1950-2049"|"1960-2059"|"1970-2069"|"1980-2079"|"1990-2089"|"2000-2099")
Column Names Start | Column Names are on line (number)
Data Starts | Data starts on line (number)
Lines to Read(number)
引用符としてアポストロフィを使用／Use Apostrophe as Quotation Mark
CompressNumericColumns(Boolean)
CompressCharacterColumns(Boolean)
CompressAllowListCheck(Boolean)
```

次のスクリプトは、テキストがカンマで区切られているテキストファイルを開きます。このファイルには、列名は含まれていません。このスクリプトで、列名と列の幅を定義します。

```
Open("$SAMPLE_IMPORT_DATA/EOF_comma.txt", End of Field(comma), Labels(0),
   Columns(name=character(12), age=numeric(5), sex=character(5),
   height=numeric(3), weight=numeric(3)) );
```

次の例では、フィールドの区切りにスペースが使われているテキストファイルを開きます。

```
Open("$SAMPLE_IMPORT_DATA\EOF_space.txt", Labels(0), End of Field(Other),
   EOFOther(Space));
```

以下の節で、各引数について詳しく説明します。

列数／Number of Columns

ソースファイルの列数を指定します。データが明確に区切られていない場合に重要です。

列／Columns

これまでの例にあったように、Columns引数を使って列名、列の種類、列の幅を指定します。

ファイルの2列目以降に対して設定を行う場合は、それより前にある列もすべて設定する必要があります。たとえば、**「名前」**、**「性別」**、**「年齢」**、**「ID」**という4つの列がこの順番にあるテキストファイルを開きたいとします。**「年齢」**は数値列で、幅を5に設定します。その場合は、**「名前」**と**「性別」**の列についても種類と幅を設定し、同じ順番でリストする必要があります。

```
Columns( 名前 =character(15), 性別 =character(5), 年齢 =numeric(5))
```

後続の列（この例では**「ID」**）に対しては、設定を行う必要はありません。

データが読み込まれたら、列の尺度を使用します。「データと尺度の設定または取得」（304 ページ）を参照してください。

注：次の引数のほとんどは、JMP の環境設定で定義されています。環境設定を上書きしたいときは、次のうちで該当する引数を、読み込みスクリプトの中に含めます。

引用符を取り除く／Strip Quotes | Strip Enclosing Quotes

文字列の値を囲む二重引用符（"）を含めるか、削除するかを指定します。これはブール値です。デフォルト値は、1（真）。

たとえば、フィールドがスペースで区切られているとします。

- 「**John Doe**」は2つの文字列と解釈されます（「**John**」と「**Doe**」）。
- 「"**John Doe**"」は1つの文字列と解釈されます。JMP を始めとする多くのプログラムは、引用符は読みますが、2つ目の引用符が現れるまでにある他の区切り文字は無視します。
- `Strip Quotes(1)` を含めると、「"**John Doe**"」は「**John Doe**」（引用符のない1つの文字列）として解釈されます。

多くのワープロには、二重引用符文字（"）を自動的に左右の（“ ”）に変換する機能があります。テキストファイルの読み込み時に JMP によって二重引用符が取り除かれる場合でも、この形式の二重引用符は、そのまま文字として解釈されます。

行の終わり／End of Line（CR | LF | Semicolon | Other）

行を区切る文字を指定します。次のような選択肢があります。

- CR（復帰文字）Macintosh OS で作成されたテキストファイルで標準的に使用されている区切り文字。
- LF（改行）UNIX と Mac OS X のテキストファイルで標準的に使用されている区切り文字。
- CRLF（復帰文字と改行の両方）Windows のテキストファイルで標準的に使用されている区切り文字。

この3つの文字が、デフォルトで行の区切り文字として設定されています。

行の区切り文字を別の文字にするには、その他（Other）オプションを使用し、`EOLOther` 引数を使って具体的に文字を指定します。JMP は、この文字とデフォルトの文字のいずれかを行の区切り文字と解釈します。

フィールドの終わり／End of Field（Tab | Space | Spaces | Comma | Semicolon | Other | None）

フィールドの区切りに使われる文字を指定します。次の点を念頭に置いてください。

- デフォルトのフィールド区切り文字はタブ（**Tab**）です。
- フィールドの区切り文字を別の文字にするには、その他（`Other`）オプションを使用し、`EOFOther` 引数を使って具体的に文字を指定します。
- `Space` オプションは1つのスペースを区切り文字として使用し、
- `Spaces` オプションは複数のスペースを区切り文字として使用します。

EOF Other、EOL Other

フィールドまたは行の区切り文字を指定します。たとえば、EOLOther("*")は、テキストファイル内の行がアスタリスク（*）で区切られていることを示します。

ラベル | テーブルに列名を含む／Labels | Table Contains Column Headers (Boolean)

テキストファイルの最初の行が、列名として使用されているかどうかを指定します。これはブール値です。デフォルト値は、1（真）。

年表記ルール／2桁の年表記ルール／Year Rule|Two digit year rule

2桁の年の値の読み込み方を指定します。データが対象とする100年間を文字列として入力します。入力できる文字列は、“1900-1999”、“1910-2009”、“1920-2019”、“1930-2029”、“1940-2039”、“1950-2049”、“1960-2059”、“1970-2069”、“1980-2079”、“1990-2089”・・・“2000-2099”です。

列名の開始 | 列名のある行／Column Names Start | Column Names Are on Line

列名の開始行を指定します。

次の例では、テキストファイルの列名が第3行から始まることを指定します。

```
Open(
    "$SAMPLE_IMPORT_DATA/Animals_line3.txt",
    Columns(
        Column( "species", Character, Nominal ),
        Column( "subject", Numeric, Continuous, Format( "Best", 10 ) ),
        Column( "miles", Numeric, Continuous, Format( "Best", 10 ) ),
        Column( "season", Character, Nominal )
    ),
   Column Names Start( 3 )
);
```

データの開始 | データの開始行／Data Starts | Data Starts on Line

データの開始行を指定します。

次の例は、テキストファイルのデータが第5行から始まることを指定します。

```
Open(
   "$SAMPLE_IMPORT_DATA/Bigclass_L.txt",
   Columns(
      Column( "name", Character, Nominal ),
      Column( "age", Numeric, Continuous, Format( "Best", 10 ) ),
      Column( "sex", Character, Nominal ),
      Column( "height", Numeric, Continuous, Format( "Best", 10 ) ),
      Column( "weight", Numeric, Continuous, Format( "Best", 10 ) )
   ),
   Data Starts( 5 )
);
```

読み込む行数／Lines to Read

データテーブルに含める行の数を指定します。JMP は、列名が読み込まれた後でカウントを開始します。

次の例は、最初の 10 行だけをデータテーブルに含めます。

```
Open(
    "$SAMPLE_IMPORT_DATA/Bigclass_L.txt",
    Columns(
        Column( "name", Character, Nominal ),
        Column( "age", Numeric, Continuous, Format( "Best", 10 ) ),
        Column( "sex", Character, Nominal ),
        Column( "height", Numeric, Continuous, Format( "Best", 10 ) ),
        Column( "weight", Numeric, Continuous, Format( "Best", 10 ) )
    ),
   Lines To Read( 10 )
);
```

引用符としてアポストロフィを使用／Use Apostrophe as Quotation Mark

引用符としてアポストロフィを使うことを宣言します。テキストデータが標準的な形式ではなく、フィールドが引用符ではなくアポストロフィで囲まれている場合以外は、このオプションは推奨されません。これはブール値です。

Microsoft Excel ファイルからのデータの読み込み

JMP で Excel ブックを開くと、自動的にデータテーブルに変換されます。JMP は、.xls、.xlsm、.xlsx 形式のファイルに対応しています。Microsoft Excel のサポートについては、『JMP の使用法』を参照してください。

JMP の環境設定では、［一般］にワークシートの読み込みに関する設定があります。

- ［**Excel ファイルを開く方法**］は、標準の Open ステートメントを使って Excel ファイルを開く際の、デフォルトの方法です。
  - ［**Excel ウィザードを使用**］に設定すると、ファイルの読み込み時に Excel 読み込みウィザードが開きます。これがデフォルトの設定です。
  - ［**すべてのシートを開く**］に設定すると、Excel ファイルに含まれるすべてのシートが開きます。
  - ［**個々の Excel シートを選択**］に設定すると、ユーザがファイルを開く際、1 つまたは複数のワークシートを選択するよう促されます。
- ［**Excel のラベルを列名として使用**］は、スプレッドシートの最初の行にあるテキストを、データテーブルの列名に変換するかどうかを指定します。

  デフォルトでは、自動識別が行われます。最初の行のすべてのセルに名前が定義されている場合は、それらのセル内のテキストが列名に変換されます。名前が定義されていないセルがある場合は、「**列 1**」、「**列 2**」・・・という列名が使用されます。

環境設定を上書きしたいときは、以下で説明する対応する引数を JSL スクリプトに含めます。

特定のワークシートの読み込み

ブックにある特定のワークシートからデータを読み込みたいとしましょう。その場合は、Worksheets という引数を使ってワークシートを指定します。次の例は、「小」という名前のワークシートを読み込みます。

```
Open("C:\My Data\cars.xlsx", Worksheets("小"));
```

または、ワークシートの番号を指定します。この例では、3 番目のワークシートを読み込みます。

```
Open("C:\My Data\cars.xlsx", Worksheets("3"));
```

複数またはすべてのワークシートを読み込む場合は、リストの形でワークシート名を指定します。

```
Open("C:\My Data\cars.xlsx", Worksheets( {"小", "中", "大"} ));
```

SAS データセットの読み込み

SAS ファイルを、SAS サーバーに接続せずにデータテーブルとして開きます。

```
sasxpt = Open("$SAMPLE_IMPORT_DATA/carpoll.xpt");
```

ラベルを列名に変換するには、Use Labels for Var Names という引数を使用します。

```
sasdbf = Open("$SAMPLE_IMPORT_DATA/Bigclass.sas7bdat", Use Labels for Var
  Names(1));
```

.xpt および .stx 形式のファイルも使用できます。

Windows では、SAS サーバーに接続して SAS データを開くこともできます。詳細は、「JMP の拡張」の章の「SAS Metadata Server への接続」（558 ページ）の節を参照してください。

Web ページおよびリモートファイルの読み込み

Web サイトや別のコンピュータからデータを読み込むことができます。JMP データテーブル、Web ページで定義されているテーブル、JMP でサポートされている他の種類のファイルを開くことができます。

Web サイト上のデータを開く、または読み込む

Web サイトにあるファイルを開くには、Open() コマンドの中で URL を指定します。その際、URL を引用符で囲んでください。JMP のデータテーブルほか、JMP で使用できる種類のファイルはこの方法で開くことができます。

```
Open("http://company1.com/Repairs.jmp");
Open("http://company1.com/My Data.txt", text); // Macintosh 上のテキストファイルを指定
```

Web ページの読み込み

Web ページには、表形式のデータが含まれていることがあります。次の例では、表を JMP データテーブルとして読み込みます。

```
Open("http://company1.com", HTML Table(n));
```

*n*は、読み込みたい表を表します。たとえば、ページ上にある4つ目の表を読み込むには、HTML Table(4)と指定します。値を省略した場合、ページ上の最初の表だけが読み込まれます。

JMPは、HTMLタグの<th>で定義されているテーブル見出しを保持しようとします。これらの見出しは、データテーブルの列名に変換されます。

共有コンピュータからのファイルの読み込み

JMPでは、共有している別のコンピュータやネットワークドライブに保存されたファイルを読み込むことができます。ファイルパスとしては、絶対パスと相対パスの両方が使えます。次の例では、**Data**という名前の共有コンピュータにあるファイルを開きます。スクリプトを共有する予定がある場合は、マップしたドライブではなく、コンピュータへの相対パスを使用するようにしてください。

```
Open("\\Data\Repairs.jmp");
Open("\\Data\My Data.txt");
```

ESRIシェープファイルの読み込み

ESRIシェープファイルは、地図の作成に使用される地理空間ベクトルデータの一形式です。JMPは、シェープファイルをデータテーブルとして読み込みます。.shpシェープファイルは、各シェープの座標で構成されます。.dbfシェープファイルには、地域を表す値が含まれます。JMPで地図を作成するには、データの構造を調整し、ファイルに専用の接尾辞をつけて保存します。

次の例は、.shpファイルを読み込み、-XYという接尾辞をつけて保存します。

```
dt = Open("$SAMPLE_IMPORT_DATA/Parishes.shp",
:X << Format("Longitude DDD",14,4);
:Y << Format("Latitude DDD", 14, 4) );
dt << Save("c:/Parishes-XY.jmp");
```

.dbfファイルを-Nameという接尾辞をつけて保存します。

```
dt = Open("$SAMPLE_IMPORT_DATA/Parishes.dbf");
dt << Save("c:/Parishes-Name.jmp");
```

データの構造を調整するには、-Name.jmpファイル内の名前に「地図の役割」列プロパティを追加するなど、いくつかの手順が必要となります。詳細は、『グラフ機能』を参照してください。

パスワードで保護されたExcelファイルの読み込み

パスワードで保護された.xlsファイルを読み込むときは、Passwordという引数を含めれば、スクリプトを実行するたびにパスワードを入力する必要がなくなります。

```
Open("Housing.xls", Password( "helloworld" );
```

注：JMPは、パスワードで保護された.xlsxファイルの読み込みをサポートしていません。JMPで読み込めるのは、パスワードで保護されたExcel 2007の.xlsファイルのみです。Excelバージョン2010およびバージョン2013のファイルは、.xlsで保存した場合でも読み込めません。

データベースの読み込み

Open Database() は、ODBC（Open Database Connectivity）を使ってデータベースを開き、データを抽出し、JMP データテーブルに読み込みます。詳細については、「JMP の拡張」の章の「データベースアクセス」（553 ページ）を参照してください。

JMP は、データベースファイル（.dbf）ファイルもデータテーブル形式に変換します。

```
sasdbf = Open("$SAMPLE_IMPORT_DATA/Bigclass.dbf", Use Labels for Var Names);
```

現在のデータテーブルの設定

ヒント：目的のデータテーブルが現在のデータテーブルになっているかどうかの判断には注意が必要です。スクリプトの冒頭で現在のデータテーブルに設定した場合でも、途中のアクションによって状況が変わっている場合があります。

次のように処理されたデータテーブルは、現在のデータテーブルになります。

- 次の方法で開いたデータテーブル

  ```
  dt1 = Open(“$SAMPLE_DATA/Big Class.jmp”);
  ```

- 次の方法で新しく作成したデータテーブル

  ```
  dt2 = New Table(“Cities”);
  ```

開いている別のデータテーブルを現在のデータテーブルとするには、Current Data Table() コマンドを使い、データテーブル名を指定します。

```
dt1 = Open("$SAMPLE_DATA/Big Class.jmp");
dt2 = New Table("Cities");
Current Data Table(dt1); //Big Class.jmp データテーブルを現在のデータテーブルにする
```

Current Data Table() も、スクリプト可能なオブジェクトの参照を引数としてとることができます。次の式は、2 番目の Bivariate オブジェクトによって使用されているデータテーブルを現在のデータテーブルにします。

```
Current Data Table(Bivariate[2]);
```

分析プラットフォームのオブジェクトの参照を使用する方法については、「プラットフォームのスクリプト」の章の「現在行われている分析にスクリプトコマンドを送る」（357 ページ）を参照してください。

データテーブルに名前をつける

データテーブルに名前をつけるには、Set Name メッセージを送ります。引数には、引用符で囲んだファイル名か、ファイル名を評価する式を指定します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Set Name("New Big Class.jmp");

s = "New Big Class";
dt << Set Name(s);
```

名前を取得するには、データテーブルに Get Name メッセージを送ります。

```
dt << Get Name();
   "New Big Class"
```

データテーブルの保存

データテーブルを保存するには、データテーブルに Save メッセージを送ります。たとえば、次の場合が該当します。

```
dt << Save(); // 現在の名前を使用して保存する
dt << Save("Newest Big Class.jmp") // 新しいファイルとして保存する
dt << Save("c:/My Data/New Big Class.jmp");  // 新しいファイルとして保存する
dt << Save("My Table", JMP(5)); // JMP 5 のテーブルとして保存する
dt << Save("") // ディレクトリを選択して任意の形式で保存するようユーザを促す
dt << Save("Big Class.xls"); // Microsoft Excel ファイルとして保存する
```

注：パスを使わずにファイル名を指定し、デフォルトのディレクトリを設定していない場合、ファイルは、プライマリパーティション（Windows の場合）または <*ユーザ名*>/Documents フォルダ（Macintosh の場合）に保存されます。デフォルトのディレクトリを設定する方法については、「データタイプ」の章の「相対パス」（122 ページ）を参照してください。

Windows では、拡張子 .txt を付けて保存すると、環境設定の［テキストデータファイル］で設定されている方法で書き出されます。Macintosh では、次のように Save 関数の 2 番目の引数として Text を追加します。

```
dt << Save("New Big Class.txt", Text);
```

データテーブルの名前を設定し、後で Save メッセージを送る予定がある場合は、Save メッセージだけを使って名前を指定することができます。

```
dt << Set Name("New Big Class.jmp");
dt << Save();
```

上の組み合わせと、次のメッセージは等価です。

```
dt << Save("New Big Class.jmp");
```

Save とパス名を含める方法も、Save As とパス名を使う方法と等価です。

データテーブルを保存されている状態に戻す

最後に保存した状態のデータテーブルに戻すには、Revert メッセージをデータテーブルに送ります。

```
dt << Revert();
```

データテーブルを非表示にする

ユーザにデータテーブルを見せる必要がない場合は、次の2つの方法でデータテーブルを非表示にしておくことができます。

- **invisible** と指定したデータテーブルは、表示はされませんが、JMP のインターフェースを通じて開くことができます。Windows の場合、invisible のデータテーブルは、ホームウィンドウのウィンドウリストと［**ウィンドウ**］>［**再表示**］メニューに表示されます。Macintosh の場合は、［**ウィンドウ**］>［**再表示**］メニューに表示されます。
- **private** と指定したデータテーブルは、完全に非表示になります。

データテーブルを非表示にして開くには、`invisible` という引数を含めます。

```
dt = Open( "$SAMPLE_DATA¥Big Class.jmp", invisible );
```

次の例は、「Abc」という名前のデータテーブルを作成し、「X」という名前の列を1つ、行を10行含め、非表示にします。

```
dt = NewTable("Abc", invisible, newColumn("X"),addRows(10));
```

開いたデータテーブルが非表示かどうかを調べるには、データテーブルオブジェクトにブール値の `Has Data View` メッセージを送ります。データテーブルが表示されている場合、この式は 0（真）を戻します。

```
dt << Has Data View();
```

非表示になっているデータテーブルが失われるのを防ぐため、これから紹介する例を参考に、必ずデータテーブルに参照を割り当ててください。参照を割り当てずに非表示のデータテーブルを作成すると、JMP によってただちにメモリから削除されてしまいます。その場合、スクリプトの後半でそのテーブルを使おうとしても、エラーになります。

注：非表示のデータテーブルでの作業を終えたら、必ずデータテーブルを閉じてください。そうしないと、JMP の終了時までメモリに保持されます。

データテーブルを完全に非表示にするには、`private` という引数を使用します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", private);
```

注：なお、非表示のデータテーブルは、明示的に閉じるまでメモリ内にとどまるため、不要になったものは閉じるよう注意してください。非表示のデータテーブルは、表示もされなければ、分析にリンクされることもありません。

データテーブルの印刷

データテーブルを印刷するには、`Print Window` メッセージを送ります。JMP は、印刷の際、コンピュータのデフォルトの設定を使用します。

```
dt << Print Window();
```

このメッセージは、他のディスプレイボックスにも適用できます。

データテーブルのサイズ変更

dt << Maximize Displayは、すべての列のサイズを再調整して、データテーブルを最適なサイズのウィンドウに拡大します。

```
Current Data Table() << Maximize Display;
```

データテーブルを閉じる

データテーブルを閉じるには、引数にデータテーブルの参照を指定して、Closeコマンドを使います。データテーブルに加えられた変更（データテーブルのリンクを含む）がまだ保存されていない場合、その変更は自動的に保存されます。そのデータテーブルを元に作成されたレポートやグラフも同時に閉じられます。

変更を保存してデータテーブルを閉じるには:

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
Close(dt);
```

変更を保存せずにデータテーブルを閉じるには:

```
Close(dt, No Save);
```

また、すべてのデータテーブルを同時に閉じることもできます。その場合も、変更を保存するかしないかを指定します。

```
Close All(Data Tables);
Close All(Data Tables, No Save);
```

ヒント：Close All() は、ジャーナルやレイアウトにも使用できます。その場合は、引数で "Journals" または "Layouts" を指定します。

コピーを新しい名前で保存し、元のバージョンを閉じるには:

```
Close(dt, Save("c:/My File.jmp"));
```

データテーブルの設定と取得

New Data Box()を使って作成されたDataBrowserBoxを使用する場合、Set Data Table()およびGet Data Table()を使ってデータテーブルを設定または戻します。

次の例は、「Big Class.jmp」と「cities.jmp」サンプルデータテーブルを使用します。まず、これらのサンプルデータテーブルが非表示で開きます。次に、新しいウィンドウに、「Big Class.jmp」を含む新しいデータボックスが作成されます。1秒待機した後、データボックスの内容が「cities.jmp」の表示に変わります。

```
dtC = Open( "$SAMPLE_DATA/cities.jmp", invisible );
dtA = Open( "$SAMPLE_DATA/Big Class.jmp", invisible );
nw = New Window( "My Table", H List Box( dtBox = dtA << New Data Box() ) );
Wait( 1.0 );
dtBox << Set Data Table( Data Table( "Cities.jmp" ) );
```

開いているすべてのデータテーブルに対するアクションの実行

現在開いているデータデーブルすべてに対してアクションを行う場合は、N Table を使用して各データテーブルへの参照リストを取得できます。

```
openDTs = List();
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) );
);
```

これで、openDTs は、開いているすべてのデータテーブルへの参照リストとなります。openDTs[n] を使用すると、任意のデータテーブルにメッセージを送ることができます。開いているデータテーブルすべてにメッセージを送るには、for ループを使用できます。次のループでは、各データテーブルに My Column という名前の新しい列を追加します。

```
For( i = 1, i <= N Items(openDTs), i++,
   openDTs[i] << New Column("My Column");
);
```

データテーブルへの参照ではなく、テーブル名のリストだけが必要な場合は、Get Name メッセージを使用します。

```
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) << Get Name());
   );
```

取得したリストを使い、開いているデータテーブルすべての名前をリストボックス（ダイアログ）に含めれば、ユーザがデータテーブルの操作を行うときに、その中から対象となるデータテーブルを選択することができます。また、リストにあるデータテーブル名をファイルに書き出すこともできます。

ジャーナルとレイアウトの作成

ジャーナルは、JMP グラフやレポート、グラフィック、テキスト、Web ページやファイルなどへのリンクで構成されます。さまざまな要素をジャーナルにまとめることで、プレゼンテーションなどの機会に繰り返し利用することが可能になるほか、他のドキュメントへの読み込みも簡単になります。

レイアウトは、複数のレポートを組み合わせたファイルで、保存する前にレポートの配置を調整することができます。

ジャーナルとレイアウトの基本的な構文は、次のようになります。

```
Current Data Table() << Journal;
Current Data Table() << Layout;
```

また、New Window() コマンド内に加えることにより、新しいジャーナルウィンドウを作成します。次の例は、空白で無題のジャーナルとレイアウトを作成します。

```
New Window( <<  Journal);
New Window( <<  Layout);
```

次の例は、「売り上げ」という名前で空白のジャーナルとレイアウトを作成します。

```
New Window("売り上げ", << Journal);
New Window("売り上げ", << Layout);
```

ここで、2つのボタンを含む、「ボタンのテスト」というジャーナルを作成する例を紹介します。

```
New Window( "ボタンのテスト", << Journal,
   Button Box( "テスト 1", New Window( "こんにちは 1", << Modal ) ),
   Button Box( "テスト 2", New Window( "こんにちは 2", << Modal ) )
);
```

ジャーナルの取得

ジャーナルとジャーナルウィンドウのコマンドには、オプションの引数があります。この引数は、現在の表示を取得する（フリーズさせる）ものです。つまり、表示ツリーを 1 つのイメージに変換します。

引数には次の4つのオプションがあります。

- << Journal (Freeze All) // 編集可能なディスプレイボックスの構造をコピーするのではなく、その時点での表示の「スナップショット」をビットマップ形式で保存し、ジャーナルに送る 表示をビットマップファイルとして保存することで、グラフ内でスクリプトの変数が参照されていても、問題が生じにくくなります。
- << Journal(Freeze Pictures) //Freeze All と同様だが、レポートのうち、ピクチャーと呼ばれる領域のみをフリーズさせる
- << Journal(Freeze Frames) ////Freeze Pictures と同様だが、（ピクチャーの内側にある）フレームボックス内のみをフリーズさせる
- << Journal(Freeze Frames with Scripts) //Freeze Frames と同様だが、描画するスクリプトがあるフレームボックスのみをフリーズさせる

Current Journal

Current Journal() は、現在のジャーナル表示ウィンドウの最上位のディスプレイボックスへの参照を戻します。ジャーナルが開かれていない場合は、1つ作成されます。引数はありません。

Append コマンドを使ってジャーナルに追加できます。次の例は、現在のジャーナルの下にテキストボックスを追加します。

```
Current Journal() <<  Append(Text Box("Hello World"));
```

また、文字列を角括弧（[]）で囲んで追加すると、既存のジャーナル内の該当文字列が検索されます。現在のジャーナルに「パラメータ推定値」という文字列が含まれているとします。次の例は、そのジャーナルの下にテキストボックスを追加します。

```
Current Journal()["パラメータ推定値"] << Append(Text Box("アスタリスクは有意確率が 0.05
   の項目を示す。"));
```

データテーブルの高度なスクリプト

この節では、データテーブルに対して実行できる、要約統計量の収集、サブセットの作成、並べ替え、連結などの高度なアクションについて説明します。

要約統計量をグローバル変数に格納する

Summarizeコマンドは、データテーブルの要約統計量を求め、グローバル変数に格納します。Summarizeコマンドは、要約統計量を求めて新しいデータテーブルに表示するSummaryコマンドとは異なります。

名前付き引数は、Count、Sum、Mean、Min、Max、StdDev、Quantileで、このそれぞれがデータ列を引数にとります。

次の点を念頭に置いてください。

- *name*=By(*groupvar*) ステートメントが含まれている場合は、サブグループの統計量が *name* という名前のリストに割り当てられます。
- Countの場合、列引数は必須ではありませんが、列を指定して非欠測値の数をカウントすると役に立つことがよくあります。
- Quantileは、どの分位点かを指定する第2引数もとります（たとえば、10パーセント点なら0.1）。

注：除外された行は、Summarizeの計算から除外されます。すべてのデータが除外された場合、Summarizeは欠測値のリストを戻します。すべてのデータが削除された（行がない）場合、Summarizeは空のリストを戻します。

次の例は、「Big Class.jmp」サンプルデータテーブルを使用します。

```
Summarize(
   a = by( 年齢 ),
   c = count,
   sumHt = Sum( Name("身長 (インチ)") ),
   meanHt = Mean( Name("身長 (インチ)") ),
   minHt = Min( Name("身長 (インチ)") ),
   maxHt = Max( Name("身長 (インチ)") ),
   sdHt = Std Dev( Name("身長 (インチ)") ),
   q10Ht = Quantile( Name("身長 (インチ)"), .10 )
);
Show(a, c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht);
```

スクリプトにByグループが含まれているので、結果は1つのリストと6つの行列になります。

```
a = {"12", "13", "14", "15", "16", "17"}
c = [8, 7, 12, 7, 3, 3]
c = [465, 422, 770, 452, 193, 200]
meanHt = [58.125, 60.28571428571428, 64.16666666666667, 64.57142857142857,
   64.33333333333333, 66.66666666666667]
```

```
c = [51, 56, 61, 62, 60, 62]
c = [66, 65, 69, 67, 68, 70]
sdHt = [5.083235752381126, 3.039423504234876, 2.367712103711172,
   1.988059594776032, 4.041451884327343, 4.163331998932229]
q10Ht = [51, 56, 61.3, 62, 60, 62]
```

TableBoxを使って結果をフォーマットすることもできます。詳細は、「表示ツリー」の章の「独自の表示を始めから作成する」（426ページ）を参照してください。

```
New Window( "要約の結果",
   Table Box(
      String Col Box( "年齢", a ),
      Number Col Box( "度数", c ),
      Number Col Box( "合計", sumHt ),
      Number Col Box( "平均", meanHt ),
      Number Col Box( "最小", minHt ),
      Number Col Box( "最高", maxHt ),
      Number Col Box( "標準", sdHt ),
      Number Col Box( "10パーセント点", q10Ht )
   );
);
```

図9.1 要約の結果

年齢	度数	合計	平均	最小	最高	標準	10パーセント点
12	8	465	58.125	51	66	5.08324	51
13	7	422	60.2857	56	65	3.03942	56
14	12	770	64.1667	61	69	2.36771	61.3
15	7	452	64.5714	62	67	1.98806	62
16	3	193	64.3333	60	68	4.04145	60
17	3	200	66.6667	62	70	4.16333	62

次のように、ウィンドウに合計を含めることができます。

```
Summarize(
   tc = count,
   tsumHt = Sum( Name("身長（インチ）") ),
   tmeanHt = Mean( Name("身長（インチ）") ),
   tminHt = Min( Name("身長（インチ）") ),
   tmaxHt = Max( Name("身長（インチ）") ),
   tsdHt = Std Dev( Name("身長（インチ）") ),
   tq10Ht = Quantile( Name("身長（インチ）"), .10 )
);

Insert Into( a, "合計" );
c = c |/ tc;
sumHt = sumHt |/ tsumHt;
meanHt = meanHt |/ tmeanHt;
```

```
minHt = minHt |/ tminHt;
maxHt = maxHt |/ tmaxHt;
sdHt = sdHt |/ tsdHt;
q10Ht = q10Ht |/ tq10Ht;

New Window( "要約の結果",
   Table Box(
      String Col Box( "年齢", a ),
      Number Col Box( "度数", c ),
      Number Col Box( "合計", sumHt ),
      Number Col Box( "平均", meanHt ),
      Number Col Box( "最小", minHt ),
      Number Col Box( "最高", maxHt ),
      Number Col Box( "標準", sdHt ),
      Number Col Box( "10パーセント点", q10Ht )
   );
);
```

図9.2 要約と全体の統計量

年齢	度数	合計	平均	最小	最高	標準	10パーセント点
12	8	465	58.125	51	66	5.08324	51
13	7	422	60.2857	56	65	3.03942	56
14	12	770	64.1667	61	69	2.36771	61.3
15	7	452	64.5714	62	67	1.98806	62
16	3	193	64.3333	60	68	4.04145	60
17	3	200	66.6667	62	70	4.16333	62
合計	40	2502	62.55	51	70	4.24234	56.2

By グループを**指定しない**場合、各名前の結果は、次のように単一の値になります。

```
Summarize(
// a=by(年齢),
   c = count,
   sumHt = Sum( Name("身長(インチ)") ),
   meanHt = Mean( Name("身長(インチ)") ),
   minHt = Min( Name("身長(インチ)") ),
   maxHt = Max( Name("身長(インチ)") ),
   sdHt = Std Dev( Name("身長(インチ)") ),
   q10Ht = Quantile( Name("身長(インチ)"), .10 )
);
Show( c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht );
   c = 40;
   sumHt = 2502;
   meanHt = 62.55;
   minHt = 51;
   maxHt = 70;
```

```
sdHt = 4.24233849397192;
q10Ht = 56.2;
```

Summarize では、複数の By グループを使用できます。たとえば、「Big Class.jmp」では、次のように指定できます。

```
Summarize(g=by(年齢 , 性別 ), c=count());
show(g, c);
   g = {"12", "12", "13", "13", "14", "14", "15", "15", "16", "16", "17", "17"},
      {"F", "M", "F", "M", "F", "M", "F", "M", "F", "M", "F", "M"}}
   c = [5,3,3,4,5,7,2,5,2,1,1,2]
```

By グループを指定すると、結果は常に行列になります。指定しない場合、結果はスカラーになります。

要約統計量の表を作成する

Summary コマンドは、ユーザが選択したグループ化列に基づいて、要約統計量から成る新しいテーブルを作成します。Summary は、データテーブルの要約統計量を求めてグローバル変数に格納する Summarize とは異なるので、注意が必要です。詳細は、「要約統計量をグローバル変数に格納する」（277 ページ）の節を参照してください。

```
summDt = dt << Summary(
   Group(groupingColumns),
   Subgroup(subGroupColumn),
   Statistic(columns),// 統計量は、Mean( 平均 )、Min( 最小値 )、Max( 最大値 )、Std Dev( 標準偏差 ) など
   Output Table Name(newName));
```

次の例は、年齢別の平均身長と平均体重の列、年齢別の最大身長と最小体重の列を含めた新しいデータテーブルを作成します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
summDt = dt << Summary(
   Group( 年齢 ),
   Mean(:Name(" 身長 ( インチ )"),:Name(" 体重 ( ポンド )")), Max(:Name(" 身長 ( インチ )")),
   Min(:Name(" 体重 ( ポンド )")),
   output table name(" 身長・体重の表 "));
```

ヒント：Output Table Name は、引用符付き文字列、または文字列を含んだ変数をとることができます。

デフォルトでは、要約テーブルは元のデータテーブルにリンクしています。元のデータテーブルとリンクしていない要約を作成したいときは、Summary メッセージに次のオプションを追加します。

```
summDt = dt << Summary( Group( :年齢 ), Mean( :Name(" 身長 ( インチ )") ),
   Link to original data table( 0 )
);
```

データテーブルのサブセットを作成する

Subsetは、指定された行から新しいデータテーブルを作成します。行を指定しない場合、Subsetは選択されている行を使います。行が選択も指定もされていない場合は、すべての行を使います。列が指定されていない場合は、すべての列を使います。Subsetに引数を指定しなかった場合、「サブセット」ウィンドウが表示されます。

```
dt << Subset(
   Columns(columns),
   Rows(row matrix),
   Linked,
   Output Table Name("name"),
   Copy Formula (1 or 0),
   Sampling rate (n),
   Suppress Formula Evaluation(1 or 0));
```

注：その他の引数については、［スクリプトの索引］を参照してください。

たとえば、Big Class.jmpのうち、年齢が「12」であるすべての行を選択するには：

```
For Each Row(Selected(Rowstate())=(年齢==12));
subdt = dt << Subset(output table name("サブセット"));
```

3つの列とすべての行を抽出するには：

```
subDt1 = dt << Subset (Columns(名前,年齢,:Name("身長(インチ)")), Output Table
   Name("Big Class NAH"));
```

2つの列と行を抽出し、元のテーブルとリンクさせるには：

```
subDt2 = dt << Subset (Columns(名前,:Name("体重(ポンド
   )")),Rows([2,4,6,8]),Linked);
```

年齢が12歳のすべての行の列を選択するには：

```
dt << Select Where(age==12);
dt << Subset(( Selected Rows ), Output Table Name("サブセット"));
```

Data Filterを使ってデータのサブセットを作成する

Data Filterは、データをサブセットしながら探索するためのさまざまな方法を提供します。Data Filterのコマンドとオプションを使えば、データの複雑なサブセットを選択し、それをプロット内で非表示にしたり、分析から除外したりできます。

基本的な構文は次のとおりです。

```
dt << Data Filter( <local>, <invisible>, <Add Filter (...)>, <Mode>, <Show Window(
   0 | 1 ), <No Outline Box( 0 | 1 )> );
```

Data Filterのオプションには、次のようなものがあります。

Add Filter、Animation、Auto Clear、Clear、Close、Columns、Conditional、Data Table Window、Delete、Delete All、Display、Get Filtered Rows、Location、Match、Mode、Report、Save and Restore Current Row States、Set Select、Set Show、Set Include、Show Column Selector、Show Subset、Use Floating Window

注：その他の引数については、［スクリプトの索引］を参照してください。

オプションなしでData Filterメッセージをデータテーブルに送ると、Data Filterの開始ウィンドウが表示され、「フィルタ列の追加」パネルにデータテーブル内の変数がリストされます。

Modeは、Select(bool)、Show(bool)、Include(bool)というオプションの引数をとります。これらの引数は、それぞれ対応するオプションをオンまたはオフにします。Selectのデフォルト値はtrue（1）です。ShowとIncludeのデフォルト値はfalse（0）です。

Add Filterは、列とデータテーブルをサブセットするWhere節を指定します。基本的な構文は次のとおりです。

```
Add Filter( columns( col, ... ), Where( ... ), ... )
```

データフィルタに列を追加するには、列名をカンマで区切って指定します。これは、リストデータ構造ではありません。

フィルタ列を指定する場合は、1つまたは複数のWhere節を次のように定義します。

```
df = dt << Data Filter(
   Mode( Show( 1 ) ),
   Add Filter(
      columns( :年齢, :性別, :Name("身長(インチ)") ),
      Where( :年齢 == {13, 14, 15} ),
      Where( :性別 == "M" ),
      Where( :Name("身長(インチ)") >= 50 & :Name("身長(インチ)") <= 65 )
   );
);
```

このスクリプトは、身長が50インチ以上65インチ以下の13歳、14歳、15歳の男子を選択します。前述のスクリプトにInvert Selectionメッセージを追加すると、このフィルタされたデータで除外されている年齢を選択できます。

```
df << (Filter Column(:年齢) << Invert Selection);
```

この例では、フィルタされたデータの13歳、14歳、15歳以外の年齢が選択されます。

Add Filterを使って、多重応答プロパティを持った列から一致する文字列を選択することもできます。

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
df = dt << Data Filter(
   Location( {437, 194} ),
   Add Filter(
```

```
        columns( :歯磨き カンマ区切り ),
        Match None( Where( :歯磨き カンマ区切り == {"Before Sleep", "Wake"} ) ),
        Display( :歯磨き カンマ区切り , Size( 121, 70 ), Check Box Display )
    )
);
```

このスクリプトは、**「歯磨き カンマ区切り」**列内で、指定された値（「Before Sleep」、「Wake」）のいずれにも一致しない行を選択します。使用可能なその他のスクリプトオプションには、Match Any、Match All、Match Exactly、Match Onlyがあります。多重応答プロパティのオプションの詳細については、『JMPの使用法』を参照してください。

既存のData Filterオブジェクトにメッセージを送ることもできます。

```
Clear(), Display( ... ), Animate(), Mode(), ...
```

Clearは、引数をとらず、データフィルタをクリアします。

スクリプトの実行時にデータフィルタが表示されないようにするには、次のようにShow Windowを0に設定します。

```
obj  = ( Current Data Table() << Data Filter( Show Window(0) ) );
```

データフィルタのコンテキストを定義する

データフィルタを使うと、データの複雑なサブセットを選択し、それをプロット内で非表示にしたり、分析から除外したりできます。選択の範囲は、データテーブルのすべての分析に影響します。

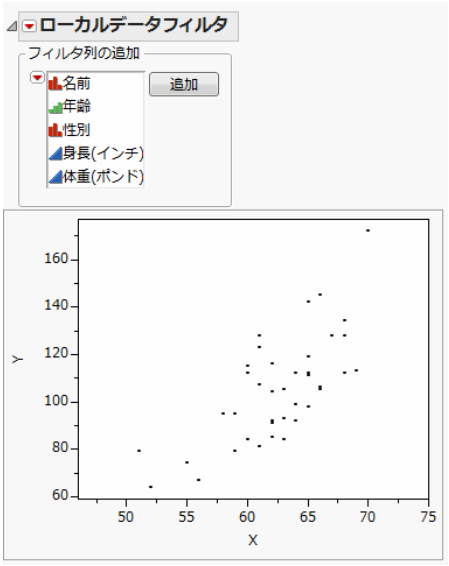
別のオプションとしては、特定のプラットフォームまたはディスプレイボックスからデータをフィルタする方法があります。Data Filter Context Box()関数の中にローカルデータフィルタを作成します。これにより、コンテキストはデータテーブルとしてではなく、現在のプラットフォームまたはディスプレイボックスとして定義されます。

次の例は、グラフボックス用のローカルデータフィルタを作成します。出力については、図9.4を参照してください。

```
New Window( "マーカーセグメントの例",
    Data Filter Context Box(
        V List Box(
            dt << Data Filter( Local ),
            g = Graph Box(
                Frame Size( 300, 240 ),
                X Scale( Min( xx ) - 5, Max( xx ) + 5 ),
                Y Scale( Min( yy ) - 5, Max( yy ) + 5 ),
                Marker Seg( xx, yy, Row States( dt, rows ) )
            );
        );
    );
);
```

ヒント：上の結果を得るには、「Local Data Filter for Custom Graph.jsl」サンプルスクリプトを開きます。

図9.3 ローカルデータフィルタとグラフ

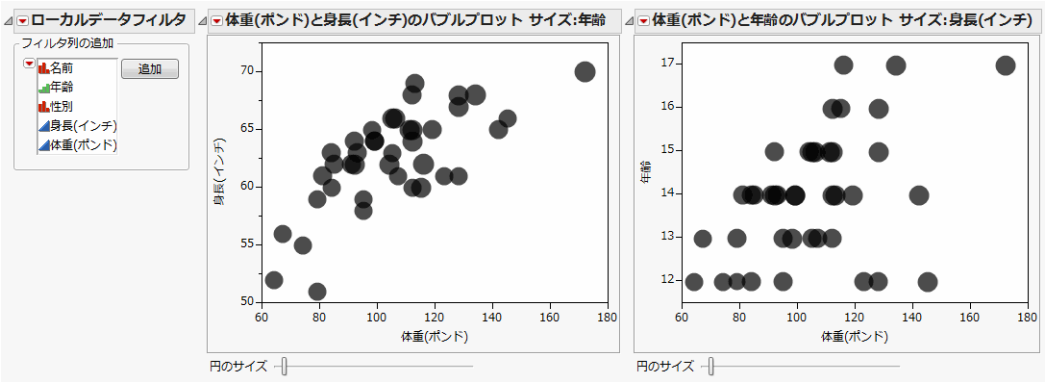


1つのコンテキストボックスに、1つのローカルフィルタと複数のグラフを含めることもできます。フィルタリングはすべてのグラフに適用されます。次のスクリプトは、1つのコンテキストボックスにバブルプロットを2つ、データフィルタを1つ作成します。出力については、図9.4を参照してください。

```
Data Filter Context Box(
   H List Box(
      dt << Data Filter( Local ),
      Platform(
         Current Data Table(),
         Bubble Plot( X( :Name("体重(ポンド)") ), Y( :Name("身長(インチ)") ),
   Sizes( :年齢 ) )
      ),
      Platform(
            Current Data Table(),
         Bubble Plot( X( :Name("体重(ポンド)") ), Y( :年齢 ), Sizes( :Name("身長(
   インチ)") ) )
      );
   );
);
```

ヒント：このスクリプトを実行するには、「Local Data Filter Shared.jsl」サンプルスクリプトを開きます。

図9.4 2つのバブルプロットを使ったローカルフィルタ

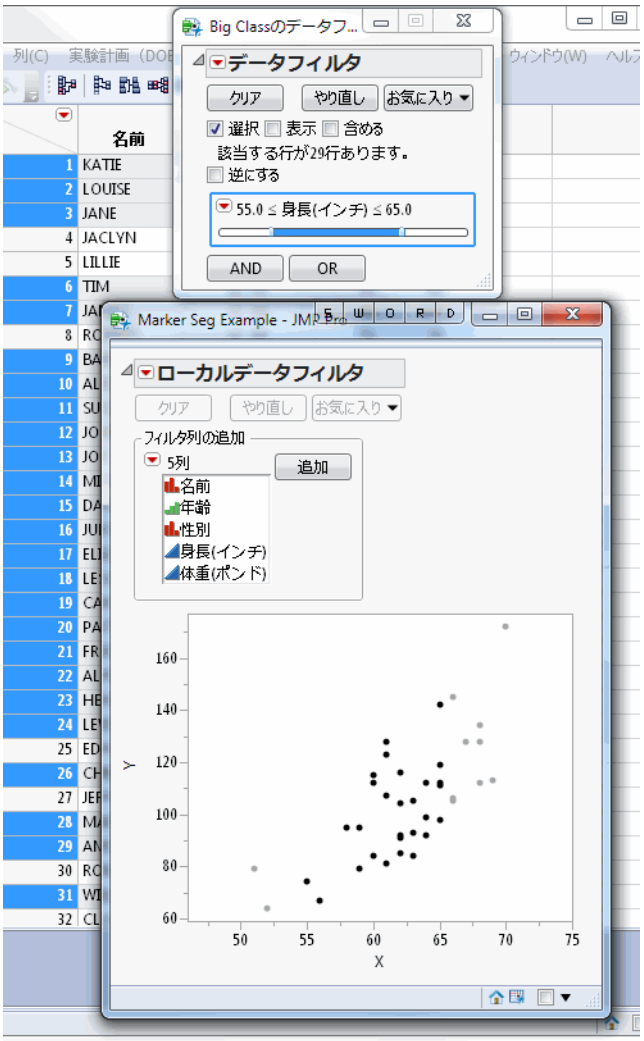


階層的なデータフィルタを作成することもできます。1つのスクリプトでデータフィルタとローカルデータフィルタが生成される場合があります。データテーブルの外側のデータフィルタが、ローカルデータフィルタに使用できるデータを決定します。これをわかりやすく示すため、例を紹介します。次のスクリプトは、図9.5に示すように両方のタイプのデータフィルタを作成します。

```
dt << Data Filter(
   Add Filter( Columns( :Name("身長(インチ)") ), Where( :Name("身長(インチ)") >=
   55 & :Name("身長(インチ)") <= 65 ) )
);
// ローカルデータフィルタ
New Window( "マーカーセグメントの例",
   Data Filter Context Box(
      V List Box(
         dt << Data Filter( Local ),
         g = Graph Box(
            Frame Size( 300, 240 ),
            X Scale( Min( xx ) - 5, Max( xx ) + 5 ),
            Y Scale( Min( yy ) - 5, Max( yy ) + 5 ),
            Marker Seg( xx, yy, Row States( dt, rows ) )
         );
      );
   );
);
```

55インチ以上65インチ以下の身長は最初にフィルタされます。その後、ユーザはローカルデータフィルタを使ってグラフの列をフィルタできます。

図9.5 データフィルタの階層



データテーブルを並べ替える

Sortは、1つ以上の列の値に基づいてテーブルの行を並べ替え、その結果で現在のテーブルを置き換えるか、または新しいテーブルを作成します。それぞれのBy列に対し、並べ替えの順序（昇順または降順）を指定します。

```
dt << Sort(
   Private, Invisible, Replace table, By(columns), Order(Descending | Ascending)
  );
```

次の例は、Big Class.jmp を元に新しくデータテーブルを作成し、年齢と名前に基づいてデータを降順に並べます。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
sortedDt = dt << Sort(
   By( 年齢 , 名前 ),
   Order(Descending, Ascending),
   output table name(" 年齢  名前で並べ替え "));
```

連想配列を使って列内の値を並べ替えることもできます。連想配列を使うと、列内の固有の値を簡単に見つけられます。詳細は、「データ構造」の章の「列の値を文字コード順に並べ替える」（194 ページ）を参照してください。

データテーブル内の値を積み重ねる

Stack は、複数の列の値を 1 つの列に積み重ねます。

```
dt << Stack(
   columns (columns),  // ひとつに積み重ねる列
   Source Label Column ("name"), // ソース列の識別
   Stacked Data Column ("name"), // 新しく積み重ねた列の名前
   Keep (columns), // データテーブルに保持する列
   Drop (columns), // データテーブルから除去する列
   Output Table ("name"), // 新しいデータテーブルの名前
   Columns(columns)); // 積み重ねたテーブルに含める列を指定
```

たとえば、*dt* を「Big Class」への参照とすると、以下のようになります。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Stack(
    columns( :Name(" 体重 ( ポンド )"), :Name(" 身長 ( インチ )")),
    Source Label Column( "ID" ),
    Stacked Data Column( "Y" ),
    Name( " 積み重ねていない列 " )(Keep( : 年齢 , : 性別 )),
    Output Table( " 積み重ねた列 " );
```

Columns(columns) 引数には、列のリスト、またはリストを導き出す式を指定できます。

積み重ねた後のデータテーブルで値を分割する

Split は、積み重ねられた列を複数の列に分割します。

```
dt << Split(
   Split(columns), // 分割する列
   Split by(columns) // 分割の基準となる列
   Group(columns), //  （オプション）行を一意に識別する列
   ColID(columns),  // 分割する基準となる変数
   Remaining Columns( Keep All | Drop All | Drop(columns) | Keep(columns)),
```

```
Sort by Value Order //「値の順序」列プロパティを使って出力を並べ替える
Output Table ("name"));
```

オプションの Remaining Columns（残りの列）引数は、新しいデータテーブルに入れるソースデータテーブルの他の列（Split、Group、または ColID には指定されていない列）を指定します。デフォルトは Keep All（すべて保持）ですが、Keep（保持）または Drop（除去）で明示的に列をリストすることもできます。

次の例は、前の Stack の例の逆で、元のテーブルと同じ内容を戻します。

```
splitDt = stackedDt << Split(
   split(y),
   ColID(ID),
   output table name(" 分割 "));
```

データテーブルを転置する

Transpose は、行と列を入れ替えることにより、新しいデータテーブルを作成します。行を指定しない場合、Transpose は選択されている行を使います。行が選択も指定もされていない場合は、すべての行を使います。

```
dt << Transpose(
   private, invisible,
   columns( columns ),
   Rows( row matrix ),
   By ( column ),
   Label column name( "name" ),
   Output Table( "name" )

);
```

```
dt << Transpose(
   Columns(columns),
   Rows(row matrix),
   Output Table Name("name"));
```

次の例は、「Big Class.jmp」データテーブルの「**身長 (インチ)**」列と「**体重 (ポンド)**」列を転置します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
tranDt = dt << Transpose(Columns(:Name(" 身長 ( インチ )"),:Name(" 体重 ( ポンド )")),
   output table name(" 転置した列 "));
```

注：この簡単な dt << Transpose コマンドは、転置ウィンドウを表示します。ウィンドウを表示したくない場合は、dt << Transpose(no option) と指定して転置を実行します。

データテーブルを縦方向に連結する

Concatenate（縦方向の連結）は、複数のデータテーブルの行を上下に連結します。

```
dt << Concatenate( DataTableReferences,...,Keep Formulas,
   Output Table Name("name"));
```

たとえば、データテーブルを男子と女子のサブセットに分けた場合、Concatenateを使うとそれを元に戻すことができます。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Select Where( 性別 =="M"); m=dt << Subset(output table name("M"));
dt << Invert Row Selection; f=dt << Subset(output table name("F"));
both=m << Concatenate(f,output table name(" 両方 "));
```

また、連結したデータは、新しいテーブルを作成せずに、現在のデータテーブルに追加することもできます。

```
dt << Concatenate(DataTableReferences, Append to first table);
```

データテーブルを横方向に連結する

Join（横方向の連結）は、データテーブルを左右に連結します。

```
dt << Join(                 // 元のテーブルへのメッセージ
   With(dataTable), // もう一方のデータテーブル
   Select(columns),      // 元のテーブルの列選択 ( オプション )
   Select With(columns), // 結合するテーブルの列選択 ( オプション )
   By Row Number, // 結合タイプ。他に、直積、
                  // またはテーブルごとに指定する
                  // By Matching Columns(col1==col2,col), がある
                  // 最初のテーブルを 2 番目のテーブルのデータで更新する
                  // 同名の列を結合する
   Copy formula(0), // デフォルトはオン。0 でオフになる
   Suppress formula evaluation(0), // デフォルトはオン。0 でオフになる
   // 各テーブルのオプション :
      Drop Multiples(Boolean, Boolean),
      Include NonMatches(Boolean, Boolean),
Preserve Main Table Order(); // 元のデータの順序を維持する
Output Table Name("name")); // 結果テーブル
```

結合を試すために、まず「Big Class.jmp」を2つに分割しましょう。

```
part1=dt << Subset(Columns( 名前 , 年齢 , :Name(" 身長 ( インチ )")), Output Table
  Name("NAH_Big Class"));
part2=dt << Subset(Columns( 名前 , 性別 , :Name(" 体重 ( ポンド )")), Output Table
  Name("NSW_Big Class"));
```

操作を実際の状況に近くするために、part 2 の行を並べ替えます。

```
sortedPart2=part2 << sort(by( 名前 ), Output Table Name(" 並べ替えた NSW_Big Class"));
```

これで、別々の2つの部分に分割され、それぞれの行の順序が異なるデータテーブルができました。2つのデータテーブルは、共通して持つ列の一致によって結合できます。

```
joinDt = part1 << Join(
   With(sortedPart2),
   By Matching Columns( 名前 == 名前 ),
```

```
Preserve Main Table Order();
Output table name("結合"));
```

結果テーブルには、各部分から1つずつ抽出された2つの「名前」変数のコピーがあり、これらを調べるとJoin（結合）の動作を理解できます。Robertの行が4つあることに注意してください。これは、各データテーブルにRobertの行が2つあり（元のテーブルにRobertの行が2つあった）、Join（結合）によってすべての可能な組み合わせが作成されたからです。

ヒント：結合されたテーブルで元のデータテーブルと同じ順序を維持する（対応のある列で並べ替えるのではなく）には、Preserve Main Table Order()を含めます。このオプションにより、結合プロセスがスピードアップします。

データテーブル内のデータを置換する

注：Merge UpdateはUpdateの別名です。

Updateは、あるデータテーブルのデータを別のデータテーブルのデータで置き換えます。

```
dt << Update(                  // 元のテーブルへのメッセージ
   With(dataTable), // もう一方のデータテーブル
   By Row Number, // デフォルトの結合タイプ。または
                  // By Matching Columns(col1==col2)
   Ignore Missing, // 欠測値で置き換えることはしない（オプション）
   );
```

実際の例として、「Big Class.jmp」のサブセットを作成してみましょう。

```
NewHt=dt << Subset(Columns(名前 , :Name("身長 ( インチ )")), Output Table
  Name("hts"));
```

次に、各生徒の身長に0～6インチを追加します。

```
diff = random uniform(6,0);
For Each Row(:Name("身長 ( インチ )")+=diff);
```

最後に、「Big Class.jmp」の生徒の身長をサブセットテーブルの新しい身長で更新します。

```
dt << Update(
   With(NewHt),
   By Matching Columns(名前 == 名前 ),
   );
```

更新されたテーブルに追加する列を制御する

更新後のテーブルには、元のテーブルよりも多くの列が含まれる場合があります。オプションのAdd Column from Update Table()を使えば、更新されたテーブルに含める列を選択できます。

列を追加しない場合は、次のように指定します。

```
Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update Table( None )
);
```

列を追加する場合は、次のように指定します。

```
Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update table( :col1, :col2, :col3 )
);
```

Tabulate を使って表を作成する

Tabulate は記述統計量のテーブルを作成します。表には、グループ列、分析列、および各種統計量が表示されます。次の例は、男子生徒と女子生徒の身長と体重の標準偏差と平均を含むデータテーブルを作成します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Tabulate(                    // データテーブルへのメッセージ
   Add Table(                      // 新しいテーブルの開始
      Column Table( Grouping Columns( : 性別 ) ),
            // 「性別」の列でグループ化
      Row Table( // テーブルに行を追加
         Analysis Columns( :Name("身長 ( インチ )"), :Name("体重 ( ポンド )") ),
            // 分析に「身長 ( インチ )」と「体重 ( ポンド )」の列を使用
         Statistics( Std Dev, Mean ) // 標準偏差と平均を表示
)));
```

注：列の幅を変更して、「表の作成」内で追加の処理を実行することもできます。詳細については、［スクリプトの索引］を参照してください。

欠測値のパターンを見つける

データテーブルに欠測値があるとき、欠測値にパターンが見られるかどうかを調べたい場合があります。

```
dt << Missing Data Pattern(              // データテーブルへのメッセージ
   columns( :miss ),                     // この列の欠測値を検索
   Output Table( "欠測値のパターン" ) // 出力テーブルの名前を指定
);
```

データテーブルを比較する

JMP では、開いた 2 つのデータテーブルを比較し、データ、スクリプト、テーブル変数、列名、列プロパティ、および列の属性の相違点をレポートに表示できます。JSL を使ってデータテーブルを比較するには:

```
obj = dt << Compare Data Tables(
Compare With( Data Table ("Data Table Name"));
```

たとえば、「Students1.jmp」と「Students2.jmp」を比較するには、次のようなスクリプトを書きます。

```
dt = Open( "$SAMPLE_DATA/Students1.jmp" );
dt2 = Open( "$SAMPLE_DATA/Students2.jmp" );
obj = dt << Compare Data Tables( Compare With( Data Table( "Students2" ) ) );
```

相違点を要約した行列を表示するには、次の関数を使用します。

```
mtx = (obj << Get Difference Summary matrix);
```

次のような行列がログウィンドウに表示されます。

```
[-1 1 2 2,
-1 2 4 3,
-1 1 7 4,
1 3 8 4,
1 1 10 9,
0 1 11 11,
-1 1 14 14,
-1 1 16 15,
1 1 18 16,
-1 1 19 18,
0 1 22 20,
0 1 26 24,
1 1 29 27,
1 1 34 33]
```

この例で表示される行列で、第 1 列の値は列に対して行われたアクションを示します。-1 は削除、0 は置換、1 は追加を表します。第 2 列は、そのアクションの作用を受けた行の数を示します。第 3 列と第 4 列は、2 つのデータテーブルの中で作用を受けた行の数を、データテーブルの順に（Students1.jmp、Students2.jmp）示します。

データテーブルに登録（Subscribe）する

データテーブルに変化があったときにメッセージを受け取るようにするようには、Subscribe メッセージを使います。たとえば、列が追加または削除されたときにログにメッセージを送りたいとしましょう。

基本的な構文は次のとおりです。

```
dt << Subscribe( "name"(<"client">), On Delete Columns | On Add Columns | On Add
   Rows | On Delete Rows | On Rename Column | On Close | On Save | On Rename (
   function ) );
```

第1引数は、登録（または"クライアント"）の名前です。これにより、後で設定を解除することが可能になります。

アプリケーションは、データテーブルのクライアント（「一変量の分布」など、ほとんどのビルトインプラットフォーム）としても登録できます。閉じようとしているデータテーブルにクライアントがある場合、データテーブルを必要とする可能性のあるアプリケーションが開いていることが警告されます。

それぞれの登録は、解除するまで有効です。登録の解除は次のように行います。

```
dt << Unsubscribe("keyname", On Delete Columns | On Add Columns | On Add Rows | On
   Delete Rows | On Close | On Col Rename | All);
```

次の例は、行が追加または削除された場合に、ログにメッセージを送ります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
delRowsFn = Function( {a, b, rows},
   dtname = (a << Get Name());
   Print( dtname );
   Print( b );
   Print Matrix( rows );
);
addRowsFn = Function( {a, b, insert},
   dtname = (a << Get Name());
   Print( dtname );
   Print( b );
   Print( insert );
);
dt << subscribe("Test Delete",onDeleteRows( delRowsFn, 3 ));
dt << subscribe("Test Add",onAddRows( addRowsFn, 3 ));
```

空のアプリケーション名

空のアプリケーション名とともにSubscribeが呼び出された場合、JMPは一意の名前を生成して呼び出し元に戻します。次の例では、appname2がクライアントとしてデータテーブルに登録されます。ユーザがデータテーブルを閉じようとすると、確認のメッセージが表示されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
appname1 = dt << Subscribe( "", On Close( Print( "データテーブルを閉じる" ) ) );
appname2 = dt << Subscribe(
   ""("client"),
   On Close(
      Function( {dtab},
         dtname = (dtab << Get Name());
         Print( dtname );
```

```
        );
    );
);
dt << Unsubscribe( appname1, On Close );
Current Data Table() << Run Script (" 一変量の分布 ")
```

行列とデータテーブル間でデータを移動する

行列とデータテーブルの間でデータを移動する方法については、「データ構造」の章の「行列とデータテーブル」（164 ページ）を参照してください。

列

この節では、データテーブルの列に対して実行できる、作成、グループ化、属性やプロパティの設定および取得などのアクションについて説明します。

注：JMP では、数学記号とギリシャ文字が表示できます（［環境設定］の「フォント」で設定）。つまり、T2 乗限界などの列を保存する場合、列名は「T2 乗限界」（特殊記号なし）と「T^2 限界」（特殊記号あり）のどちらでもかまいません。ただし、その列を参照するときに、列名を正確に記述しないとエラーになるので注意が必要です。

データ列のオブジェクトにメッセージを送る

データテーブルの参照にデータテーブルメッセージを送るときと同様に、データ列オブジェクトへの参照に列メッセージを送ることができます。Column 関数は、データ列の参照を戻します。この関数の引数は、引用符で囲んだ名前（または、引用符で囲んだ名前を導き出すもの）または番号のどちらかです。

```
Column(" 年齢 ");        // 「年齢」列への参照
col = Column(2);        // 2 番目の列への参照を割り当てる
```

このマニュアルでは、データ列の参照を *col* で表します。データ列オブジェクトに送ることのできるメッセージは、［スクリプトの索引］（**［データテーブル］＞［列のスクリプト］**）で確認できます。または、Show Properties コマンドを次のように使用します。

```
Show Properties(col);
```

注：列の参照を使用する場合、個々のデータ値を指す添え字を含めなければなりません。添え字を使わないと、列オブジェクト全体を参照することになります。

データ列の参照をグローバル変数に格納した後で、列に変更を加えたいときは、データ列の参照にメッセージを送ります。

列にメッセージを送る方法は、データテーブルにメッセージを送る方法と同じです。オブジェクト、<<演算子、その後に括弧で囲んだ引数を持つメッセージを記述するか、または Send() 関数を使ってオブジェクトとその後にメッセージを記述します。引数を必要としないメッセージもあるので、末尾の括弧はオプションです。

```
col << message(arg, arg2, ...);
Send(col, message(arg, arg2, ...));
```

メッセージは、データテーブルや他のタイプのオブジェクトの場合と同様、次のようにして 1 つにまとめたり、リストにしたりできます。

```
col << message << message2 << ...
col << {message, message2, ...};
```

ヒント：列を削除するにはメッセージをデータ *テーブル*の参照に送る必要があります。オブジェクトを削除できるのはそのオブジェクトのコンテナだけで、オブジェクトは自身を削除できないからです。

列の参照によるセル値へのアクセス

列のセルに含まれている値にアクセスするには、列の参照で添え字を使います。現在の行を参照するときは、空白の添え字を使うことができます。

```
x = col[irow]    // 特定の行
x = col[]        // 現在の行
col[irow] = 2;   // 割り当て式の左辺として
currentDataTable()<<Select Rows Where(col[]<14);  // WHERE 節で使用
```

列の作成

データテーブルに新しい列を追加するには、New Column メッセージをデータテーブルの参照に送ります。第 1 引数の列名は必須です。列名は、引用符で囲むか、名前を導き出す式の形で指定します。

```
dt = Current Data Table();
dt << New Column("ウエハー");
```

または

```
a = "ウエハー";
dt << New Column(a);
```

テーブルにすでに同名の列がある場合は、新しい列名に連番式の数字が追加されます（**ウエハー**、**ウエハー 2**、**ウエハー 3** など）。

特に指定しない限り、列の値は連続量の数値で、列幅は 10 文字です。

- データタイプ（数値、文字、行の属性）
- 尺度（連続、名義、順序）
- 列幅（数値列の場合のみ）
- 数値の表示形式

次の例は、データタイプが数値で、尺度が連続尺度、列幅が5の新しい列を作成します。数値の表示形式は「最適」に設定します。

```
dt << New Column("ウエハー", Numeric, Continuous, Format("最適",5));
```

次の列は、文字列の列を作成し、自動的に名義尺度を割り当てます。

```
dt << New Column("姓", Character);
```

列の属性に合わせて、式やその他のスクリプトメッセージを入れることもできます。

```
dt << New Column("比率", Numeric, Continuous, Formula(:Name("身長(インチ)")/
   :Name("体重(ポンド)")));
dt << New Column("マーカー", Row State, Set Formula(Marker State(age-12)));
```

特定の列を後で操作する（グループ化する、データタイプを変更するなど）予定がある場合は、次のようにして列の参照を作成します。

```
myCol = dt << New Column("部品番号");
```

列にデータを挿入する

列にデータを挿入するには、Values、またはそれと等価のSet Valuesを使用します。各セルの値をリストに含めます。

次の例は、現在のデータテーブルに「姓」という新しい列を追加し、そこに「Smith」、「Jones」、「Anderson」という3つの値を挿入します。

```
dt = Current Data Table();
dt << New Column("姓", Character, Values({"Smith", "Jones", "Anderson"}));
```

列には、数値を入れることもできます。次の例は、現在のデータテーブルに「行番号」という新しい列を追加します。N Row関数がテーブル内の行の数を戻し、新しい列のすべての行に1から始まる数値が挿入されます。

```
dt = Current Data Table();
dt << New Column("行番号", Numeric, Values(1::NRow()), Format("最適",5));
```

乱数の列を追加するには、乱数関数と式を挿入します。

```
dt = Current Data Table();
dt << New Column("乱数", Numeric, Formula(Random Uniform()));
```

また、各セルに一定の数値を挿入することもできます。次の例は、各セルに「5」を挿入します。

```
New Column("数");
:Number << Set Each Value(5);
```

New Columnはビルトイン関数として使用することもできます。データテーブルの参照に対して<<（Send）コマンドを使用しない方法です。ビルトイン関数として使うと、列は現在のデータテーブルに追加されます。

```
dt << New Column("logHt"); // コマンドをデータテーブルの参照に送る
New Column("logHt"); // 列は現在のデータテーブルに追加される
```

一度に複数の列を追加

Add Multiple Columns メッセージは、一度に複数の列を作成します。引数は、列名の接頭辞、列の数、列の挿入位置（Before First、After Last、After(col)）、データタイプ（Numeric、Row State、Character(width)）です。また、数値列を対象とするオプションの引数に、フィールド幅があります。

```
dt = Current Data Table();
dt << Add Multiple Columns("最初", 2, Before First, Row State);
   // 行属性を持つ「最初 <番号>」という列を 2 つ、最初の列の前に挿入する

dt = Current Data Table();
dt << Add Multiple Columns("中間", 3, After(:Name("身長 ( インチ )")), Numeric);
   // 「中間 <番号>」という数値列を 3 つ、「身長 ( インチ )」列の後ろに挿入する

dt = Current Data Table();
dt << Add Multiple Columns("最後", 4, After Last, Character(4));
   // 「最後 <n>」という名前の文字列の列を 4 つ、最後の列の後ろに追加する
```

列名は接頭辞です。同名の列が複数追加される場合、列名に連番が付きます（**最初 1**、**最初 2**、など）。

列のグループ化

列をグループ化するには、対象となる列のリストを引数として指定した Group Columns メッセージをデータテーブルに送ります。たとえば、次のコードは、「Big Class.jmp」データテーブルを開いて、「**年齢**」と「**性別**」の列をグループ化します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Group Columns( {:年齢, :性別} );
   "年齢など"
```

また、グループに含める最初の列の列名に、グループに含める列数を添えて、送ることもできます。その場合、名前で指定した列から数えて *n* 個の列がグループに含まれます。次の行は、前述の行と等価で、「**年齢**」と「**性別**」をグループ化します。

```
dt << Group Columns( :年齢, 2 );
```

グループ名は、引数で指定した第 1 列に基づきます。前述の例では、グループに自動的に「**年齢など**」という名前が付きます。名前をカスタマイズするには、グループ名を第 1 引数に含めます。

```
dt << Group Columns( “グループ”, :年齢, 2 );
```

列のグループ化を解除するには、対象となる列のリストを引数として指定した Ungroup Columns メッセージを使います。たとえば、次の行は、前述の例でグループ化した 2 つの列のグループ化を解除します。

```
dt << Ungroup Columns( {:年齢, :性別} );
```

グループ化とその解除に関しては、次のことに注意してください。

- どちらのメッセージも引数には 1 つのリストを指定します。リストは括弧で囲む必要があります。

- 1つのメッセージで複数のグループを作成することはできません（Group Columnsで2つの列リストを指定するなど）。2つの別々のGroup Columnsメッセージを送る必要があります。
- Ungroup Columnsメッセージでは、列のグループ名ではなく、グループ解除する列のリストを指定します。グループから一部の列を削除できます。たとえば、次の行は、4つの列から成るグループを作成します。

```
dt << Group Columns( {: 年齢 , : 性別 , :Name(" 身長 ( インチ )"), :Name(" 体重 ( ポンド )")} );
```

次の行は、このうち2つの列だけをグループから削除します。

```
dt << Ungroup Columns( {: 年齢 , : 性別 } );
```

グループ列は「**身長(インチ)**」と「**体重(ポンド)**」ですが、グループ名は「**年齢**」のままであることに注目してください。いったんグループが作成されたら、元々グループ化されていた最初の列を削除しても名前は変わりません。

列グループまたは名前の取得

列グループまたは名前を取得するには、Get Column Groupまたは Get Column Groups Namesを使用します。この例では、まず前の節で示した手順でグループの列を2つ作成します。Get Column Group()は指定のグループに含まれる列の名前を戻します。Get Column Groups Names()は列グループの名前を戻します。

```
dt = Open( "$sample_data/big class.jmp" );
dt << Group Columns( {: 年齢 , : 性別 } );
   "年齢など"
dt << Group Columns( {:Name(" 体重 ( ポンド )"), :Name(" 身長 ( インチ )")} );
   "身長 ( インチ ) など"
dt << Get Column Group( " 年齢など " );
   {: 年齢 , : 性別 }
dt << Get Columns Groups Names();
   {"年齢など", "身長 ( インチ ) など"}
```

列グループの選択

列グループを選択する、または選択を解除するには、次のメッセージを使用します。

```
dt << Select Column Group (" 名前 "); // グループ内の列を選択
dt << Deselect Column Group (" 名前 "); // グループ内の列の選択を解除
```

次の例は、**X**と**Y**の列、「**オゾン**」から「**鉛**」までの列をそれぞれグループ化し、データテーブル内でこれらの列を選択します。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << group columns( "xy", {:X, :y} );
dt << group columns(
   " 汚染物質 ",
   : オゾン :: : 鉛
);
dt << select column group( "xy", " 汚染物質 " );
```

列グループの名前の変更

列グループの名前を変更するには、次のメッセージを使用します。

```
dt << Rename Column Group ("名前", "変更後の名前");
```

次の例は、列グループの名前を「**xy**」から「**座標**」に変更します。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << group columns( "xy", {:X, :y} );
dt << group columns(
   "汚染物質",
   :オゾン :: :鉛
);
Wait( 1 );
dt << rename column group( "xy", "座標" );
```

列グループの移動

列グループを移動させるには、次のメッセージを使用します。

```
dt << Move Column Group (To First | To Last | After (col) "名前")
```

次の列は、「**汚染物質**」グループの列を最後尾に移動させます。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << group columns( "xy", {:X, :y} );
dt << group columns(
   "汚染物質",
   :オゾン :: :鉛
);
dt << move column group( to last, "汚染物質" );
```

列の選択

列を選択するには、Set Selected メッセージを使用します。

```
col << Set Selected(1);
```

たとえば、「Big Class.jmp」データテーブルのすべての連続尺度の列を選択するには、次のスクリプトを使用します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
cc=dt << Get Column Names(Continuous);
ncols = N Items(cc);
for(i=1,i<=ncols,i++,
   cc[i]<<Set Selected(1);
);
```

選択されている列の取得

現在選択されている列のリストを取得するには、Get Selected Columnsメッセージを使います。

```
dt << Get Selected Columns();
```

どの列が選択されているかがわかったら、それらの列に対してアクションを行うスクリプトを書くことができます。または、列を繰り返し選択し、一度に1つずつ処理するスクリプトを書きます。

列を選択してから取得する場合は、列にSet Selected()メッセージを送ります。詳細は、「列属性」（302ページ）を参照してください。

列への移動

特定の列を選択し、そこに移動するには、Go Toメッセージを使用します。

```
dt << Go To ( 列名 | 列番号 );
```

列数が多いデータテーブルに対し、このメッセージを使うと、データテーブルが左へとスクロールされ、第1列が選択された形で画面に表示されます。

```
dt=Open("$SAMPLE_DATA/Tiretread.jmp");
dt << Go To (1)
```

列を並べ替えて移動

以下のメッセージを使うと、データテーブル内の列を並べ替えることができます。

```
dt << Reorder By Name;           // 名前順
dt << Reverse Order;             // 現在と逆の順序
dt << Reorder By Data Type;      // 行の属性、文字列、数値の順
dt << Reorder By Modeling Type;  // 連続尺度、順序尺度、名義尺度の順
dt << Original Order;            // 保存されている順序
```

次のコマンドは、現在選択されている列を指定した移動先に移動します。

```
dt << Move Selected Columns(To First);
dt << Move Selected Columns(To Last);
dt << Move Selected Columns(After(“名前”));
```

次の構文を使うと、事前にデータテーブル内の列を選択しなくても、列を移動できます。

```
dt << Move Selected Columns({“名前”}, To First);
dt << Move Selected Columns({“名前”}, To Last);
dt << Move Selected Columns({“名前”}, After(“名前”));
```

列スイッチャーの追加

JSLを使ってレポートウィンドウに列スイッチャーを追加します。列スイッチャーがあると、分析を再度呼び出す手間をかけずに、異なる列をすばやく分析できます。

```
obj << Column Switcher(<default_col>(<col1>, <col2>, ...); // レポートに列スイッチャー
   を追加する
obj << Remove Column Switcher(); // レポートから列スイッチャーを削除する
```

たとえば、「Car Poll.jmp」データテーブルの「分割表」レポートに列スイッチャーを追加するには、次のようにします。

```
dt = Open( "$SAMPLE_DATA/Car Poll.jmp" );
obj = Contingency(
   Y( :サイズ ),
   X( :既婚 / 未婚 )
);
ColumnSwitcherObject = obj <<
Column Switcher(
   :既婚 / 未婚 ,
   {:性別 , :生産国 , :既婚 / 未婚 }
);
ColumnSwitcherObject<<setsize(200); // スイッチャーの幅を示すピクセル数
ColumnSwitcherObject<<setnlines(6); // スイッチャーに表示する列の数
```

列スイッチャーの詳細については、『JMP の使用法』を参照してください。

選択された列の圧縮

データテーブルが大きい場合に、サイズを最小に抑えるには、Compress Selected Columns メッセージを使用します。各列ができる限りコンパクトな形に圧縮されます。

```
dt << Compress Selected Columns( { 列名 , 列名 } );
```

たとえば、「Big Class.jmp」内の「**年齢**」、「**性別**」、「**身長 (インチ)**」、「**体重 (ポンド)**」の各列を圧縮するには、次のようにします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Compress Selected Columns(
{:年齢 , :性別 , :Name("身長 ( インチ )"), :Name("体重 ( ポンド )")}
);
```

この機能の詳細については、『JMP の使用法』を参照してください。

列の削除

列を削除するには、Delete Columns メッセージを送って、削除する列（複数可）を指定します。複数の列を削除するには、列を複数の引数として個別に指定するか、または 1 つのリストで指定します。次の例は、「Big Class.jmp」サンプルデータテーブルを使用します。

```
dt << Delete Columns("体重 ( ポンド )");
dt << Delete Columns("体重 ( ポンド )", "年齢", "性別");
dt << Delete Columns({"体重", "年齢", "性別"});
```

引数がない場合、Delete Columnsは、前回選択された列を削除します。詳細については、「列属性」（302 ページ）を参照してください。

列名の取得

Column Name(*n*)は、*n*番目の列の参照を戻します。ここでは、「Big Class.jmp」サンプルデータテーブルを用います。

```
column name(2); // 「年齢」列の参照を戻します。
```

戻される値は、引用符で囲まれた文字列ではなく、「年齢」列の参照値です。これは、スクリプト内で通常、実際の列の名前を使う場所で、どこでもこの演算子を使用できます。たとえば、この演算子に添え字を使うことができます。

```
column name(2)[1];
   12
```

名前をテキストの文字列として取得したい場合は、Charでクォートします。

```
char(column name(2));
   "年齢"
```

データテーブルのすべての列の名前をリストで取得するには、Get Column Namesを実行します。

```
dt << Get Column Names(引数)
```

オプションの*引数*は、次のようにGet Column Names関数の出力を制御します。

- Numeric、Character、またはRow Stateを指定すると、これらのタイプに相当する列の名前だけが戻ります。
- Continuous、Ordinal、またはNominalを指定すると、指定した尺度の列の名前だけが戻ります。
- Stringを指定すると、列名ではなく文字列のリストが戻ります。

たとえば、「Big Class.jmp」データテーブルで連続尺度の数値列を選択するには、次のスクリプトを使用します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
names=dt << Get Column Names (Numeric, Continuous)
   {Name( "身長(インチ)" ), Name( "体重(ポンド)" )}
```

列属性

データテーブル列へのメッセージを使うと、名前、データ、状態、メタデータなど、さまざまな列の属性や特性をすべて制御できます。これらのメッセージは対になっており、一方は各属性を設定（set）または割り当て、もう一方は各属性の現在の設定を取得（get）または問い合わせます。

たとえば、列の非表示、除外、ラベル、スクロールロック／ロック解除などをスクリプトで実行できます。値はブール値なので、1を入力するとその属性がオンになり、ゼロを送るとオフになります。

次の例では、「名前」列が「表示する」、「除外しない」、「ラベルあり」、「スクロールロック」の状態になります。

```
column("名前") << hide(0);
column("名前") << exclude(0);
column("名前") << label(1);
column("名前") << scroll lock(1);
```

注：各種の引数を設定するメッセージ（Set Name、Set Values、Set Formula など）はすべて Set で始まりますが、Set という語は Set Name 以外のメッセージでは省略可能です（Name はすでに名前に特殊な文字を使うときのコマンドとして使われています）。好きな方の形式、または覚えやすい方の形式を使ってください。引数の現在の設定値を取得するメッセージ（Get Formula など）は、Set ではなく Get で始まり、Get は**省略できません**。

列の選択を解除するには、データテーブルオブジェクトに Clear Column Selection メッセージを送ります。

```
dt << Clear Column Selection;
```

列名の設定または取得

Set Name を使って、列に名前をつけたり列の名前を変更したりできます。Get Name は列の名前を戻します。次の例は、「Big Class.jmp」データテーブルを使い、第 2 列（「年齢」）を「比率」に変更します。その後、現在の列名をログに戻します。

```
col = Column(2);
col << Set Name("比率");
col << Get Name;
```

列の値の設定または取得

同様に、Set Values は列に値を設定します。変数が文字タイプの場合、引数はリストである必要があります。数値タイプの場合は行列（ベクトル）です。値の数がデータテーブルの現在の行数より多い場合は、必要な行が追加されます。Get Values は、値をリストまたは行列の形で戻します。Get As Matrix は Get Values に似ていますが、数値列の値を戻します。

```
col << Set Values(myMatrix);    // 数値変数の場合
col << Set Values(myList);      // 文字変数の場合
col << Get Values;           // 行列を戻す。文字の場合はリストを戻します
col << Get Matrix(<列名のリスト>|<列番号のリスト>|<列範囲>; // 指定の列を行列として戻す
```

次の例は、値のリストと行列を戻します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
column("名前") << values({Fred, Wilma, Fred, Ethel, Fred, Lamont});
myList = :名前 << get values;    // リストを戻す

column("年齢") << values([28,27,51,48,60,30]);
myVector = :年齢 <<get values;
   [28, 27, 51, ... ]
```

```
myMatrix = :Name("体重(ポンド)")<<get as matrix;
   [95, 123, 74, ... ]
```

値ラベルの設定または取得

注：値ラベルの詳細については、『JMPの使用法』を参照してください。

値ラベルを使うと、簡略化されたデータの意味を説明するラベルを表示できます。たとえば、0と1の値を持つデータ列があり、0は男性、1は女性を表すとします。0の値ラベルを「男性」、1の値ラベルを「女性」とすると、わかりやすくなります。

値ラベルは、次の3つの方法のどれかで指定できます。「Big Class.jmp」サンプルデータテーブルでは、Mが男性、Fが女性を表しています。

```
:sex << Value Labels({“F”, “M”}, {“女性”, “男性”}); // 2つのリストを使う
:sex << Value Labels({"F", "女性", "M", "男性"}); // ペアのリストを使う
:sex << Value Labels({"F" = "女性", "M" = "男性"}) // 割り当てのリストを使う
```

値のラベルを有効にするには、列に`Use Value Labels`メッセージを送ります。

```
:性別 << Use Value Labels(1);
```

列の実際の値を表示するには、次のように指定します。

```
:性別 << Use Value Labels(0);
```

同じメッセージを使って、データテーブルのすべての列に対して値ラベルのオン／オフを切り替えることができます。

```
Current Data Table()<<Use Value Labels(1);
```

データと尺度の設定または取得

JSLを使って列のデータタイプを設定または取得することができます。設定できるタイプは、Character（文字）、Numeric（数値）、Row State（行の属性）です。次の例は、「Big Class.jmp」データテーブルに文字タイプのデータを入れる新しい列を追加します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "新規" );
Column( "新規" ) << Data Type( Character );
Column( "新規" ) << Get Data Type;
   "Character"
```

列の尺度を設定または取得するには:

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
col = New Column( "新規" );
col << Modeling Type( "Continuous" );
col << Get Modeling Type;
   "Ordinal"
```

列のデータタイプを変更するときは、次のようにして列の形式を指定できます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "日付" );
Column( "日付" ) << Data Type( Numeric, Format( "ddMonYYYY" ) );
```

Set Data Type と Set Modeling Type は、それぞれ Data Type および Modeling Type と等価です。

形式の設定または取得

Format メッセージは、数値と日付／時間の形式を制御します。最初の引数は、列情報ウィンドウにある表示形式の選択肢を引用符で囲んだ文字列です。後続の引数は、表示形式によって異なります。フィールド幅も個別に設定できます。

```
col << Format("best",5);              // 幅が 5
col << Format("Fixed Dec",9,3);       // 幅が 9、小数点以下の桁数が 3
col << Format("PValue",6);
col << Format("d/m/y",10);
col << set fieldwidth(30);
```

日付の形式では、Format メッセージにより日付をデータテーブルにどのように表示するかを指定します。データの入力形式や、入力または編集に備えて現在のセルを選択したときの表示形式を設定するには、Input Format メッセージを使用します。

```
col << Format("d/m/y",10); // 日付を日 - 月 - 年の順序で表示する
col << Input Format("m/d/y"); // 日付を月 - 日 - 年の順序で入力する
```

日付時間の詳細については、「データタイプ」の章の「日付時間の関数と形式」（123 ページ）を参照してください。

注：Format メッセージと、指定された形式に従って数値を文字列に変換する **Format 関数**を混同しないでください（「データタイプ」の章の「日付時間の関数と形式」（123 ページ）で説明しているように、通常、Format 関数は日付／時間の表記に使われます）。オブジェクトにメッセージを送った場合は、同じ名前の関数を使った場合と結果がまったく異なります。

列の現在の形式を取得するには、次のように Get Format メッセージを実行します。

```
col << Get Format;
```

計算式の設定、取得、評価

列の計算式は、次のようにして設定、取得、および評価することができます。

```
col=New Column("比率");          // 列を作成し、その参照を格納する
col << Set Formula(:Name("身長（インチ）")/:Name("体重（ポンド）")); // 計算式を設定する
col << EvalFormula;                  // 計算式を評価する
col << Get Formula;        // 式 :Name("身長（インチ）")/ :Name("体重（ポンド）") を戻す
```

列の値をスクリプトの中で使用する場合は、計算式を評価するコマンドを必ず加えてください。計算式の評価のタイミングは、JMPのバージョンによって異なります。次の点を念頭に置いてください。

- 計算式を追加すると、バックグラウンドで評価するようにスケジュールされます。スクリプト実行中の列の値に依存するような場合は、これが問題となる可能性があります。
- 1つの列を評価するときには、列に Eval Formula コマンドを送ることができます。これは、列を作成するコマンド内の計算式の節の直後で実行できます。

  ```
  dt << NewColumn("比率",Numeric,Formula(:Name("身長(インチ)")/:Name("体重(ポンド)")),EvalFormula);
  ```

  ここでの Formula は Set Formula の別名です。

  ただし、計算式を追加する作業がすべて終わってから、次のように Run Formulas コマンドを使って、正しい順序ですべての計算式を実行させるようにしてください。

  ```
  Current Data Table()<<Run Formulas;
  ```

- Eval Formula は計算式の評価をしますが、バックグラウンドでの再計算を行うため、Run Formulas コマンドの方が Eval Formula より適しています。システムのバックグラウンドで行われる計算タスクは、計算式が正しい順序で評価されるよう、十分な注意を払う必要があります。Run Formulas は、すべての計算処理が終了するまで、他のタスクを呼び出しません。
- 同一の値のセットを生成させるために乱数関数と Random Reset(*seed*) の機能を使用している場合には、Run Formulas コマンドを使うと2回目の評価を行わずに済みます。

範囲チェックおよびリストチェックの設定と取得

JSLを使って、リストチェックおよび範囲チェックのプロパティを操作できます。次の例は、「Big Class.jmp」サンプルデータテーブルを使用します。

「性別」という列のリストチェックプロパティを設定し、クリアします。

```
column("性別") << List Check({"M", "F"});     // リストチェックプロパティを設定
column("性別") << List Check();               // リストチェックプロパティをクリア
```

範囲チェックでは、表9.1の構文を使って範囲を指定する必要があります。

表9.1 範囲チェックの構文

指定する範囲	使用する関数
$a \le x \le b$	LELE(a, b)
$a \le x < b$	LELT(a,b)
$a < x \le b$	LTLE(a,b)
$a < x < b$	LTLT(a,b)

次の例は、「年齢」列の値が0より大きく、120未満でなければならないことを指定します。

```
Column("年齢") << Range Check(LTLT(0, 120));
```

どの演算子の前にも Not を置くことができます。また、上側または下側だけの範囲も指定できます。次の例は、「年齢」列の値が 12 以上でなければならないことを指定します。

```
Column("年齢") << Range Check(not(lt(12)));
```

範囲チェックの属性をクリアするには、次のように空の range check() を実行します。

```
Column("年齢") << Range Check();
```

列に割り当てられているリストチェックまたは範囲チェックを取得するには、列に Get List Check または Get Range Check メッセージを送ります。

```
Column("性別") << Get List Check;
Column("年齢") << Get Range Check;
```

たとえば、上記の例の「**年齢**」列（値が 0 より大きく 120 未満に設定されている）に Get Range Check を送ると、次のような出力が表示されます。

```
Range Check(LTLT(0, 120))
```

Set Property、Get Property、および Delete Property を使って、リストチェックと範囲チェックを設定、取得、削除することもできます。詳細については、「列プロパティ」（308 ページ）を参照してください。

注：列の範囲チェックに関する処理を JSL で行う場合は、すべての警告が、インタラクティブなウィンドウではなくログウィンドウに表示されます。

列スクリプトの取得

Get Script は、列を作成するためのスクリプトを戻します。

```
New Column("比率",Set Formula( :Name("身長（インチ）")/ :Name("体重（ポンド）")));
Column("比率") << Get Script;
   New Column("比率");
      Numeric,
      Continuous,
      Format( "Best", 10 ),
      Formula(Name("身長（インチ）") / :Name("体重（ポンド）"))
   );
```

役割の事前選択

列の役割をあらかじめ選択するには、Preselect Roles メッセージを使用します。選択できる役割は、No Role、X、Y、Weight、および Freq です。Get Role メッセージは、現在の設定を戻します。

```
col = New Column( "新規" );
col << Preselect Role( X );
col << Get Role;
```

列のロック

列をロックまたはロック解除するには、ブール引数を指定して Lock または Set Lock を使います。Get Lock は現在の設定を戻します。

```
col << Lock(1);      // ロック
col << Set Lock(0); // ロック解除
col << Get Lock;     // 現在の状態を表示
```

列プロパティ

ヒント：JSL で使用できるプロパティは、「列情報」ウィンドウの［**列プロパティ**］メニューにあるものと同じです。各プロパティの引数は、「列情報」ウィンドウの設定に対応します。構文を習得する簡単な方法は、まず列情報ウィンドウで目的のプロパティを設定し、次に Get Property を使って JSL を表示することです。

データ列には、多数のメタデータの属性オプションがあります。この属性は、Get Property、Set Property、および Delete Property 使って、設定、取得、または削除することができます。

```
col << Set Property( " プロパティ名 ", { 引数リスト } );
col << Get Property( " プロパティ名 " );
col << Delete Property( " プロパティ名 " );
```

設定する際のプロパティ名は、常に Set Property の第 1 引数になり、設定するプロパティによって後続の引数が決まります。

- Get Property と Delete Property は常にプロパティ名を指定する引数を 1 つだけとります。
- Get Property は、プロパティの設定を戻します。Delete Property は、列からプロパティを完全に削除します。
- 列に設定されている列プロパティ名のリストを取得するには、列を指定し、Get Property List を使います。

複数のプロパティを設定する場合は、Set Property メッセージを個別に複数回送る必要があります。単一の JSL ステートメントに複数のメッセージをまとめることもできます。

```
col << Set Property(" 軸 ",{ 最小値 (50), 最大値 (180)})<<Set Property(" ノート ", " 比率を
   取得する ");
```

プロパティの値を取得するには、引数に目的のプロパティの名前を指定して Get Property メッセージを送ります。

```
Column(" 比率 ")<<Get Property(" 軸 "); // 軸の設定を戻す
```

列をラベル列として設定するには：

```
dt << Set Label Columns(col1, col2, col3);
```

すべてのラベル列をクリアするには：

```
dt << Set Label Columns();
```

同じ構文を Set Scroll Lock Columns でも使用できます。

表 9.2 データテーブルの列のプロパティ

プロパティ	説明	引数の使用例
Notes（ノート）	列についての注記を保存する。引用符で囲んだ文字列	col<<Set Property("Notes", "Fisher のアヤメのデータから抽出 "); col<<Get Property("Notes");
List Check（リストチェック） Range Check（範囲チェック）	列に入力できる値を指定する	col<<Set Property(List Check,{"F","M"}); col<<Set Property(Range Check,LTLT(0, 120)); col<<Get Property(List Check); col<<Delete Property(Range Check);
Missing Value Codes（欠測値のコード）	欠測値として扱う列の値を指定する	col<<Set Property("Missing Value Codes", {0, 1});
Value Labels（値ラベル）	値の代わりに表示するラベルを指定する	col<<Value Labels({0 = "Male", 1 = "Female"}); ラベルのオンとオフは、ブール値を使って切り替えます。 col<<Use Value Labels(1);.
Value Ordering（値の順序）	レポートに表示するデータの順序を指定する	col<<Set Property("Value Ordering", {"Spring", "Summer", "Fall", "Winter"});

表 9.2 データテーブルの列のプロパティ（続き）

プロパティ	説明	引数の使用例
Value Colors（値の色）および Color Gradient（カラーグラデーション）	カテゴリカルデータまたは連続尺度のデータに色を設定する	各値にJMPの色番号を割り当てることで色を指定する。 `col<<Set Property( "Value Colors", {"Female" = 3, "Male" = 5} );` カラーテーマを指定するには: `col<<Set Property( "Value Colors", Color Theme( White to Blue ));` カラーテーマを指定しなかった場合、カラーテーマの環境設定が使用されます。 カラーグラデーション、グラデーションの範囲および中間点を指定する。 `col<<Set Property( "Color Gradient", {"White to Blue", Range( {18, 60, 25} )});`
Axis（軸）	ほとんどのプラットフォームで、グラフ軸の構築に使われる。ブール値を使うのが普通です。	`col<<Set Property("Axis",{Min(50), Max(180), Inc(0), Minor Ticks(10), Show Major Ticks(1), Show Minor Ticks(1), Show Major Grid(0), Show Labels(1), Scale(Linear)});`
Coding（コード変換）	実験計画（DOE）とあてはめに使われる。下限値と上限値のリスト	`col<<Set Property("Coding",{59,172});` `col<<Get Property("Coding");`
Mixture（配合）	DOE、あてはめ、およびプロファイルに使われる。リストの形で「配合」列プロパティを指定する	`col<<Set Property("Mixture", {0.2, 0.8, 1, L PseudoComponent Coding} );` `col<<Get Property("Mixture");`
Row Order Levels（データの出現順）	水準を、値の順にではなくデータ内での出現順に並べるよう指定する	`col<<Set Property( "Row Order Levels", 1 );`
Spec Limits（仕様限界）	工程能力分析と変動性図に使われる	`col<<Set Property("Spec Limits", {LSL(-1), USL(1), Target(0)});` `col<<Get Property("Spec Limits");`
Control Limits（管理限界）	管理図に使われる	`col<<Set Property("Control Limits", {XBar(Avg(44), LCL(29), UCL(69))});` `col<<Get Property("Control Limits");`

表 9.2 データテーブルの列のプロパティ（**続き**）

プロパティ	説明	引数の使用例
Response Limits（応答変数の限界）	DOE で指定し、満足度関数で使われる	col<<Set Property("Response Limits", {Goal(Match Target), Lower(1,1), Middle(2,2), Upper(3,3)}); col<<Get Property("Response Limits"); Goal（目標）に指定できるのは、Maximize、Match Target、Minimize、None です。他の引数は、数値引数と満足度の引数をとります。
Design Role（因子の役割）	DOE に使われる。役割を 1 つ指定します。	col<<Set Property("Design Role",Covariate); col<<Get Property("Design Role"); 指定できる役割は、Continuous、Categorical、Blocking、Covariate、Mixture、Constant、Signal、Noise です。
Factor Changes（因子の変更）	因子変更の困難さ（Easy、Hard、Very Hard）を設定する	col<<Set Property("Factor Changes", Hard); col<<Get Property("Factor Changes");
Sigma	管理図に使われる 既知の標準偏差値を指定する。	col<<Set Property("Sigma",1.332); col<<Get Property("Sigma"); 管理図の種類によって sigma の計算方法が異なる。
Units（単位）	カスタムで使われる 計測単位を指定する。	col<<Set Property("units", grams); col<<Get Property("units");
Distribution（分布）	列にあてはめる分布の種類を設定する	col<<Set Property("Distribution", Distribution(GLog)); col<<Get Property("Distribution");
Time Frequency（時間の単位）	時間の単位の種類を設定する	col<<Set Property("Time Frequency", Time Frequency(Annual)); col<<Get Property("Time Frequency");
Map Role（地図の役割）	地図シェープデータと名前データを接続するための列の使用方法を設定する。役割と、必要に応じてその他の情報を指定する	col<<Set Property("Map Role", Map Role(Shape Name Use("filepath to data table", "column name"))); col<<Get Property("Map Role");

表 9.2 データテーブルの列のプロパティ（続き）

プロパティ	説明	引数の使用例
[カスタムプロパティ]	カスタムで使われる 列情報ウィンドウ内の［列プロパティ］>［その他］に対応する。 第 1 引数はカスタムプロパティの名前で、第 2 引数は式です。	col<<Set Property(" 記録日 ",12Dec1999); long date(col<<Get Property(" 記録日 ")); // リストのリスト

行

この節では、データテーブルの行の追加や操作を行うメッセージについて説明します。行メッセージは、データテーブルの参照に送られ、ほとんどの場合、現在選択されている行に対して動作します。行メッセージの中には、スクリプトでは実用的でないものも多くあります（たとえば、Move Rows）。

行の追加

行を追加するには、Add Rows メッセージを送って行数を指定します。また、新しい行をどの行の後に挿入するかも指定できます。引数に指定できるのは、数値または数値を導き出す式のどちらかです。

```
dt<<Add Rows(3); // データテーブルの最後に 3 行追加する
dt<<Add Rows(3, 10); /* 10 行目の後、11 行目の前に 3 行追加する */
```

Add Rows を利用すると、割り当てのリストを 1 つの引数として指定できます。割り当て式はカンマまたはセミコロンで区切ります。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
dt<<Add Rows({: 名前 ="Peter", : 年齢 =14, : 性別 ="M", :Name(" 身長 ( インチ )")=61,
   :Name(" 体重 ( ポンド )")=124});
add point = expr( dt<<addRows({:xx=x; :yy=y}) );
```

複数の引数を指定して、複数のリストやリストのリストを送ることもできます。次のスクリプトは、さまざまな方法で Add Rows コマンドを使ってデータテーブルを作成します。

```
dt = new Table(" 都市 ");
dt<<NewColumn("xx",Numeric);
dt<<NewColumn("cc",Character,width(12));

dt<<AddRows({xx=12,cc=" シカゴ "});                                    // 1 つのリスト
dt<<AddRows({xx=13,cc=" ニューヨーク "},{xx=14,cc=" ニューアーク "});  // 複数のリスト
dt<<AddRows({{xx=15,cc=" サンフランシスコ "},{xx=sqrt(256),cc=" オークランド "}});
                                                                       // リストのリスト
```

```
a = {xx=20,cc=" マイアミ "};
dt<<AddRows(a); // 1 つのリストとして評価する

b={{xx=17,cc=" サンアントニオ "},{xx=18,cc=" ヒューストン "},{xx=19,cc=" ダラス "}};
dt<<AddRows(b); // リストのリストとして評価する
```

メッセージで指定できる行の詳細については、「行の属性と演算子」（321 ページ）を参照してください。

行の削除

行を削除するには、Delete Rows メッセージを送って、削除する行（複数可）を指定します。複数の行を削除するには、行番号（*rownum*）引数でリストまたは行列を指定するか、Delete Rows コマンドを For などの他のコマンドと組み合わせて使います。行番号（*rownum*）引数として指定できるのは、番号、番号のリスト、番号の範囲、行列、またはこれらのどれかを導き出す式です。引数を指定しないと、Delete Rows は、現在選択されている行を削除します。引数も現在選択されている行もない場合は、Delete Rows は何の動作もしません。

```
dt<<Delete Rows(10); // 行 10 を削除する
dt<<Delete Rows({11,12,13}); // 行 11 ～ 13 を削除する
myList={11,12,13}; dt<<Delete Rows(mylist);// 行 11 ～ 13 を削除する
dt<<Delete Rows(1::20); // 最初の 20 行を削除する
dt<<Delete Rows([1 2 3]); // 最初の 3 行を削除する
```

たとえば次のスクリプトは、「Big Class.jmp」を開き、10 行目を削除します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
selected(Row State(10))=1;dt<<delete rows;  // 行 10 を選択して削除する
```

同じ行を複数回リストしてもかまいません。リストする順序も自由です。

以下に、任意のサイズのデータテーブルから最後の *x* 行を削除する一般的な方法を示します。次の例は、データテーブルの最後の 5 行を削除します。

```
x=5;
n=NRow(dt); For(i=n,i>n-x,i--,
   dt<<Delete Rows(i));
```

NRow は、テーブル内の行をカウントします。詳細は、「各行に対してスクリプトを反復する」（319 ページ）を参照してください。

行の選択

Select All Rows は、データテーブル内のすべての行を選択（強調表示）します。

```
dt<<Select All Rows;
```

すべての行が選択されている場合、Invert Row Selectionを使用してそれらの選択を解除できます。このコマンドは、各行の選択の状態を逆にするので、選択されている行は選択が解除され、選択されていない行は選択されます。

```
dt<<Invert Row Selection;
```

注：現在の選択内容によって結果が異なるInvert Row Selectionを除き、選択メッセージを新しく使用した場合、新たに選択が行われます。すでに選択している行があり、選択された行に新しいメッセージを送ると、それらの行は、まず選択が解除されます。

特定の行を選ぶには、Go To Rowを使います。

```
dt<<Go To Row(9);
```

データテーブルの特定の行を行番号で選択するには、Select Rowsコマンドを使います。コマンドの引数は行番号のリストです。たとえば、データテーブルの行1、3、5、および7を選択するには、次のように指定します。

```
dt<<Select Rows({1, 3, 5, 7});
```

行の範囲を選択するには、次のメッセージのいずれかを指定します。

```
Current Data Table() << Select Rows( Index( 7, 10 ) );
Current Data Table() << Select Where( Any( Row() == Index( 7, 10 ) ) );
```

どちらの例も、現在のデータテーブルの中で7行目から10行目までを選択します。

データ値に基づいて行を選ぶには、Select Whereを使い、引数として条件式を指定します。

ヒント：Select Where メッセージの中で使用できる関数と演算子については、「JSL の構成要素」の章の「演算子」（86 ページ）を参照してください。

たとえば、「Big Class.jmp」データテーブルのうち、生徒の年齢が14歳以上である行を選択するには：

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt<<Select Where(年齢>13);
```

生徒の年齢が14歳未満である行を選択するには：

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
col = Column( dt, 2 );
dt << Select Where( col[] < 14 );
```

次の例は、生徒の年齢が15歳未満で性別がF（女性）である行を選択します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Select Where( 年齢 < 15 & 性別 == "F" );
```

現在除外されている行、表示されていない行、またはラベル付きの行を選択するには:

```
dt<<Select Excluded;
dt<<Select Hidden;
dt<<Select Labeled;
```

除外されていない行、表示されている行、またはラベルがない行を選ぶには、選択メッセージと選択逆転メッセージの両方を同じステートメントに入れるか、またはこの2つのメッセージを続けて送ります。

```
dt<<Select Hidden<<Invert Row Selection;
dt<<Select Hidden;
dt<<Invert Row Selection;
```

特定のセルを参照するには、そのセルの行番号に添え字を割り当てます。次の例では、「**体重 (ポンド)**」列に添え字 [1] が使われ、計算式によって「**体重 (ポンド)**」列の最初の値に対する各「**身長 (インチ)**」の比率が算出されます。

```
New Column("比率", Formula(Name("身長 (インチ)") / Name("体重 (ポンド)"))[1]));
```

無作為に選択した行を取得するには、次のように指定します。

```
dt<<Select Randomly(number)
dt<<Select Randomly(probability)
```

条件付き確率を使って、必要な度数を選択しています。

JSL には、行メニューコマンド **Select Matching Cells** もあります。

```
dt << Select Matching Cells; //現在のデータテーブルの中で一致するセルを選択する
dt << Select All Matching Cells; //開いているすべてのデータテーブルの中で一致するセルを選択する
```

複雑な選択を行う場合や、選択を行の属性データとして永続的に格納する方法については、「行の属性と演算子」(321 ページ) を参照してください。

行の検索

Get Rows は、指定の条件に一致する行を行列の形で戻します。次の例は、年齢が 16 歳以上の行を選択します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Get Rows Where(:年齢>=16);
   [35, 36, 37, 38, 39, 40]
```

Get Selected Rows は、現在選択されている行を行列の形で戻します。次の列は、1、3、5、7 行目を選択し、行番号を行列の形で戻します。

```
dt << Select Rows({1, 3, 5, 7});
dt << Get Selected Rows;
   [1, 3, 5, 7]
```

選択されている行の検索

Next Selected と Previous Selected は、画面の外にある次の選択行を表示するために、データテーブルウィンドウを上下にスクロールします。テーブルは循環するので、Next Selected では、選択行の最後に来ると、最初の選択行に移動します。Previous Selected ではそれとは**逆に**先頭の選択行から最後の選択行に移動します。

```
dt << Next Selected;
dt << Previous Selected;
```

選択されている行のクリア

選択を取り消して、選択されている行がない状態にするには、次のように Clear Select を使います。

```
dt << Clear Select;
```

行の移動

次のコマンドは、現在選択されている行を指定した移動先に移動します。

```
dt << Move Rows(AtStart);
dt << Move Rows(AtEnd);
dt << Move Rows(After(rowNumber));
```

行に色とマーカーを割り当てる

Colors メッセージと Markers メッセージを使うと、行に使われる色とマーカーを割り当てたり変更したりできます。通常、これらの設定は、データテーブルから生成されるグラフに影響します。どちらのメッセージも、どの色またはマーカーを使うかを指定する数値の引数をとります。色とマーカーの番号については、「色とマーカー」（331 ページ）を参照してください。

```
dt << Colors(3);  // 選択した行に赤色のマーカーを設定する
dt << Markers(2); // 選択した行に X マーカーを設定する
```

他の行メッセージと同様、行を選択するメッセージもその他のメッセージと一緒に使うことができます。

```
dt << Select Where( 年齢 ==13)  // 最年少の被験者を選択する
    << colors(8) << markers(8);  // そして、それらに紫色の円のマーカーを設定する
```

Color by Column と Marker by Column は、指定された列の値に基づいて、それぞれ色とマーカーを設定します。

```
dt << Color by Column(: 年齢 );
dt << Marker by Column(: 年齢 );
```

その他の名前付き引数は次のとおりです。

- Continuous Scale （Color by Column のみ） 指定の列の値に従って、グラデーションになった色を割り当てます。
- Reverse Scale 使用中の色を反転させます。

- Make Window with Legend　凡例のウィンドウを別に作成します。
- Excluded Rows　除外されている列に行の属性を適用します。
- Marker Theme　マーカーの種類を指定します。
- Color Theme　カラーテーマを指定します。

セルの色

データグリッドの個々のセルを色分けできます。たとえば、次の行は行の属性の色を使ってセルを色分けします。

```
dt << Color Rows by Row State;
```

カテゴリの列、連続尺度の列のいずれもカラーテーマを指定することができます。

```
:Name("身長(インチ)") << Set Property(
   "Color Gradient",
   {"White to Blue", Range( 40, 80 )}
);
:Name("身長(インチ)") << Color Cell By Value( 1 );

:年齢 << Set Property(
   "Value Colors",
   {12 = Red, 13 = Yellow, 14 = Green, 15 = Blue, 16 =
   Magenta, 17 = Gray}
);

:年齢 << Color Cell By Value( 0 ); // セルの色分けを無効にする
```

特定のセルに色をつけることもできます。次の例は、「名前」列の1、5、8行目を赤に設定します。

```
:名前 << Color Cells(red, {1, 5, 8});
```

特定のセルの色を削除するには、色を黒に設定します。次の例は、「名前」列の1行目の色を削除します。

```
:名前 << Color Cells( black, {1} );
```

セルに、値に応じて色をつけることができます。次の例は、**「身長(インチ)」**列のセルに色をつけます。値が60より大きいセルは青、値が60以下のセルは紫に設定します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
For(i=1, i<=N Row(dt),i++,
   If(Column (dt, "身長(インチ)")[i]>60,
   Column(dt, "身長(インチ)") << Color Cells(5,{i}),
   Column(dt, "身長(インチ)") << Color Cells(8,{i})
   );
);
```

注：Color Cellsの最初の引数は、色の値を表します。2番目の引数には、行番号を含めます。

行の非表示、除外、ラベル

注：行の属性演算子を使って行を非表示にしたり、除外したり、ラベルをつけたりする方法について詳しくは、「行の属性と演算子」（321 ページ）を参照してください。

行を非表示にしたり、除外したり、ラベルをつけたりするには、Hide、Exclude、Label メッセージを使用します。これらのメッセージは切り替え式なので、1 度送るとオンになり、もう 1 度送るとオフになります。

たとえば、「Big Class」内の「年齢」が 13 を超えるすべての行に非表示の属性を設定するには、次のようにします。

```
dt << Select Where( 年齢 > 13 );
dt << Hide;
```

同じオブジェクトへのメッセージは、単一のステートメントにまとめることができるので、次のようにしても同じ結果が得られます。

```
dt << Select Where( 年齢 > 13 ) << Hide;
```

テーブルの行に対する反復

JSL には、組み込みの反復用プログラミング演算子の他に、データテーブルの行、グループ、または行の条件付き選択を使って反復するための演算子もあります。

通常、式はデータテーブルの現在の行に対してだけ実行されます。例外は、計算式列内の式、Summarize 演算子と統計量を事前計算する演算子、および分析プラットフォームがデータテーブルの列を使用する場合です。

現在の行の設定

注：スクリプトの現在行は、データテーブルで選択（強調表示）されている行や、データテーブルウィンドウでの現在のカーソル位置とは**関係ありません**。スクリプト操作の対象となる行の番号です。デフォルトではゼロ（行なし）になっています。

スクリプトのために現在の行を設定するには、For Each Row または Row()=X を使用します。

```
Row()=3; ...
For Each Row(...);
```

For Each Row は、現在のデータテーブルの各行に対し、スクリプトを 1 回実行します。Row()=1 はスクリプトが実行されている間だけ有効で、実行が終了すると、Row() はデフォルトの値であるゼロに戻ります。そのため、一度にスクリプトすべてを実行したときと、一度に数行ずつスクリプトを実行したときとでは、異なる結果になる場合があることに注意してください。

この章では、現在の行を明示的に指定していない例の場合、現在の行を決定するコンテキストの中だけで実行されるものとします。詳細は、「現在の行とは」（319 ページ）を参照してください。

現在の行とは

デフォルトでは、現在の行の番号は0です。テーブル内の先頭の行が行1なので、行0は原則として行がないことを表します。つまり、**デフォルトでは、操作はどの行にも実行されません。**現在の行を設定するか、または行のセットを指定しない限り、データがないために欠測値となります。たとえば、列名は、現在の行の該当する列の値を戻します。あいまいさを避けるために（強制的に名前を列名として解釈させるために）、接頭演算子「:」を使います。

```
:性別; // "" を戻す
:年齢; // . を戻す
```

接頭演算子を使うことで、実際には1行だけに基づいているのに、データテーブル全体に当てはまるように見える結果を取得しないようにするためです。また、大部分の環境で、値をあいまいな名前に割り当てた場合に、不用意にデータ値が上書きされないようにします （このような事態を完全に防ぐには、接頭または間に入れる演算子「:」を使ってデータ列を明確に参照し、接頭演算子「::」を使ってグローバルスクリプト変数を明確に参照します。詳細は、「プログラミング手法」の章の「高度な適用範囲指定と名前空間」（218ページ）を参照してください。

Row() 演算子を使って、現在の行の番号を取得または設定できます。Row() は、JSL の **L-value** 式の一例です。この演算子は、代入演算子（=、+= など）の前に置いて値を設定しない限り、デフォルトの値を戻します。

```
Row();      // 現在の行の番号を戻す（デフォルトでは 0）
x=Row();    // 現在の行の番号を x に格納する
Row()=7;    // 7 番目の行を現在の行にする
Row() = 7; :年齢;    // 7 番目の行を現在の行にし、12 を戻す
```

現在の行の設定は、ユーザが選択して実行するスクリプト部分が終わるまで有効です。スクリプトの実行が終わると、現在の行の設定はデフォルト（行0または行なし）にリセットされます。そのため、同じスクリプトでも、一度にすべて実行した場合と数行ずつ実行した場合では、異なる結果が出ることがあります。

行と列の数

演算子N Rows とN Cols は、データテーブル内の行と列の数をカウントします。

```
N Rows(dt); // 行数
N Cols(dt); // 列数
```

N Rows とN Cols は、行列内の行の数もカウントします。NRow とNCol は同義語です。詳細については、「データ構造」の章の「問い合わせ関数」（159ページ）を参照してください。

各行に対してスクリプトを反復する

現在のデータテーブルの各行にスクリプトを繰り返すには、スクリプトにFor Each Row を入れます。

```
For Each Row( If(:年齢>15, Show(:年齢)) );
```

For Each Row は、データテーブルに新しい計算式列を作成するのではなく、行の属性を設定するために使います。次の2つのスクリプトは似ていますが、最初のスクリプトでは行の属性列を作成し、For Each Row のスクリプトでは列を作成せずに行の属性を設定します。

```
New Column("行の属性", Row State, Formula( Color State(年齢 -9)) );
For Each Row(Color of(Row State()) = 年齢 -9);
```

指定した条件に一致する各行にスクリプトを繰り返すには、次のように For Each Row と If を組み合わせます。

```
For Each Row(Marker of(Row State()) = If(性別 =="F",2,6));
```

For Each Row ループの実行を制御するには、Break と Continue を使用します。詳細は、「JSL の構成要素」の章の「Break および Continue」（102 ページ）を参照してください。

行の値を戻す

Dif と Lag は、統計量計算、特に時系列データや累積データを処理するときに有効です。

- Lag は、現在の行から *n* 行前の列の値を戻します。
- Dif は、現在の行の列の値と *n* 行前のその列の値との差を戻します。

以下の式は同じ結果になります。

```
dt << new column("身長の差");
For Each Row( :身長の差 =:Name("身長 ( インチ )")-lag(:Name("身長 ( インチ )"),1) );
For Each Row( :身長の差 =dif(:Name("身長 ( インチ )"),1) );
```

シーケンスデータの追加

Sequence() は、計算式エディタの Sequence 関数に相当し、テータテーブル列のセルを埋めるのに使用します。4 つの引数のうち、後の 2 つはオプションです。

```
Sequence(from, to, stepsize, repeat)
```

必須である開始値（from）と終了値（to）は、セルに挿入する値の範囲を指定します。from=4、to=8 とした場合、セルは 4、5、6、7、8、4、... という値で埋められます。

ステップサイズ（stepsize）はオプションです。指定しない場合のデフォルト値は 1 です。ステップサイズは範囲内の値の増分です。上記の from と to の値を使い、stepsize = 2 とした場合、セルは 4、6、8、4、6、... という値で埋められます。

繰り返し（repeat）はオプションです。指定しない場合のデフォルト値は 1 です。繰り返しは、次の値にインクリメントするまでに同じ値を繰り返す回数を指定します。上記の from、to および stepsize の値を使い、repeat=3 とした場合、セルは 4、4、4、6、6、6、8、8、8、4、... という値で埋められます。繰り返し値（repeat）を指定した場合は、ステップサイズ（stepsize）も指定しなければなりません。

列のすべての行が埋まるまで数列（シーケンス）が繰り返されます。

例：

```
// 新しいデータテーブルを作成
dt = New Table("Sequence の例");
```

```
// 2 列、50 行を追加
dt << New Column("5 まで数える ");
dt << New Column("4 間隔で 17 まで数える "); dt << Add Rows (50);

/* 最初の列を 1、2、3、4、5、... の数列で埋める
2 列目を 1、1、5、5、9、9、13、13、17, 17、... の数列で埋める */

For Each Row (
   column(1)[ ] = Sequence(1,5);
   column(2)[ ] = Sequence(1,17, 4, 2);
);
```

Sequence() は計算式関数なので、列の計算式に Sequence() を設定して列を埋めることもできます。次の例では「Sequence 関数の計算式」という新しい列を作成し、それに計算式を追加しています。この計算式によって作成される数列は、25 から 29 までの値をそれぞれ回繰り返しながら、1 ずつ増加します（25、25、26、26、27、27、28、28、29、29、25、...）。

```
dt << New Column("Sequence 関数の計算式 ", formula(Sequence(25, 29, 1, 2)));
```

Sequence() の結果例をさらにいくつか紹介しましょう。

- Sequence(1,5) の場合、1,2,3,4,5,1,2,3,4,5,1,... となります。
- Sequence(1,5,1,2) の場合、1,1,2,2,3,3,4,4,5,5,1,1,... となります。
- Sequence(10,50,10) の場合、10,20,30,40,50,10,... となります。
- 10*Sequence(1,5,1) の場合、10,20,30,40,50,10,... となります。
- Sequence(1,6,2) の場合、1,3,5,1,3,5,... となります。リミットはありません。

注：通し番号の値で行列を作成したい場合は、Sequence 関数ではなく、Index 関数を使用します。

行の属性と演算子

データテーブルには、さまざまな属性を格納する「行の属性」という特殊なデータ要素があります。行の属性は、次のような属性を格納します。

- 行が選択されているか、除外されているか、非表示になっているか、ラベルがついているか
- マーカーの種類、色、色合い、グラフ上での色相

JSL では、行の属性演算子を使って行の属性を操作することができます。

行の属性について

行の属性に応じて、JMP による処理の内容が変わってきます。表 9.3 で各属性について説明します。複数の行の属性を組み合わせて使用することができます。

表 9.3 行の属性

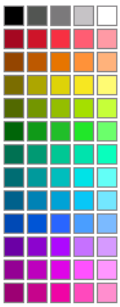
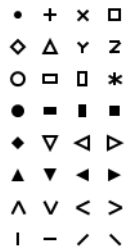
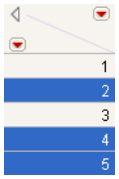
行の属性	結果への影響
除外する 	行が除外されている場合、JMP はそれらの行を統計分析（テキストレポートとチャート）の計算から除く。結果は、データが入力されていないときと同じになります。ただし、その行は**プロット**内に点として表示されます。（プロットから点を除くには「表示しない」を設定します。すべての結果から行を除外するには、「除外する」と「表示しない」の両方を指定します。）
表示しない 	行が「表示しない」になっている場合、JMP はそれらの行をプロットに表示しない。ただし、行はまだテキストレポートおよび**チャート**内に含まれています。（レポートおよびチャートから行を除くには「除外する」を指定します。すべての結果から行を除くには、「除外する」と「表示しない」の両方を指定します。）
ラベルあり 	行にラベルがついている場合、JMP は、散布図内の点に、行番号のラベルまたは指定されたラベル列の値を配置する。
色 	行に色が指定されている場合、JMP はそれらの色を使って、散布図内の点をわかりやすく表示する。
マーカー 	行にマーカーが指定されている場合、JMP はそれらのマーカーを使って、散布図と回転プロット内の点をわかりやすく表示する。

表 9.3 行の属性（続き）

行の属性	結果への影響
選択 	行が選択されている場合、JMP はプロットおよびチャート内の対応する点や棒を強調表示する。

行の属性の演算子について

行の属性の演算子に引数として数値（または数値を導き出す式）を指定すると、演算子はその数値を可能な値へのインデックスと解釈します。

表 9.4 は、行の属性演算子を比較した表です。この表を見ると、行の属性から数値に、または数値から行の属性に変換する演算子がわかります。表には、各演算子で使用できる数値も記載されています。

表 9.4 行の属性の演算子

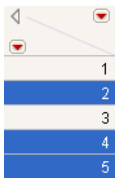
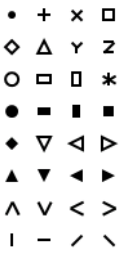
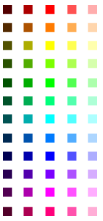
数	数値から行の属性への変換 →	行の属性	行の属性から数値への変換 →	数
1 または 0	`Excluded State(n)`	除外する 	`Excluded(rowstate)`	1 または 0
1 または 0	`Hidden State(n)`	表示しない 	`Hidden(rowstate)`	1 または 0
1 または 0	`Labeled State(n)`	ラベルあり 	`Labeled(rowstate)`	1 または 0
1 または 0	`Selected State(n)`	選択 	`Selected(rowstate)`	1 または 0

表 9.4 行の属性の演算子（続き）

数	数値から行の属性への変換 →	行の属性	行の属性から数値への変換 →	数
0 ～ 31	Marker State(*n*)	マーカー 	Marker Of(*rowstate*)	0 ～ 31
0 ～ 84 （0-15 は基本色、16-31 は暗い、32-47 は明るい、48-63 は非常に暗い、64-79 は非常に明るい、80-84 は灰色）	Color State(*n*)	色 	Color Of(*rowstate*)	0 ～ 84 （0-15 は基本色、16-31 は暗い、32-47 は明るい、48-63 は非常に暗い、64-79 は非常に明るい、80-84 は灰色）
0 ～ 11 （虹の色の順）	Hue State(*n*)	色相 		
-2 ～ 2 （暗い～明るい）	Shade State(*n*)	色合い 		

行の属性の割り当て

JSLを使って行の属性を割り当てるには、Select Whereを使って対象となる行を指定してから、それらの行に割り当てる属性を指定します。次の例は、性別が「F」の行にマーカーの種類「5」を割り当てます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :性別 == "F" ) << Markers( 5 );
```

計算式を使って行の属性を割り当てることもできます。それをデータテーブルで行う方法を紹介します。

1. 行の属性列を作成します。
2. 列に、性別が「F」である行にマーカーを割り当てる計算式を追加します。
3. 「列」パネルで、行の属性列の隣にある星を右クリックし、［**行の属性へコピー**］を選択します。

この手順をJSLで表現すると、次のようになります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "行の属性",
   Row State,
   Set Formula( If( :性別 == "F", Marker State( 4 ) ) )
);
Column( "行の属性" ) << Copy To Row States();
```

行の属性情報を格納

行の属性の列は、行の属性を情報として**格納**するだけで、属性を実際に有効にするわけではありません。行の属性の列を有効にするには、続いてFor Each Rowを使用します。行の属性の列について詳しくは、『JMPの使用法』を参照してください。

次の例は、行の属性の列を作成し、それらの属性を有効にします。

1. 次のスクリプトを実行、新しいデータテーブルの作成を開始します。

   ```
   dt = New Table( "行の属性テスト" );
   ```

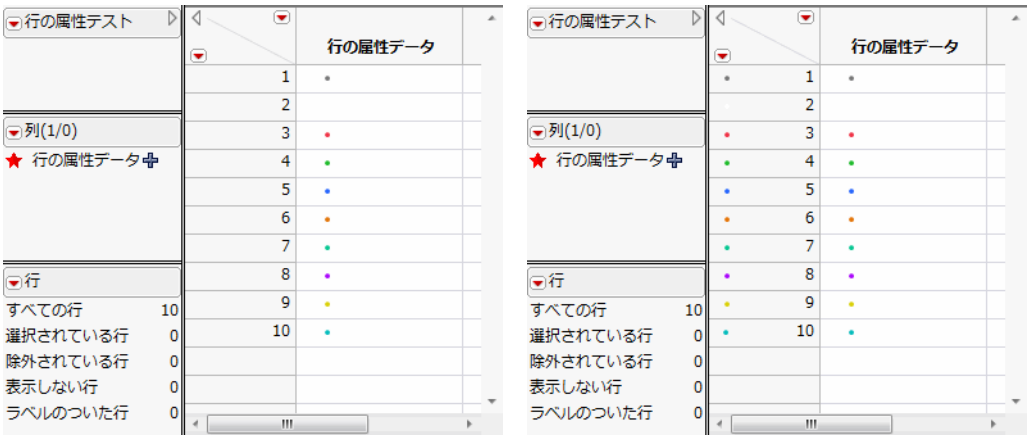
2. 次のスクリプトを実行して列に行の属性を追加し、10行を追加します。

   ```
   dt << New Column( "行の属性データ", Row State, Set Formula( Color State( Row() ) )
     );
   dt << Add Rows(10);
   ```

3. 次のスクリプトを実行し、行の属性を有効にします。

   ```
   For Each Row( Row State() = :行の属性データ );
   ```

図 9.6 行の属性のないテーブル（左）と行の属性のあるテーブル（右）



このスクリプトは、現在有効な行の属性を、行の属性の列に設定された組み合わせで置き換えます。他の属性を変更したり取り消したりしないで、行の属性のうち選択したものだけを変える方法については、「1 つの特性だけを設定し、他の特性をキャンセルする」（329 ページ）を参照してください。

行の属性の設定または取得

Row State 演算子を使うと、JSL から直接、行の属性を設定または取得できます。行 *n* の属性は、Row State(n) で設定または取得できます。引数を指定しなければ、現在の行を設定または取得します。すべての行を処理するには、For Each Row ループを使うか、計算式列で処理します。

行の属性を設定するには、行の属性に式を割り当て、式の左辺（L-value）に置きます。

```
Row State( 1 ) = Color State( 3 ); // 行 1 を赤にする
Row() = 8; Row State() = Color State( 3 ); // 8 番目の行を赤にする
For Each Row( Row State() = Color State( 3 ) ); // 各行を赤にする
```

行の属性を取得するには、行の属性式を割り当て式の右辺に置きます。

```
x = Row State( 5 );  // 行 5 の行の属性を x に保存する
x = Row State();   // 現在の行の行の属性
```

行の属性の設定に関するメモ

Row State() のすべての属性を設定するのか、Color Of(Row State()) のように特定の 1 つの属性だけを設定するのかという点に注意してください。動作を確かめるには、まず、すべての行に色とマーカーの属性を設定します。

```
For Each Row(
   Row State() = Combine States( Color State(Row() ), Marker State(Row())));
```

ここで、行の属性のうち 1 つの属性だけを変更してみましょう。

```
Color Of( Row State( 1 ) )=3; // マーカーを変更しないで行 1 を赤にする
```

行の属性のすべての要素をクリアして1つの属性を設定するには、次のようにします。

```
Row State( 1 ) = Color State( 5 ); // 行1を青にし、そのマーカーを削除する
```

現在の行の属性をすべて行属性列にコピーするには、次のようにします。

```
New Column( "rscol", Set Formula(Row State() );
For Each Row( rscol = Row State() );
```

現在の行の属性をすべてではなく一部（複数）だけ行属性列にコピーするには、次のようなスクリプトを使います（必要のない属性は、コメントにするか削除してください）。

```
New Column( "rscol2",
   Set Formula(
      Combine States(
         Color State( Color Of() ),
         Excluded State( Excluded() ),
         Hidden State( Hidden() ),
         Labeled State( Labeled() ),
         Marker State( Marker Of() ),
         Selected State( Selected() )
      );
   );
);
```

行の属性のうち1つの特性を設定するには:

```
Color Of( Row State(i) ) = 3; // 行 i の色を赤に変更
Selected( Row State(i) ) = 1; // 行 i の行の属性を選択し、1に設定
```

上の例のように、演算子の中には、数値を属性に変換するものと、属性を数値に変換するものがあります。このどちらであるかを判別するためのヒントを示します。

数値を属性に変換する演算子には、「State」という語が含まれています。 数値引数をとり、属性を戻すかまたは属性割り当てを受け取る演算子はすべて、次のように、名前に「State」という語が含まれています。`Row State`、`As Row State`、`Color State`、`Combine States`、`Excluded State`、`Hidden State`、`Hue State`、`Labeled State`、`Marker State`、`Selected State`、`Shade State`。

属性を数値に変換する演算子は、1語か、または「Of」という語が含まれています。 行の属性引数をとる（引数が指定されていない場合は、`Row State()` が引数とみなされる）演算子と、数値を戻すか数値に設定されている演算子は、次のように、1語であるか、または2番目の語が「Of」です。`Color Of`、`Excluded`、`Hidden`、`Labeled`、`Marker Of`、`Selected`。

表9.4（323ページ）は、これらの演算子の違いを説明した便利な表です。

次のコマンドは、インタラクティブなコマンドと同じものです。

```
Copy From Row States
Add From Row States
Copy To Row States
Add To Row States
```

行の属性を取得する例

たとえば、行の属性を含むデータテーブルを作成してみましょう。

```
dt = New Table( "行の属性テスト" );
dt << New Column( "行の属性データ", Row State, Set Formula( Color State( Row() ) )
  );
dt << Add Rows(10);
For Each Row( Row State() = :行の属性データ );
```

行の属性を取得するには、次のようなスクリプトを実行します。

```
Row State(1); // 行 1 の行の属性を戻す
   Color State(1)

Row() = 8; Row State();  // 現在の行（8 番目の行）の行の属性を戻す
   Color State(8)

For Each Row( Print( Row State()) ); // 各行の行の属性を戻す
   Color State(1)
   Color State(2)
   Color State(3)
   Color State(4)
   Color State(5)
   Color State(6)
   Color State(7)
   Color State(8)
   Color State(9)
   Color State(10)
```

行の属性を取得し、グローバル変数に格納するには、次のように割り当て式の右辺に Row State() を置きます。

```
::x = Row State(1); // 行 1 の行の属性をグローバル変数 x に保存する

Row() = 8; ::x = Row State();  // 8 番目の行の行の属性を x に保存する
Show(x);
   x = Color State(8)

dt << New Column( "rsco l", Row State) ;
For Each Row( :rscol = Row State() );
// 行の属性を、「rscol」（行の属性の列）に保存する
```

行の属性のうち 1 つの特性だけを取得するには、行の属性式を割り当て式の右辺に置き、左辺に指定できる演算子のどれかを使います。

```
x = Selected( Row State() ); // 現在の行が選択されているかどうかのインデックス
```

複数の属性を一度に取得／設定する

Combine States の中で属性の設定を組み合わせることで、多数の属性を一度に取得または設定できます。また、属性が組み合わされた後も、各属性ごとに取得または設定することが可能です。次の例は、「Big Class.jmp」サンプルデータテーブルを使用します。男子に緑のＹマーカーと非表示の設定をし、女子に赤のＸマーカーを設定します。

```
For Each Row(
   if( 性別 =="M",
      /*then*/ row state()=Combine States(
         Color State(4), Marker State(6), Hidden State(1)),
      /*else*/ row state()=Combine States(
         Color State(3), Marker State(2), Hidden State(0))));
```

1つの行、たとえば6行目の行の属性を取得します。

```
row state(6);
   Combine States(Hidden State(1), Color State(4), Marker State(6))
```

JMP は、Combine State で属性の組み合わせを戻すことに注意してください。これは、行の属性データは、色などの1つの属性だけではなく、除外、非表示、ラベル付け、選択、マーカー、色、色相、濃淡など、設定された**すべての**属性をまとめたものだからです。このような特性のリストは、**行の属性の組み合わせ**と呼ばれます。

複数の行の属性が有効になっている場合もあるので、行の属性列は複数の特性を値として持つことができるようになっています。

1 つの特性だけを設定し、他の特性をキャンセルする

行の属性のすべてを取得または設定するための全体的な Row State 演算子の他に、一度に1つの特性を優先的に取得または設定する演算子があります。つまり、1つの行に1つの特性だけを与えるため、他の特性をすべて取り消します。1つの特性を設定し、その他をキャンセルする演算子は、Color State、Combine States、Excluded State、Hidden State、Hue State、Labeled State、Marker State、Selected State、Shade State です。

たとえば、行4を表示しないという属性だけを設定するには、次のようにします。

```
Row State( 4 ) = Hidden State( 1 );
```

一度に 1 つの特性を設定または取得する

行の属性とは、1つの特性ではなく多数の特性が組み合わされたものです。一度に1つだけ処理するには、Row State と左辺に指定できる演算子のどれかを、取得か設定かによって等号のどちらかの側で使います。左辺に指定できる演算子として、Color Of、Excluded、Hidden、Labeled、Marker Of、Selected があります。

この例は、他の特性には影響を与えずに、行4を非表示にします。

```
Hidden( Row State( 4 ) ) = 1
```

この例は、他の特性には影響を与えずに、行3の色を格納します。

```
::color = Color Of( Row State( 3 ) );
```

行の属性の変化を特定する

(データテーブルオブジェクトに送られた) MakeRowStateHandler メッセージは、行の属性が変更されるとコールバックを取得します。例:

```
f = Function( {X}, Show( x ) );
obj = Current Data Table() << Make Row State Handler( f );
```

これで、棒グラフなどで行をグループ選択すると、行の属性が変更されている行の番号がログに送られます。たとえば、次のような設定が行えます。

```
x:[3, 4, 28, 40, 41]
```

複数行が強調表示されている場合は、選択がクリアされた行に対して1回、新しく選択された行に対して1回の計2回、ハンドラが呼び出されることがあります。

除外、非表示、ラベル、選択

この節では、オンまたはオフのブール値をとる状態について説明します。このような状態としては、行の除外、非表示、ラベル、選択があります。

- Excluded は、除外されているかどうか示す除外インデックスを取得または設定します。インデックスの値が1なら真、0なら偽です。
- Hidden は、非表示は1、表示は0の非表示インデックスを取得または設定します。
- Labeled は、ラベルがついているなら1、ついていないなら0のラベルインデックスを取得または設定します。
- Selected は、選択されている場合は 1、選択されていない場合は 0 の選択インデックスを取得または設定します。

次の例は、これらの状態を示します。

```
Excluded(Row State()); // 現在の行が除外されている場合は 1、除外されていない場合は 0 を戻す
Hidden();              // 現在の行が表示されない場合は 1、表示される場合は 0 を戻す
Labeled(Row State());  // 現在の行にラベルがついている場合は 1、// ラベルがついていない場合
   は 0 を戻す
Selected();            // 現在の行が選択されている場合は 1、選択されていない場合は 0 を戻す

Excluded(Row State())=1; // 現在の行を除外する
Hidden()=0;              // 現在の行を表示する
Labeled(Row State())=1;  // 現在の行にラベルをつける
Selected()=0;            // 現在の行の選択を取り消す
```

これらの関数では、引数が指定されていない場合は、Row State() が引数であるとみなされます。

Excluded State、Hidden State、Labeled State、Selected State は、逆に、引数に基づいて、行の属性の状態を真または偽として取得または設定します。ゼロ以外の値は行の属性を真に、ゼロの値は偽に設定します。欠測値の場合は、行の属性に変更はありません。

```
Row State()=Excluded State(1); // 現在の行の属性を「除外されている」に変更する
Row State()=Hidden State(0);   // 現在の行の属性を「再表示」に変更する
Row State()=Labeled State(1);  // 現在の行の属性を「ラベルがついている」に変更する
Row State()=Selected State(0); // 現在の行の属性を「選択されていない」に変更する
```

上記の最初の 2 つの式は、行の属性のうち除外または表示を置き換えるだけで、他の既存の行属性の特性は失われません。通常、設定したいすべての特性の state コマンドを Combine States の中に記述します。

```
Row State()=Combine States(Color State(4), Marker State(3), Hidden State(1));
// 現在の行を非表示にし、緑の正方形のマーカーに変更する
```

また、行の属性に（累積特性として）追加できる値を持つ行の属性データ列内で -State コマンドを使うのも一般的な方法です。次の例は、各奇数行を除外します。

```
dt << New Column("myExcl",Row State, set formula(Excluded
   State(Modulo(Row(),2))));
For Each Row(Row State()=Combine States(:myExcl, Row State()));
```

色とマーカー

この節では、多数の選択肢がある状態について説明します。このような状態には、色、マーカー、色相、色合いがあります。

Color Of は、色番号を戻すか、設定します。色番号とは、*row state* に対応する JMP カラーマップの色番号で、色が割り当てられていない場合は 0 となります。

```
Color Of(Row State());   // 現在の行の色番号を戻す
Color Of()=4;            // 現在の行を色 4 に設定する
```

同様に、Marker Of は、マーカー番号を戻すか、設定します。マーカー番号は、アクティブなマーカーに対応する JMP マーカーマップのマーカー番号で、マーカーが割り当てられていない場合は 0 となります。

```
Marker Of();               // 現在の行のマーカー番号を戻す
Marker Of(Row State())=4; // 現在の行をマーカー 4 に設定する
```

Color Of と Marker Of はどちらも、すべての行の属性の式もしくは列、または Row State() を引数として受け取り、引数が指定されていない場合は、引数 Row State() を引数とみなします。

Color State と Marker State は、逆方向に動作することを除くと、Color Of および Marker Of と同じです。-Of 関数は、実際の属性をインデックスに変え、-State 関数は、インデックスを属性に変えます。

```
Row State()=Color State(4);  // 現在の行を緑に変更する
Row State()=Marker State(4); // 現在の行をひし形のマーカーに変更する
```

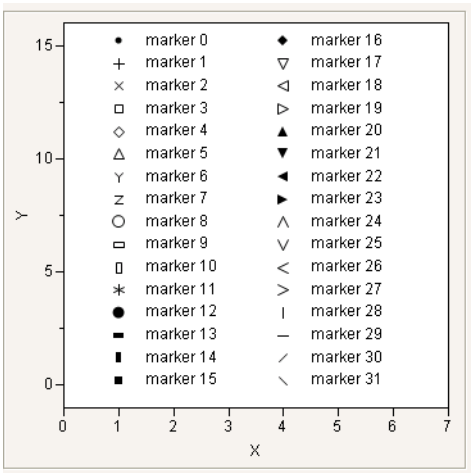
最後の 2 つのコマンドは、行の属性のうち色またはマーカーを置き換えるだけで、他の既存の特性は失われません。通常、設定したいすべての特性の -State コマンドを Combine States の中に記述します。

```
Row State()=Combine States(Color State(4), Marker State(3), Hidden State(1));
// 現在の行を非表示にし、緑の正方形のマーカーに変更する
```

次のスクリプトは、JMP の標準的なマーカー（0 ～ 31）を表示します。0 ～ 31 の範囲外にあるインデックスの動作は定義されていません。

```
New Window( " マーカー ",
   Graph Box(
      FrameSize( 300, 300 ),
      Y Scale( -1, 16 ),
      X Scale( 0, 7 ),
      For(
         i = 0;
         jj = 15;,
         i < 16;
         jj >= 0;,
         i++;
         jj--;, // 16 行、2 列
         Marker Size( 3 );
         Marker( i, {1, jj + .2} ); // マーカー 0 ～ 15
         Marker( i + 16, {4, jj + .2} ); // マーカー 16 ～ 31
         Text( {1.5, jj}, "marker ", i ); // マーカーラベル 0 ～ 15
         Text( {4.5, jj}, "marker ", i + 16 ); // マーカーラベル 16 ～ 31
      );
   );
);
```

図 9.7 JMP マーカー



ヒント：このスクリプトについて詳しくは、「表示ツリー」（377 ページ）の章を参照してください。

色については、以下の点に留意してください。

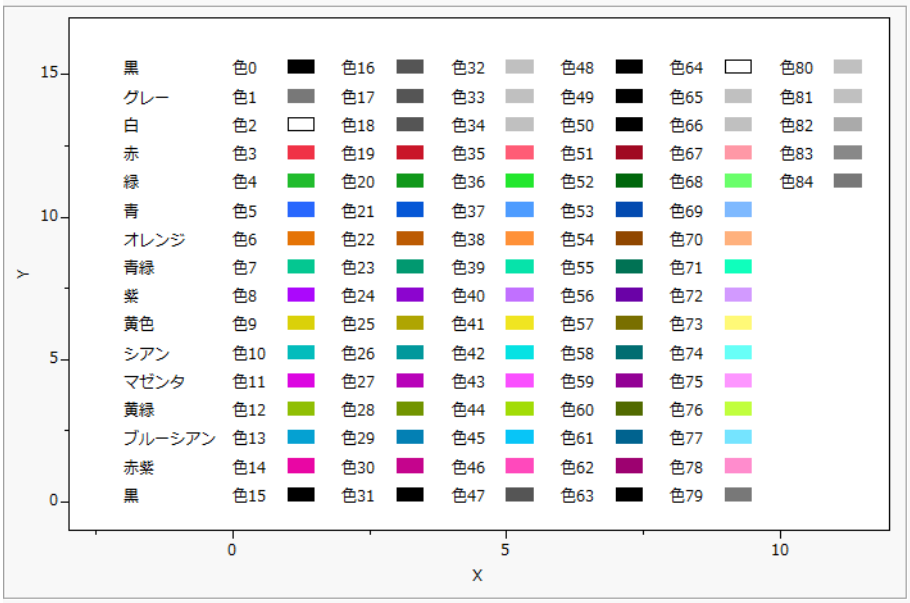
- JMP カラーには 0～84 の番号が付いています。
- 最初の 16 個は基本色です。下のスクリプトを参照してください。
- 17 以降は、それぞれの基本色の色合いを暗め、または明るめにしたものです。
- 0～84 の範囲外にあるインデックスの動作は定義されていません。
- JMP カラーの使用方法については、「スクリプトによるグラフ作成」の章の「色」（487 ページ）を参照してください。

次のスクリプトは、JMP の標準の色を表示します。

```
Text Color( 0 );
New Window( "色",
   Graph Box(
      FrameSize( 640, 400 ),
      Y Scale( -1, 17 ),
      X Scale( -3, 12 ),
      k = 0;
      For( jj = 1, jj <= 12, jj += 2,
         l = 15;
         For( i = 0, i <= 15 & k < 85, i++,
            thiscolor = Color To RGB( k );
            Fill Color( k );
            thisfill = 1;
            If( thiscolor == {1, 1, 1},
               Pen Color( 0 );
               thisfill = 0;
            ,
               Pen Color( k )
            );
            Rect( jj, l + .5, jj + .5, l, thisfill );
            Text( {jj - 1, l}, "色", k );
            k++;
            l--;
         );
      );
      jj = -2;
      color = {"黒", "グレー", "白", "赤", "緑", "青", "オレンジ", "青緑",
      "紫", "黄色", "シアン", "マゼンタ", "黄緑", "ブルーシアン", "赤紫", "黒"};
      For(
         i = 0;
         l = 15;, i <= 15 & l >= 0,
         i++;
         l--;,
         Text( {jj, l}, color[i + 1] )
      );
```

```
    );
);
```

図9.8 JMPカラー



RGB値を使うには、赤、緑、青の順でそれぞれの含有率をリストして、各色を指定する必要があります。たとえば、次のように指定すると、緑がかった青になります。

```
pen color({.38,.84,.67}); // 小鴨色
```

色相と色合いの例

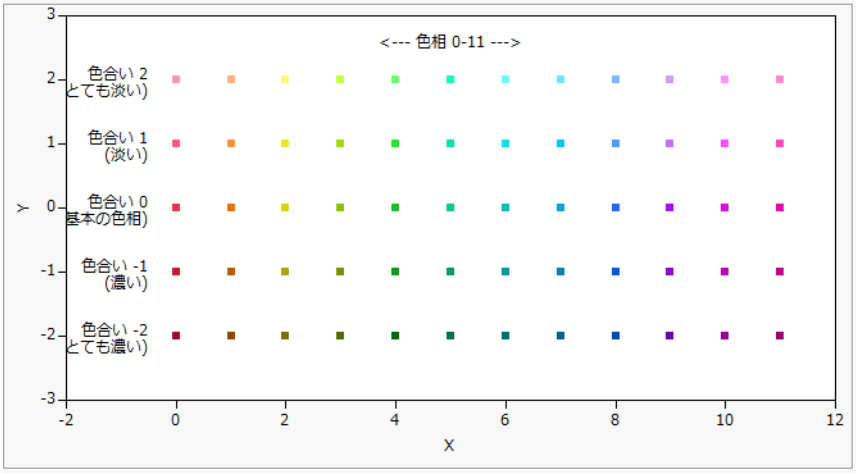
色を選ぶ際、Color Stateを使う代わりにHue StateとShade Stateを一緒に使うこともできます。Hue Stateを使った場合は、黒、白、またはグレーの色合いは選べません。このような場合は、Shade Stateを単独で使うか、Color Stateを使う必要があります。

次のスクリプトは、色相および色合いの値と色との関係を表示します。

```
New Window( "色相と色合い",
   Graph Box(
      FrameSize( 600, 300 ),
      Y Scale( -3, 3 ),
      X Scale( -2, 12 ),
      k = 0;
      For( h = 0, h < 12, h++,
         For( s = -2, s < 3, s++,
            myMk = Combine States( Hue State( h ), Shade State( s ), Marker State(
   15 ) );
```

```
                Marker Size( 3 );
                Marker( myMk, {h, s} );
             );
          );
          Text( Center Justified, {5, 2.5}, " <--- 色相 0-11 ---> " );
          Text( 右揃え ,
             {-.5, -2}, "色合い -2", {-.5, -2.25}, "(とても濃い)",
             {-.5, -1}, "色合い -1", {-.5, -1.25}, "(濃い)",
             {-.5, 0}, "色合い 0", {-.5, -.25}, "(基本の色相)",
             {-.5, 1}, "色合い 1", {-.5, .75}, "(淡い)",
             {-.5, 2}, "色合い 2", {-.5, 1.75}, "(とても淡い)",
          );
       );
    );
```

図9.9 色相と色合い



Hue と Shade には、-Of 演算子はありません。Color Of は、Hue State と Shade State、またはそのどちらかで設定された色の行属性について、Color State と同じ番号を戻します。たとえば、行4と行5に同じ濃赤色のマーカーを設定します。

```
Row State( 4 ) = Combine States( Hue State( 0 ), Shade State( -1 ), Marker State(
   12 ) );
Row State( 5 ) = Combine States(
   Color State( Color Of( Row State( 4 ) ) ),
   Marker State( Marker Of( Row State( 4 ) ) )
);
```

行の属性と行列の例

次の例では、行の属性の値は事前に準備され、座標の情報とともに Marker ルーチンに渡されます。

```
// 次にシミュレートされるような CP および CA データを仮定
dt = New Table( "CP および CA データの例 ",
   Add Rows( 26 ),
   New Column( "cover_cp",
      Numeric,
      Continuous,
      Formula( Random Uniform() / 100 + 0.94 )
   ),
   New Column( "cover_ca",
      Numeric,
      Continuous,
      Formula( Random Uniform() * 0.04 + 0.94 )
   ),
   New Column( "p", Numeric, Continuous, Formula( Random Uniform() ) ) );
);
dt << Run Formulas;
greenMark = Combine States( Marker State( 2 ), Color State( 4 ) );
redDiamond = Combine States( Marker State( 3 ), Color State( 3 ) );
New Window( "CP および CA 比較 ",
   Graph Box(
      title( "CP および CA 比較 " ),
      FrameSize( 400, 350 ),
      X Scale( 0, 1 ),
      Y Scale( 0.94, 1 ),
      For Each Row(
         Marker( greenMark, {p, cover_cp} );
         Marker( redDiamond, {p, cover_ca} );
      );
   );
);
```

行の属性の内部番号

この節では、内部の数値コードを使って行の属性を処理する**上級ユーザ向けのオプション**について説明します。

行の属性とは、データテーブル内のすべての行が持っている 6 つの属性の総称です。これら 6 つの属性は、内部で単一の数値にまとめられています 必要に応じて、行の属性の内部コードを表示できます。行の属性を列にコピーし、列のタイプを数値に変更するだけで、JMP で使われている数値を表示できます。

内部の数値コードと `As Row State` 演算子を使って、整数を等価の行の属性に変換するだけで、行の属性を割り当てることもできます。たとえば、行番号に基づいて行の属性を割り当てるには、次のようにします。

```
For Each Row(Row State()=as row state(row()));
```

また、`Set Row States` コマンドを使うと、コードを行列にして実行し、一度に行の属性すべてを割り当てることもできます。この行列には、1 行単位の次元があり、各行に 1 つの値が含まれている必要があります。値は、その行に必要な属性に対応する行の属性コードです。

ただし、このような行の属性は、あまり使われません。実際のアプリケーションでは、数値と行の属性の関連の仕方を理解することが重要です。簡単に説明すると、ある行の属性 *r*（以下の例では、3 番目の行の行属性）の行の属性コードは次の計算式で算出されます。

```
r=Row State(3);
rscode=selected(r)+
   2*excluded(r)+
   4*hidden(r)+
   8*labeled(r)+
   16*marker of(r)+
   256*color of(r);
```

例

この例は、この方法を利用して、女子を計算から除外し、男子をプロットで表示しないようにし、女子と男子を X と Y で区別し、さらに、年齢ごとに異なる色を割り当てる簡潔な計算式を作成します。

論理演算子 == は、等しいことをテストする演算子で、真の場合は 1、偽の場合は 0 を戻します。

```
Open("$SAMPLE_DATA¥Big Class.jmp")

For Each Row(
   Row State() = as row state(
      (: 性別 =="F")*2 + // 女子を除外する
      (: 性別 =="M")*4 + // 男子を表示しない
      ((: 性別 =="F")*2+(: 性別 =="M")*6) * 16 +
         // 女子用のマーカーは 2、男子用のマーカーは 6
      (: 年齢 -11)*256));
         // 1 ～ 6 の色
```

データ値へのアクセス

データテーブル内の値を処理する場合、次のような手順が一般的です。

1. 使用したい値が入っているデータテーブルを、現在のデータテーブルとして設定する。すでにデータテーブルの参照がある場合は、その参照を使用することができます。「現在のデータテーブルの設定」（271 ページ）を参照してください。
2. 使用したい値が入っている行または列を指定し、使用したい値が含まれている列名を指定します。「列名による値の設定または取得」（338 ページ）を参照してください。

次の例では、「Big Class.jmp」データテーブルを開き（それにより、このデータテーブルが「現在のデータテーブル」になります）、「体重」列の行 2 を指定します。ログを見ると、123 という値が戻されていることがわかります。これは、行 2 の「Louise」の体重です。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt:weight[2];
   123
```

列名による値の設定または取得

列を参照する最も簡単な方法は、名前で参照することです。同名のグローバル変数と列がある場合は、混同を避けるため、接頭演算子「:」を使って列名の適用範囲を指定してください。

現在の行にあるセルの値を設定するには、列名と新しい値を指定します。次の例では、「Big Class.jmp」を使って、まず行 5 を現在の行にします。

```
Row() = 5;
dt:年齢 = 19; // 行 5 の「年齢」を 19 に設定する
dt:名前 = "Sam"; // 行 5 の「名前」を Sam に設定する
```

別の行にあるセルの値を設定するには、添え字で行番号を指定します。

```
dt:年齢[10] = 20; // 行 10 の「年齢」の値を 20 に設定する
```

何も含まれていない添え字は現在行を表すため、**:年齢[]**は**:年齢**と同じです。

現在の行にあるセルの値を取得するには、列名を指定します。次の例では、「Big Class.jmp」の行 16 が選択されているとします。

```
Row() = 16;
myGlobal = :年齢;
   14
:年齢;
   14
Show(:年齢);
   年齢 = 14;
```

特定の行にあるセルの値を取得するには、列名と行番号を指定します。次の 2 つの例は、どちらも「Big Class.jmp」の行 12 の「**年齢**」である「13」を戻します。

```
:年齢[12];
myGlobal = :年齢[12];
```

データ列の参照の値を取得するには、Column() と As Column() を使用します詳細は、「列の参照によるセル値へのアクセス」（295 ページ）を参照してください。

まとめ

- 行番号を指定しなかった場合は、現在の行が戻されます。
- 現在の行を参照するときは、:年齢[ ] のように、空白の添え字を使用します。
- 必ず、存在しているテーブルの行を指定してください。デフォルトの行番号はゼロなので、行ゼロを参照する :name というステートメントを指定すると、「無効な行番号です。」というエラーが表示されます。

データ値にアクセスするその他の方法

データテーブル、行、列は、他の方法で指定することもできます。これら3つの要素を1つの式で指定するには、次のように二項演算子「:」と添え字を使います。

```
dt:年齢[2] = 12;// テーブル、列、および行
```

複数の行を対象にする場合は、行番号のリストまたは行列で添え字を使用できます。

```
年齢[i] = 3;
年齢[{3, 12, 32}] = 14;
list = 年齢[{3, 12, 32}]; // 値をリストに入れる
vector = 年齢[1 :: 20]; // 値を行列に入れる
```

値の変更に関するメモ

データテーブル内の値を変更するたびに、最新の情報が表示されるように、メッセージが送られます。しかし、スクリプトに多数の変更箇所がある場合、この方法では更新を完了するのに時間がかかってしまいます。

変更速度を上げるには、変更する前に`Begin Data Update`を使って更新メッセージを止め、すべて変更し終わってから`End Data Update`を屋って更新メッセージを解放し、表示を更新します。

```
Current Data Table() << Begin Data Update;
...<変更作業>...
Current Data Table() << End Data Update;
```

必ず`End Data Update`メッセージを送ってください。そうしないと、何らかの方法で強制的に更新しない限り、表示は更新されません。

データテーブルへのメタデータの追加

データテーブルは、サブジェクトの特定のセットに関するさまざまな変数の計測値である観測データを格納します。ただし、JMPデータテーブルには、**メタデータ**、またはデータについてのデータも格納できます。メタデータには、次のようなものがあります。

- テーブル変数（メモなどのテキスト文字列）
- プロパティ（スクリプトなどの式）
- スクリプト
- 計算式

テーブル変数

テーブル変数は、「ノート」のような単一のテキスト文字列を保存するために使います。変数の働きを理解するために、まず、`Get Table Variable`メッセージを送って変数の既存の値を取得してみましょう。

```
dt=Open("$SAMPLE_DATA/Solubility.jmp");
dt << Get Table Variable("ノート");
   異なる溶剤で化合物の溶解度を測定したデータ。このテーブルには、6 つの水性 / 無極性システムの 72
      の有機溶質の LogP(partition coefficient) が表示されている。
```

次に、文字列の既存の値を、Set Table Variable または New Table Variable で変更し、再び Get Table Variable を使って文字列が更新されたことを確認します。

```
dt << Set Table Variable("ノート","化合物の溶解度");
```

または

```
dt << New Table Variable("ノート","化合物の溶解度");
dt << Get Table Variable("ノート");
   "化合物の溶解度"
```

次の例は、データテーブルに 2 つの新しいテーブル変数を追加します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
myvar = "これは、JMP Big Class サンプルデータのマイバージョンです。";
dt << Set Table Variable("key1", myvar);
dt << Set Table Variable("key2", myvar);
```

値を設定したとき、指定した変数が存在しない場合のみ、新しい変数が作成されます。同名のテーブル変数を 2 つ追加した場合、変数は 1 つしか作成されません。

テーブルスクリプト

JSL を使って、新しいスクリプトの追加、スクリプトの実行、スクリプトの取得ができます。

```
dt = Current Data Table();
dt << New Script("二変量", Bivariate(Y(:Name("体重 ( ポンド )")),X(:Name("身長 ( インチ
   )"))));
dt << Get Script("Bivariate");
   Bivariate(Y(:Name("体重 ( ポンド )")),X(:Name("身長 ( インチ )")))
dt << Set Property("二変量",Bivariate(Y(:Name("体重 ( ポンド )")),X(:Name("身長 ( イン
   チ )")),Fit Line));
dt << Run Script("Bivariate")
```

次の例は、新しいスクリプトを作成し、それを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Script(
   "新しいスクリプト",
   Distribution(
      Column(:Name("身長 ( インチ )"),:Name("体重 ( ポンド )")),
      By( :性別 )
   );
);
dt << Run Script( "一変量の分布" );
```

同僚に電子メールで送ったり、スクリプトの一部として使ったりするために、データテーブルをテキスト形式で表したものが必要な場合があります。Get Scriptを使うと、データテーブルの情報を再構成するスクリプトを取得できます。次の例は、「Big Class.jmp」を開き、データ、テーブル変数、列プロパティをログに出力します。その出力の一部をここに示します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
dt << Get Script;
   New Table( "Big Class",
      Add Rows( 40 ),
      New Script(
         ["en" => "Distribution",...],
         Distribution(
            Continuous Distribution( Column( :Name("体重(ポンド)") ) )
            Nominal Distribution( Column( :年齢 ) )
         );
      ),
```

注：New Script()またはSet Property()を使用してください。New Property()とNew Table Property()は廃止されました。

データテーブルを開いたときにスクリプトを自動的に実行する

On Openという名前のテーブルスクリプトは、データテーブルを開くと同時に自動的に実行されます。On Openスクリプトを指定するには、次のアクションのいずれかを実行します。

- **Save Script to Data Table**オプションを使用してスクリプトを作成した後、プロパティ名をダブルクリックし、名前をOn Openに変更します。
- New Scriptメッセージを使用してスクリプトを保存します。

次の例では、データテーブルを開いたときに*dt*が定義されない可能性があるため、Sortメッセージを、*dt*ではなくCurrent Data Table()に送ります。

```
dt << New Script("OnOpen", sortedDt=Current Data Table() << Sort( By(Name), Output
   Table Name ("並べ替えたBig Class")) );
```

システムの環境設定を使って、On Openスクリプトが自動実行されないようにすることができます。それには、次のように指定します。

```
preference(suppress On Open script eval(1));
```

予防措置として、他者から受け取ったデータテーブルを開く際はスクリプトが自動実行されないよう設定することをお勧めします。

計算式

Suppress Formula Eval メッセージは、自動評価を実行するかどうかを指定するブール値の引数をとります。データテーブルに多数の変更を加える必要があり、ステップ間で計算式が更新されるのを待ちたくない場合は、式の自動評価機能をオフにできます。

```
dt << Suppress Formula Eval(1);
dt << Suppress Formula Eval(0);
dt << Suppress Formula Eval;    // 現在の属性をオンまたはオフにします
```

すべてのデータテーブルで自動評価をさせないようにするには、Suppress Formula Eval コマンドを使い、全体にわたって自動評価をオフにします。このコマンドの機能は、前のメッセージと同じですが、データテーブルオブジェクトには送りません。

```
Suppress Formula Eval(1); // セッション内のすべての計算式を自動計算しない
Suppress Formula Eval(0); // セッション内のすべての計算式を自動計算する
```

計算式は、列に組み込まれているときは評価されません。強制的に評価しようとしても、今度はバックグラウンドで評価されます。スクリプト実行中の列の値に依存するような場合は、これが問題となる可能性があります。評価を制御する手段が必要な場合は、EvalFormula コマンドまたは Run Formulas コマンドを使用します。

1 つの列の評価を実行するには、その列に EvalFormula コマンドを送ります。計算式の節の後で、列を作成するコマンド内でも実行できます。その例を示します。

```
Current Data Table() << New Column("比率", Numeric,Formula(:Name("身長(インチ)")/
   :Name("体重(ポンド)")), EvalFormula)
```

dt << Run Formulas は、他の計算式を評価した後で評価するために保留されている式も含め、データテーブルのすべての式の評価を実行します。この関数は、すべての列の計算式を実行する場合に便利です。

ヒント：この方法は、EvalFormula より適しています。EvalFormula も計算式を評価しますが、バックグラウンドで再評価されないようにはしません。バックグラウンドで行われるタスクは、計算式が正しい順序で評価されるよう、十分な注意を払う必要があります。

Run Formulas コマンドをデータ列に送った場合は、コマンドを送った時点で評価が実行されますが、そのためにスケジュール設定されてペンディング中だった計算式が本来の時間に評価されなくなるわけではありません。そのため、コマンドをデータテーブルとデータ列にも送ると、計算式が 2 回評価される場合があります。2 回評価されることは、乱数関数が含まれている計算式では望ましい場合もありますが、設定されている乱数シード値に依存する場合は望ましくないこともあります。同一の値のセットを生成させるために乱数関数と Random Reset(seed) の機能を使用している場合には、Run Formulas コマンドを使うと 2 回目の評価を行わずに済みます。

注：すべてのプラットフォームで Run Formulas コマンドがデータテーブルに送られ、すべての計算式の評価が完了してから、分析が開始されます。

計算後の値を設定する

col << Set Each Value(*expression*) は、データテーブルの行ごとに式を計算し、結果を列に割り当てます。この式は、計算式としては格納されません。

メタデータの削除

テーブルに含まれるテーブル変数、テーブルプロパティ（スクリプトなど）、計算式は、次のコマンドを使って削除することができます。

```
dt << Delete Table Variable(name);
dt << Delete Table Property(name); //
col << Delete Formula;
col << Delete Property(name);
col << Delete Column Property(name);
```

計算

この節では、列または行ごとに統計量を事前計算する関数について説明し、計算式エディタの裏で JSL の式がどのように動作しているかを示します。

事前計算される統計量

JMP には特殊な「事前計算」関数、Col Maximum、Col Mean、Col Minimum、Col N Missing、Col Number、Col Quantile、CV（Coefficient of Variation：変動係数）、Col Standardize、Col Std Dev、Col Sum、Maximum、Mean、Minimum、NMissing、Number、Std Dev、Sum があります。

注：統計量は Summarize277 ページでも計算されます。Summarize の名前付き引数でこれらの事前計算統計関数と同じ名前のものがありますが、それらの引数が事前計算統計関数を呼び出すということではありません。たまたま名前が一致しているだけです。

統計量はすべて**事前計算**されます。つまり、JMP は指定された行または列で統計量を一度計算しており、その結果を定数として使っているのです。一度計算された統計量が何度も繰り返して使われるので、同等の計算式で算出された結果を使うよりも計算効率が良くなります。

JMP は、スクリプト内に事前計算関数を見つけると、その関数をすぐに計算し、以後はその結果を定数として使います。そのため、事前計算関数を使うと、列ごとの計算結果を行ごとの計算に使用できるようになります。たとえば、列の計算式内で Col Mean を使う場合は、まず指定された列の平均を計算し、次に、各行に関する残りの計算式の計算でその結果を定数として使います。たとえば、事前計算した平均と標準偏差を使って列を標準化する計算式などが考えられます。

```
(:Name("身長（インチ）")-Col Mean(:Name("身長（インチ）")))/Col Std Dev(:Name("身長（
  インチ）"))
```

「Big Class.jmp」データセットの場合、Col Mean(:Name("身長(インチ)")) は 62.55、Col Std Dev(:Name("身長(インチ)")) は 4.24 です。したがって、上の計算式では、各行の「身長(インチ)」の値から 62.55 を引いた後で 4.24 で割ります。

注：事前計算関数は、行の除外の属性を無視するので、除外された行が計算に含まれます。除外された行を無視して要約統計量を得るには、「一変量の分布」プラットフォームを使ってください。

列ごとの関数

名前が「Col」で始まる関数はすべて、**列ごと**、つまり指定された列の値を上から下に評価し、単一の数値を戻します。たとえば、Col Mean(:Name("身長(インチ)")) は、列「身長(インチ)」のすべての行の値の平均を算出し、それをスカラー値の結果として戻します。次に例を挙げます。

```
Average Student Height = Col Mean(:Name("身長(インチ)"));
Height Sigma = Col Std Dev(:Name("身長(インチ)"));
```

行ごとの関数

次に示す「Col」が**付いていない**関数は、指定された変数の値について**行ごと**に動作し、結果を戻します。たとえば、Mean(:Name("身長(インチ)"), :Name("体重(ポンド)")) は、データテーブルの現在の行の**「身長(インチ)」**と**「体重(ポンド)」**の平均を算出します。行ごとの統計量は、適切なデータテーブルの行コンテキスト内で使われた場合にだけ有効です。いくつかの例を挙げます。

```
// JSL のグローバル変数に割り当てられた行 7 のスカラー値の結果
row()=7; ::scalar = Mean(:Name("身長(インチ)"), :Name("体重(ポンド)"));

// データテーブル内に作成される計算式の列
New Column("計算列",
   formula(mean(:Name("身長(インチ)"), :Name("体重(ポンド)"))/:年齢));

// 結果のベクトル
vector=J(1,40); // 結果を保持するために 1x40 の行列を作成する
For Each Row(vector[row()]=mean(:Name("身長(インチ)"),:Name("体重(ポンド)"))); //
   ベクトルを埋める
```

行ごとの関数は、次のように、ベクトル（列ベクトル）またはリストの引数をとることもできます。

```
myMu=mean([1 2 3 4]); mySigma=stddev({1, 2, 3});
```

計算式エディタの計算式

自動的に評価され、列のセルの値を生成する計算式を列に保存できます。計算式を開くと、計算式を構造的に編集するための計算式エディタインターフェースが表示されます。計算式は JSL により実装されているので、計算式エディタ内の式をダブルクリックすれば、JSL テキスト形式の式を取得できます。テキストは編集でき、フォーカスされていないときは、構造形式に戻ります。

注：計算式エディタウィンドウで作成した計算式の列と、`New Column(..., Formula(...))`や`Col << Formula(...)`などのコマンドを使ってJSLから直接作成した計算式の間に違いはありません。

第 10 章

プラットフォームのスクリプト
分析の作成、反復、変更

プラットフォームをスクリプトから起動し、その後もスクリプトから制御することができます。インタラクティブな方法で分析した場合は、分析を再度実行するためのスクリプトを保存できます。

この章では、プラットフォームを制御するスクリプトの記述方法と、プラットフォーム特有のコマンドに関するヒントを紹介します。その他の情報については、［スクリプトの索引］を参照してください（［**ヘルプ**］>［**スクリプトの索引**］を選ぶ）。

目次

概要 349
分析プラットフォームのスクリプト 350
インタラクティブなプラットフォームの起動とそのスクリプトの取得 351
プラットフォームの起動 351
スクリプトの保存 351
変更を加える 352
プラットフォームスクリプトの構文 353
By グループレポート 353
By グループスクリプトの保存 356
現在行われている分析にスクリプトコマンドを送る 357
コマンドと引数の規則 357
複数のメッセージを送る 358
オブジェクトが対応するメッセージ 359
Show Properties が戻すリストの読み方 359
プラットフォームの起動 361
列の指定 361
プラットフォームの Action コマンド 361
非表示のレポート 362
タイトル 362
プラットフォームウィンドウの一般メッセージ 363
追記 367
スプライン曲線 367
モデルのあてはめの効果 367
モデルのあてはめに送るコマンド 369
DOE（実験計画）のスクリプト 369
三次元散布図のスクリプト 371
管理図 371

概要

プラットフォームをスクリプトで操作する方法を学ぶのは、インタラクティブな操作を学ぶのと同じくらい簡単です。しかし、実際にスクリプトを自分で書くためには多少の努力が必要です。スクリプトを入力する手間がかかり、正しい構文を習得する必要があるからです。幸いなことに、必要なスクリプトの大部分をプラットフォーム自身に作成させることができ、分析を繰り返し行うため自動化する必要がある場合も、それにいくらか追加の作業をすれば済みます。また、スクリプトを作成することで、分析をカスタマイズし、組み合わせることもできます。

現在、プラットフォームのスクリプト保存機能にはいくつかの制限があります。JMP では、分析コマンドと表示のカスタマイズが保存されますが、一部の表示のカスタマイズは保存されません。そのため、表示のカスタマイズが重要な場合は、表示インターフェースのプログラミングを学習する必要があります。また、分析の状態は JMP に記録されますが、その状態を導く一連のイベントは記録されません。したがって、実行時の JMP プラットフォームの動作を再生するスクリプトが必要な場合は、そのための学習も必要です。

プラットフォームの結果は 2 層に分かれて存在します。1 つはプラットフォーム自身で、分析結果を含み、分析コマンドに応答します。もう 1 つはプレゼンテーションの表示で、様々な表示用のコマンドに応答します。3 つめのオブジェクトはデータテーブル自身で、現在行われている分析に関連して使用されます。

1. 直線のあてはめなど分析にオプションを追加する場合は、この章で説明する手法を使って、コマンドをプラットフォーム自身に送ります。
2. グラフの枠を大きくする場合は、コマンドをその表示に送ります。「表示ツリー」の章の「表示の操作」（379 ページ）を参照してください。
3. データテーブル内の行に対応する特定の点を強調表示する場合は、行の属性コマンドをデータテーブルに送ります。「データテーブル」の章の「行の属性と演算子」（321 ページ）を参照してください。

分析プラットフォームやレポートを独自にカスタマイズしたい場合は、「表示ツリー」の章の「表示ツリーの作成」（399 ページ）を参照してください。JSL の簡潔な行列表記を使って独自の統計分析を行いたい場合は、「データ構造」の章の「行列」（154 ページ）を参照してください。

分析プラットフォームのスクリプト

分析プラットフォームは、ウィンドウとメニューに表示されるもの（レポート内の赤い三角ボタンのメニューやコンテキストメニューに表示されているコマンドなど）と同じキーワードを使って実行します。

JMPをインタラクティブに使用した後で、作成した分析に対応するスクリプトを取得できます。この機能は、以下のような場合に利用できます。

1. 生成されたスクリプトを参考にしてJSLを学習します。プラットフォームを起動し、インタラクティブな方法で作業してから、作業に対応するスクリプトを保存してテストします。インターフェースを介して変更した後で、スクリプトの変更を調べます。
2. 分析から生成された多数のスクリプトをスクリプトウィンドウに保存した後、ファイルに保存します。そのスクリプトを使えば、後で分析を再現したり、新しいデータで分析を再作成したりできます。
3. プラットフォームのスクリプトを保存します。そして、JSL のプログラミング機能を使って分析やレポートをカスタマイズするために、そのスクリプトを編集します。

各プラットフォームで使用できるスクリプトコマンドのリストは、［**ヘルプ**］>［**スクリプトの索引**］を選択し、［**オブジェクト**］を選択すると表示されます。ここには、すべてのスクリプト可能なプラットフォームオプションの説明、構文、および例が含まれています。

この章の表記法

この章では、示されているとおりの表記で使用する必要があるコマンドはコマンド名の頭文字を大文字で示し、実際に選択したものを入力する引数は小文字で示します。次の例で説明すると、`Connect Color`は、そのとおりに入力する必要があるコマンドで、`color`の部分は、ユーザが好みで入力する色を指します。

```
Connect Color(color)
```

この場合、括弧内の引数は色の値でなければなりません。色の値とは、JMPの色番号、`red`や`blue`などのサポートされている色名、または`{.75, .50, .50}`のようにリストで指定するRGB値です。このように、複数の指定方法がある場合は、次のように、「または」を意味する「|」文字で示します。

```
Connect Color( number | "color name" | {r,g,b} );
```

大文字と小文字は区別されません。また、実際のスクリプトには「|」を入力する必要はありません。

コマンド、オプション、メッセージという用語は、ほとんど同じものを指していますが、操作のどの側面を強調するかによって使い分けています。「引数」とは、ある項目の後ろに括弧で囲んで指定するものを指します。引数の引数を指定する（引数の中に更に引数をとる）ことができるものもあります。

インタラクティブなプラットフォームの起動とそのスクリプトの取得

ここでは、JMP のインターフェースとスクリプト言語がどのように関連するかについて、例を示して説明します。

プラットフォームの起動

1. サンプルのデータテーブル「Big Class」を開きます。
2. ［**分析**］>［**二変量の関係**］を選択します。
3. 「**身長 (インチ)**」を選択し、［**Y, 目的変数**］をクリックします。
4. 「**年齢**」を選択し、［**X, 説明変数**］をクリックします。
5. ［**OK**］をクリックします。

スクリプトの保存

すべての分析プラットフォームに以下の図のような［**スクリプト**］サブメニューがあり、現在の状態を再現する JSL スクリプトを生成するためのコマンドが表示されます。これらのコマンドは、スクリプトの出力先を選べるようになっています。

分析のやり直し プラットフォームを新たに起動して、すべての分析を指定どおりに再実行します。これを利用すれば、データテーブルの状態が変化したときに分析結果を更新することができます。たとえば、データテーブルに対してデータの修正、サブセットの選択、データの追加などを行った場合です。

分析の再起動 起動ウィンドウを開きます。レポートの生成に必要な指定がすでに行われています。

自動再計算 自動再計算のオン／オフを切り替えます。

スクリプトのコピー レポートを再作成するスクリプトをクリップボードにコピーします。コピーしたスクリプトは、スクリプトウィンドウや他のアプリケーションに貼り付けることができます。

スクリプトをデータテーブルに保存 スクリプトを新しいプロパティとしてデータテーブルに保存します。データテーブルウィンドウに、プラットフォームの名前がついたスクリプトが表示されます。スクリプトには［**スクリプトの実行**］ができるポップアップメニューがあります。保存されたスクリプトなどのプロパティは、後で使えるように、データテーブルと一緒に保存されます。サンプルデータの多くにスクリプト例が含まれています。

スクリプトをジャーナルに保存 現在のジャーナル（ジャーナルが開いていない場合は新しいジャーナル）に、レポートを再現するスクリプトの実行ボタンを保存します。

スクリプトをスクリプトウィンドウに保存 プラットフォームの現在の状態のスクリプトをスクリプトウィンドウに保存します。このウィンドウでそのスクリプトを表示、編集、および実行できます。

スクリプトをレポートに保存 スクリプトをレポートの最上部のテキストボックスに保存し、プラットフォームウィンドウや、そこから作成したジャーナルで、分析を再現するスクリプトを確認できます。この機能は、分析結果とともに、分析とその引数の設定を一覧表示するのに便利です。

すべてのオブジェクトのスクリプトを保存 ウィンドウ内におけるすべてのオブジェクトのスクリプトをスクリプトウィンドウに保存します。スクリプトウィンドウでは、そのスクリプトを表示、編集、および実行できます。たとえば、「二変量の関係」を選択した場合、連続尺度、名義尺度、順序尺度の列をさまざまに組み合わせることで、1つのレポートの中で「二変量」、「一元配置」、「分割表」、および「ロジスティック」プラットフォームが実行されます。[スクリプトをスクリプトウィンドウに保存]は、ユーザがこのオプションを選んだところのオブジェクトだけのスクリプトを保存します。[すべてのオブジェクトのスクリプトを保存]は、どのオブジェクトのメニューオプションを使ったかに関係なく、ウィンドウ内のすべてのオブジェクトのスクリプトを保存します。また、By グループがある場合、[スクリプトを〜に保存]は、1 グループだけのスクリプトを生成するのに対し、[すべてのオブジェクトのスクリプトを保存]は、ウィンドウ内のすべての By グループのスクリプトを保存します。このため、レポートを再実行すると、すべての By グループが同じウィンドウに表示されます。これは、[すべてのオブジェクトのスクリプトを保存]を実行すると、New Window() コマンド内にすべての By 変数に基づく分析が含まれるからです。たとえば、男性と女性でグループ化された身長と体重の二変量のあてはめを実行するスクリプトは、次のようになります。

```
New Window("Big Class.jmp: 二変量の関係 ",
Bivariate(Y(:Name(" 身長 ( インチ )")), X( :Name(" 体重 ( ポンド )")), Fit Line,
Where( :性別 == "F"));
Bivariate(Y(:Name(" 身長 ( インチ )")), X( :Name(" 体重 ( ポンド )")), Fit Line,
Where( :性別 == "M")));
```

スクリプトをプロジェクトに保存 レポートを再現するスクリプトを作成し、それを JSL ファイルとして JMP プロジェクトに保存します。

データテーブルウィンドウ By グループがある場合は、分析に関連付けられた By グループのデータだけを含むデータテーブルウィンドウを表示し、そうでない場合は、元のデータテーブルウィンドウを表示します。

変更を加える

「一元配置分析」レポートウィンドウでの作業を続けましょう。

1. 「年齢による身長 (インチ) の一元配置分析」の赤い三角ボタンのメニューで[**平均 /ANOVA**]を選択します。
2. 同じメニューで［**平均の比較**］>［**各ペア ,Student の t 検定**］を選択します。
3. 同じメニューで［**スクリプト**］>［**スクリプトをスクリプトウィンドウに保存**］を選択します。

その結果、スクリプトは次のようになります。

```
Oneway(
   Y( :height ),
   X( :age ),
   Each Pair( 1 ),
   Name( "Means/Anova" )(1),
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
```

スクリプトには、メニューで選択したオプションが記録されます。それらはブール値をとる（オンまたはオフ）オプションであり、それをオンにするという意味の1という引数を持っています。選択したオプションの他に、それらのオプションを選ぶと自動的に追加される3つの表示オプションが含まれています。メニューで選択する選択肢とそれに対応するJSLコマンドは、まったく同じ効果を持ちます。

このスクリプトを実行すると、プラットフォームのウィンドウとメニューの手順をすべて省略し、同じレポートをすぐに得ることができます。スクリプトウィンドウからスクリプトを実行するには、テキストを選択してから、[**編集**] > [**スクリプトの実行**] を選ぶか、Ctrlキー（Windows）またはcommandキー（Macintosh）を押しながらRキーを押します。テキストを選択しないと、ウィンドウに含まれているスクリプト全体が実行されます。

プラットフォームスクリプトの構文

では、このスクリプトを詳しく見ていきましょう。

```
Oneway(
   Y( :height ),
   X( :age ),
   Each Pair( 1 ),
   Name( "Means/Anova" )(1),
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
```

プラットフォームのスクリプトはすべて、プラットフォームを呼び出すコマンドから始まります。この場合はOnewayです。Onewayコマンド内には、YおよびXの列の役割リストのように起動時に必要な引数と、Each Pair(1)のように起動後にプラットフォームに送られるオプションの、2種類の引数があります。

オプションのほとんどは、チェックマークをつけるかつけないかで示す単純なオン／オフ選択です。チェックマークの有無に対応するスクリプトは、ブール引数の1または0をとります。

それ以外のコマンドでは、値を指定したり、選択したりするためのダイアログボックスが表示されます。スクリプト内では、このような指定は、通常はウィンドウに表示される順序（上から下、左から右）に従ってカンマで区切られ、括弧に囲まれて表示されます。

Byグループレポート

ほとんどのプラットフォームで、1つまたは複数のBy列で定義した行のサブグループに対して処理を繰り返し実行できます。これをスクリプトで実行するには、起動コマンドの中にBy引数を入れ、列名をByの引数としてリストします。

例

1. サンプルのデータテーブル「Big Class」を開きます。
2. 次のスクリプトを実行して、性別ごとの二変量のレポートを作成します。

```
biv = Bivariate(Y(:Name("体重（ポンド）")), X(:Name("身長（インチ）")), By(:性別));
```

この起動メッセージは、データテーブル内で性別が「F」のグループと「M」のグループの2つのアウトラインノードが表示されるレポートウィンドウを生成します。プラットフォームである二変量 [] への参照を戻すのではなく、プラットフォームオブジェクトは参照のリスト { 二変量 [], 二変量 []} を戻します。

```
show(biv);
   biv = { 二変量 [], 二変量 []}
```

参照が含まれたリストにメッセージを送ると、すべてのノードを同時に変更できます。また、個々の参照に個別にメッセージを送ると、ノードを個別に変更できます。

3. 次の行を実行して、両ノードに回帰のあてはめを追加します。

   ```
   biv << fit line;
   ```

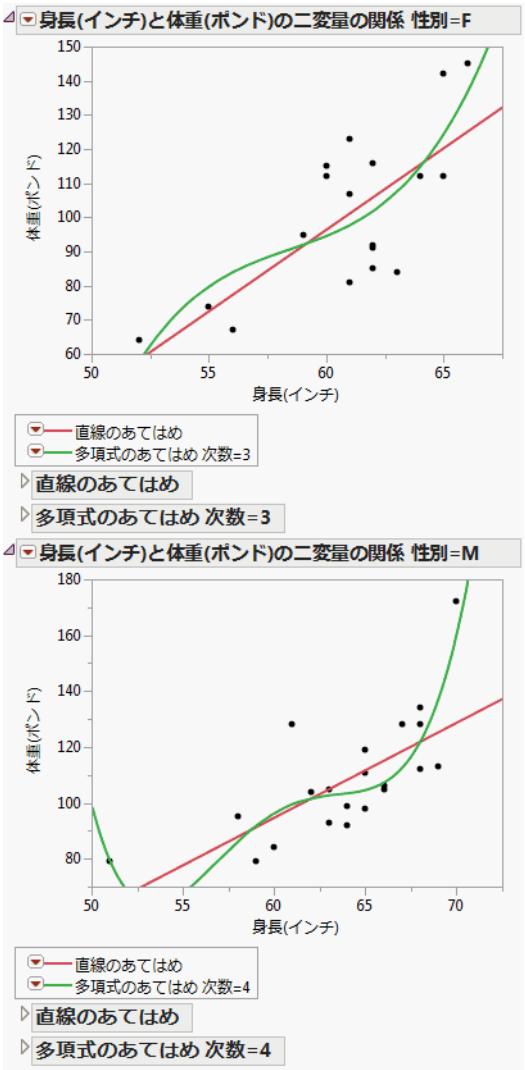
4. 次の行を実行して、F ノードに 3 次のあてはめを追加します。

   ```
   Bivariate[1] << fit polynomial(3);
   ```

5. 次の行を実行して、M ノードに 4 次のあてはめを追加します。

   ```
   Bivariate[2] << fit polynomial(4);
   ```

図10.1 By グループレポートへのアクセス



By() にリストされた複数の列により、各サブグループのノードが生成されます。たとえば、By(性別 , 年齢) により、「性別F 年齢12」、「性別F 年齢13」、...、「性別F 年齢17」、「性別M 年齢12」、「性別M 年齢13」...、「性別M 年齢17」のノードが生成されます。

以下に、By グループを指定してプラットフォームを起動し、各グループの結果を取り出す方法を示します。

```
// データテーブル「Big Class」を開く
dt=Open("$SAMPLE_DATA¥Big Class.jmp");

// 「一元配置」プラットフォームを起動する
onew = Oneway(x(: 年齢 ),y(:Name(" 身長 ( インチ )")),by(: 性別 ),anova);
                    // onew はプラットフォームオブジェクトの参照リスト
r = onew<<report; // r はディスプレイボックスのリスト
nBy = nItems(r);  // By グループの数
vc = j(nBy,1,0);  // 誤差平方和を保存する行列

// 結果の抽出
for(i=1, i<=nBy, i++,
   vc[i] = r[i][
      OutlineBox(" 分散分析 "),
      ColumnBox(" 平方和 ")][2]);
show(vc);

summarize(byValues=by(: 性別 ));
newTable(" 分散 ")
   << newColumn(" 性別 ",character,width(8),values(byValues))
   << newColumn(" 誤差平方和 ",numeric,continuous,values(vc));
```

By グループスクリプトの保存

場合によっては、分析プラットフォームの［**スクリプト**］サブメニューに加えて、［**By グループでまとめてスクリプト化**］が表示されます。このサブメニューを使うと、By グループを使って作成したレポートの再現スクリプトを保存することができます。以下のオプションがあります。

- **分析のやり直し**
- **分析の再起動**
- **スクリプトのコピー**
- **スクリプトをデータテーブルに保存**
- **スクリプトをジャーナルに保存**
- **スクリプトをスクリプトウィンドウに保存**

これらのオプションは、レポート内の By グループをすべて再現する点を除き、［スクリプト］サブメニューのコマンドと同様に機能します。

注: スクリプトをデータテーブル、ジャーナル、またはスクリプトウィンドウに保存するには、Save ByGroup 関数を使用します。例については、［スクリプトの索引］を参照してください。

現在行われている分析にスクリプトコマンドを送る

プラットフォームが起動した後、プラットフォームを制御するには、Send 関数またはこれと等価の << 演算子を使って、現在アクティブなプラットフォームにメッセージを送ります。

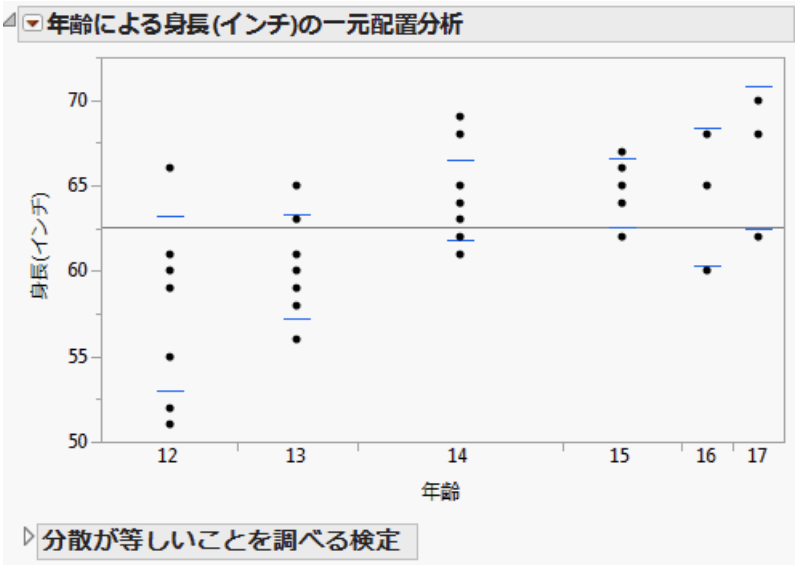
まず、オブジェクトを指定する方法が必要です。最も簡単な方法は、プラットフォームを起動するスクリプトでオブジェクトへの参照をグローバル変数に割り当てる方法です。たとえば、次のスクリプトは、一元配置分析への参照を変数 *oneObj* に保存します。

```
oneObj = Oneway(Y(:Name("身長(インチ)")), X(:年齢));
```

次に、Send 関数または等価の << 演算子を使ってメッセージを送ります。

```
Send(oneObj, Unequal Variances(1));
oneObj << Unequal Variances(1);
```

図10.2 レポートの要素をスクリプトで追加



コマンドと引数の規則

1. 状態を**切り換える**ためのブール値をとるオプションの引数は省略できます。切り換えるとは、メッセージを送ると、状態がオフの場合はオンになり、オンの場合はオフになる機能のことです。前記の例では、以下の2つのコマンドは同じ結果になります。最初のコマンドを繰り返し実行しても結果は変わりませんが、2番目のコマンドを繰り返し実行すると、そのたびに機能のオン／オフが切り換わります。

   ```
   oneObj<<Unequal Variances(1);
   oneObj<<Unequal Variances;
   ```

2. 前記の［**平均/ANOVA/t検定**］のように、メニューにカンマやスラッシュで区切られた複数のオプションがある場合は、そのコマンドのうちのどれでも使用できます。ここでは別名が「Means」（平均）となるコマンドが2つありますが、このように複数のコマンドの別名が同じになる場合、スクリプト言語では、GUIのポップアップメニューで先（上）に出てくるオプションが優先します。
3. グラフのサイズ変更など、表示に変更を加えた場合も、それがスクリプトに保存されます。
4. サブメニューの項目が値やパラメータの設定ではなくコマンドの場合、対応するスクリプトには親項目はなく、その項目自身が記録されます。たとえば前記の例で、「一元配置」のメニュー項目［**ノンパラメトリック**］には、［Wilcoxon検定］などの3つのコマンドを提供するサブメニューがあります。スクリプトでは、以下の例のように、サブメニューの項目名だけを使います。

   ```
   oneObj = Oneway(Y(:Name("身長(インチ)")), X(:年齢), Wilcoxon Test(1));
   oneObj << Wilcoxon Test(1);
   ```

5. サブメニューの項目が、独立したコマンドではなく設定の値である場合は、スクリプトに親項目を指定し、その引数としてサブメニューでの選択を指定します。たとえば、「一元配置」では［α 水準の設定］にサブメニューがあり、その選択肢は［0.10］、［0.05］、［0.01］、［その他...］です。スクリプトで値を変更するには、選択した値をSet Alpha Levelの引数として指定します。

   ```
   oneObj << SetAlphaLevel(0.01);
   ```

6. プラットフォームが戻すスクリプトと、自分で作成するスクリプトが異なっているように見えることがよくあります。たとえば、「一変量の分布」は、次の短いスクリプトで起動できます。

   ```
   dist=Distribution(Y(:Name("身長( インチ)"),:Name(" 体重( ポンド)")));
   ```

しかし、「一変量の分布」にそのスクリプトを保存するように要求すると、次のようなスクリプトが表示されます。

```
Distribution(Continuous Distribution(Column( :height), Axis
   Settings(Scale(Linear), Format("Fixed Dec", 0), Min(50), Max(72.5), Inc(5),
   Minor Ticks(1))), Continuous Distribution(Column( :Name("体重(ポンド)")),
   Axis Settings(Scale(Linear), Format("Fixed Dec", 0), Min(70), Max(180),
   Inc(10))));
```

複数のメッセージを送る

複数のメッセージを送るには、<<演算子を追加するか、Sendの引数を追加します。

```
dist<<quantiles(1)<<moments(1)<<more moments(1)<<horizontal layout(1);
Send(dist, quantiles(1), moments(1), more moments(1), horizontal layout(1));
```

<<は**省略（eliding）演算子**なので、複数のオペランドを連結し、オペランドがグループ化されている場合とそうでない場合とでは異なる動作をします。<<記号を使って複数のメッセージを1つにまとめ、左から右へ順に実行できます。括弧を使ってグループ化し、メッセージを送った結果にさらにメッセージを送ることができます。

```
(dist<<stem and leaf(1)) << horizontal layout(0);
```

上の場合は、括弧を使ってグループ化しても問題はありません。どちらにしても、メッセージは左から右に実行されるからです。しかし、メッセージを子オブジェクトに送るときは、これが**問題になります**。このことについては次に説明します。

メッセージのリストにより、メッセージを1つにまとめて送ることもできます。

```
dist<<{quantiles(1), moments(1), more moments(1), horizontal layout(1)};
```

オブジェクトが対応するメッセージ

現在あるオブジェクトに対して送れるメッセージにはどのようなものがあるでしょうか。使用できるオプションを調べるには、次のようないくつかの方法があります。

1. まずインタラクティブインターフェースによる手順を試し、その後で、保存されたスクリプトを調べます。
2. プラットフォームウィンドウのインターフェースを調べます。ポップアップメニューとコンテキストメニューのすべての項目と同じことを行うために、同じ名前のJSLメッセージと引数が用意されています（「インタラクティブなプラットフォームの起動とそのスクリプトの取得」（351ページ）を参照）。
3. ［**ヘルプ**］メニューから、［**スクリプトの索引**］を選択します。目的のオブジェクトタイプを見つけて、リスト内の項目をクリックします。
4. `Show Properties(objectRef)`を使って、オブジェクトが受け取ることのできるすべてのメッセージを「ログ」ウィンドウに一覧表示します。

```
show properties(oneObj);
   Quantiles [ブール](Shows or hides a quantile report.)
   Means/Anova [ブール](Shows or hides both an ANOVA and a means report.)
   Means/Anova/Pooled t [ブール](Shows or hides a t test, an ANOVA, and a means
      report.)
   Means and Std Dev [ブール](Shows or hides a report with both the mean and standard
      deviation for each level of the X variable.)
   ...
```

`Show Properties`は、データテーブルとディスプレイボックスでも使用できます。「データテーブル」の章の「データテーブルオブジェクトに送ることができるメッセージ」（260ページ）および「表示ツリー」の章の「ディスプレイボックスでできることを調べる」（388ページ）を参照してください。

Show Propertiesが戻すリストの読み方

`Show Properties`が出力するほとんどの項目の行の末尾に、括弧 [] で囲まれたヒントがあります。この節では、一般的な例として、Bivariate（二変量）に対して`Show Properties`を使った場合を取り上げます。

```
biv= Bivariate(Y(:Name(" 身長 ( インチ )")), X(: 年齢 ));
show properties(biv);
```

［サブテーブル］は、サブメニューに表示されるコマンドセットを指します。サブテーブル内のコマンドはインデントされて表示され、使用する場合は、親項目ではなくサブ項目を直接使います。

```
Script  [ サブテーブル ]
 Redo Analysis [アクション] (Rerun this same analysis in a new window.The analysis
  will be different if the data has changed.)
 Save Script to Datatable  [ アクション ] (Return to the launcher for this
  analysis.)
  ...
```

［ブール］は、オプションのオン／オフを切り換え、通常、引数は 1 または 0 です。引数を指定しないでメッセージを送ると、そのたびに逆の状態に切り換えられます。［ブール］メッセージが、［デフォルトはオン］であることもよくあります。

```
Show Points [ ブール ] [ デフォルトはオン ]
```

[新規エンティティ] は、ユーザインターフェースにウィンドウを表示するコマンドです。次は、一変量の分布のオブジェクトプロパティの一例です。

```
Prediction Interval [ 新規エンティティ ] (Prediction Interval to contain a single
   future observation or the mean of m future observations.)
```

[アクション] は、ユーザインターフェースでユーザが行う選択です。アクションの引数に決められた標準値はないので、まず、インターフェースの項目を試し、その後で、プラットフォームが保存したスクリプトを調べてください。

```
Fit Mean [ アクション ] (Fits a flat line at the mean.)
Fit Line [ アクション ] (Fits a regression line to the data.)
```

［アクションの選択］と［選択肢］には、引数に指定する特定の選択肢のセットがあります。

```
Fit Polynomial [ アクションの選択 ] {2,quadratic, 3,cubic, 4,quartic, 5, 6}
```

ここで、バブルプロットのプロパティにおける [選択肢] の例を挙げます。

```
Draw [ 選択肢 ] {Filled, Outlined, Filled and Outlined}
```

注意事項！プラットフォームへの参照と**レポート**への参照を混同しないようにしてください。これらのオブジェクトはタイプが異なり、受け取ることのできる JSL メッセージのタイプも異なります。たとえば、プラットフォームは、検定、プロットの描画、ウィンドウ全体を閉じるなどを実行できます。レポートは、イメージとしてコピー、ディスプレイボックスの選択、アウトラインノードを閉じるなどを実行できます。

この章では、プラットフォームのスクリプトを作成する方法を説明します。レポートのスクリプトを作成する方法については、「表示ツリー」の章の「表示の操作」（379 ページ）を参照してください。

プラットフォームの起動

列の指定

通常、プラットフォームを起動するスクリプトでは、分析する列を指定する必要があります。列および役割を指定していないプラットフォーム起動スクリプトを実行すると、プラットフォームを起動するためのダイアログボックスが表示されます。列を選んでから［**OK**］をクリックすると、スクリプトで指定した分析が実行されます。つまり、JMP は、必要な列の割り当てをダイアログから取得した後、スクリプト内の他のメッセージに戻ってその指示に従います。

列名の引数に対して式を指定して、その式を評価したい場合には、`Eval` または `EvalList` 関数の中にその式を指定します。

```
Distribution(Y(Eval("X"||char(i))));
```

プラットフォーム起動スクリプトの列名の引数は、{ } で囲んだリストで指定することもできます。したがって、以下のスクリプトはすべて有効です。

```
Distribution(Y(:Name("身長(インチ)"),:Name("体重(ポンド)")));
Distribution(Y({:Name("身長(インチ)"),:Name("体重(ポンド)")})); // 上と同じ
Distribution(Weight({}));
                                // 空のリスト（プラットフォームの起動ウィンドウが表示される）
```

このマニュアル全体をとおして、*col* という表記はデータテーブルの列の参照を表し、*nomCol*、*ordCol*、*contCol* の表記はそれぞれ名義尺度の列、順序尺度の列、連続尺度の列を表していると考えてください。多くの場合、他の尺度の列も受け入れられます。許可されていない役割に列を割り当てようとすると、JMP はエラーを戻します。

プラットフォームの Action コマンド

`Action` コマンドをプラットフォームに送ると、任意の式が評価されます。このコマンドを使うと、プラットフォームの起動ウィンドウを表示し、ユーザが列を選択してプラットフォームが起動されるまで待機した後、続けて別のプラットフォームの起動ウィンドウを呼び出す、という処理が実行できます。

次の例では、スクリプトはまず 4 つのプラットフォームのそれぞれに列を割り当てるようユーザを促し、次にログに `Done` と表示します。スクリプトは、プラットフォームの起動に必要な列の選択が行われるのを待機するため、4 回停止します。そのたびに、ユーザは、起動ウィンドウに値を入力します。スクリプトが完了したときには、4 つのレポートが表示されています。

このスクリプトは、最初のプラットフォーム起動ウィンドウが表示されたときに、最後まで実行されます。他の処理は、保存された式に基づいて実行されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Distribution( Action( doit ) );
doit = Expr( New Window( "Bivariate",
   Bivariate( Action( doit2 ) ) )
```

```
   );
doit2 = Expr( New Window( "Oneway",
   Oneway( Action( doit3 ) ) )
   );
doit3 = Expr( New Window( "Contingency",
   Contingency( Action( doit4 ) ) )
   );
doit4 = Expr(
   Print("Done");
);
```

非表示のレポート

プラットフォームの起動では、ウィンドウを表示しない**invisible**オプションを使うことができます。Fit Modelのスクリプトでこのオプションを使用すると、「モデルのあてはめ」起動ウィンドウも結果のウィンドウも表示されなくなります。

このオプションを使う場合、スクリプトでウィンドウの動きを把握して閉じるようにしてください。これは、非表示ウィンドウによって使用されるリソースは手動で解放する必要があるためです。

次の例では、「二変量の関係」のレポートから*F*値を抽出しています。このスクリプトを実行する前に、「Big Class.jmp」を開いておいてください。

```
biv =  bivariate(X(:Name(" 身長 ( インチ )")),Y(:Name(" 体重 ( ポンド )")),invisible);
biv<<fit line;
r = biv<<report;
fratio = r[OutLineBox(" 分散分析 "),ColumnBox("F 値 ")][1];
r<<close window;
```

invisibleオプションは、[**テーブル**] メニューの操作でも使用でき、ウィンドウが（非表示になるのではなく）作成されないようにします。

```
Current Data Table()<<Select Where(: 年齢 ==14);
subDt = Current Data Table() << Subset(invisible);
subDt<<bivariate(X(:Name(" 身長 ( インチ )")),Y(:Name(" 体重 ( ポンド )")),fit line);
```

Bivariateコマンドで**Where**節を使うと、上のスクリプトはもっと簡単に実行できます。

```
bivariate(x(:Name(" 身長 ( インチ )")),y(:Name(" 体重 ( ポンド )")), Where(: 年齢 ==14),fit
   line);
```

タイトル

起動要求に**title**コマンドを追加することで、タイトル（プラットフォームのレポートのタイトルバーに表示されるもの）が指定できます。たとえば次のようなスクリプトを作成すると、二変量の関係で表示される標準のレポートタイトルがユーザ定義のものに変わります。

```
Bivariate(X(:Name(" 身長 ( インチ )")),Y(:Name(" 体重 ( ポンド )")), Title(" タイトル "));
```

プラットフォームウィンドウの一般メッセージ

Save Scriptコマンドは、ほとんどすべてのプラットフォームで使用できます。その他のコマンドは、表10.2「プラットフォームウィンドウに送れるメッセージ」（364ページ）に記載しています。特に、Reportコマンドを使うと、ディスプレイボックスごとにアクセスして、ウィンドウのズーム、スクロール、その他の外観をきめ細かく制御できます。たとえば、レポートのアウトラインノードを閉じるには、まず、ディスプレイボックスツリーへの参照を取得し、該当するアウトラインボックスを添え字で指定し、それにCloseメッセージを送ります。

```
r = platformRef << Report;
r[" あてはめの要約 "] << close;
```

これについては、「表示ツリー」の章の「表示の操作」（379ページ）で詳しく説明しています。

表10.1 分析プラットフォームのスクリプト

目的	コマンド	説明
使用できるメッセージを調べる	Show Properties(*obj*)	与えられたオブジェクトが解釈できるメッセージと、その基本的な構文情報を表示する。プラットフォームだけでなく、すべてのスクリプト可能なオブジェクトで機能します。
メッセージを送る	*obj* << *message* Send(*obj*, *message*)	プラットフォームオブジェクト（*obj*）にメッセージ（*message*）を送る。
複数のメッセージを送る	*obj* << *message* << *message*... *obj* << {*message*, *message*, ...}	プラットフォームオブジェクト（*obj*）に（左から右の順に）一連のメッセージ（*messages*）を送る。
メッセージを子に送る	*obj* << (*child* << *message*)	メッセージ（*message*）を、プラットフォームオブジェクト（*obj*）内の子オブジェクト（*child*）に送る。
レポートの非表示	Platform name(...,invisible)	レポートからの出力が画面に表示されないようにする。Fit Modelでは、モデルのあてはめウィンドウとレポートの両方が出力されなくなります。
レポートのタイトルを変更する	Platform name (...,title("*string*"))	レポートのタイトルを（*string*）に変更します。

表 10.1 分析プラットフォームのスクリプト（続き）

目的	コマンド	説明
自動再計算の設定を変更する	`Platform name (...,automatic recalc(Boolean))`	True（1）に設定すると自動再計算がオンになり、データ内での変更や除外された状態の変更が即座にレポートに反映される。False（0）に設定すると自動再計算がオフになり、分析のやり直し（`Redo Analysis`）を使用しないと変更を確認できません。 **注:** 自動再計算がオンの場合、再計算を実行させるには `wait(0)` コマンドを使う必要があります。
現在のプラットフォーム環境設定を使用しない	`Platform name (Ignore Platform Preference (Boolean))`	真（1）の場合、現在のプラットフォーム環境設定は適用されない。偽（0）の場合、現在のプラットフォーム環境設定が適用される。

表 10.2 プラットフォームウィンドウに送れるメッセージ

メッセージ	構文	説明
`Redo Analysis`	*obj*<<Redo Analysis	同じオプションのままで、プラットフォームを再起動する。
`Save Script to Data Table`	*obj*<<Save Script to Data Table	分析を再度実行するためのスクリプトを、関連するデータテーブルのプロパティとして保存する。
`Save Script to Report`	*obj*<<Save Script to Report	分析を再度実行するためのスクリプトを、レポートの最上部にあるテキストボックスに保存する。
`Save Script to Script Window`	*obj*<<Save Script to Script Window	分析を再度実行するスクリプトを、スクリプトウィンドウに保存する。
`Save Script for All Objects`	*obj*<<Save Script for All Objects	オブジェクトのウィンドウ内にある分析を再度実行するためのスクリプトを、スクリプトウィンドウに保存する。
`Get Script`	*x* = *obj*<<Get Script	分析を再度実行するためのスクリプトを、式で戻す。
`Data Table Window`	*obj*<<Data Table Window	関連するデータテーブルウィンドウをアクティブにする（最前面に出す）。
`Close Window`	*obj*<<Close Window	指定されたオブジェクト（*obj*）を表示しているウィンドウを閉じる。オブジェクトは通常、プラットフォームのウィンドウです。
`Move Window`	*obj*<<Move Window(*x*, *y*)	ウィンドウをスクリーン上の (x, y) の位置に移動する。
`Show Window`	*obj*<<Show Window	指定されたウィンドウを表示する。

表10.2 プラットフォームウィンドウに送れるメッセージ（**続き**）

メッセージ	構文	説明
Zoom Window	*obj*<<Zoom Window	ウィンドウのサイズをその内容の最大サイズに合わせる。
Scroll Window	*obj*<<Scroll Window(*x*, *y*) *obj*<<Scroll Window({*x*, *y*})	ウィンドウを、右方向に *x* ピクセル、下方向に *y* ピクセルスクロールする。負の座標の場合は左方向および上方向になります。座標が { } で囲まれたリストとして指定されている場合、その座標は絶対座標になります。ウィンドウは、左側から *x* ピクセル、最上部から *y* ピクセルの点にスクロールします。
Bring Window to Front	*obj*<<Bring Window To Front	指定したウィンドウを最前面に移動する。
Size Window	*obj*<<Size Window(*x*, *y*)	ウィンドウのサイズを、幅 *x* ピクセル、高さ *y* ピクセルに変更する。
Maximize Window	*obj*<<Maximize Window	ウィンドウを最大化する。ウィンドウの右上隅にある最大化ボタンをクリックするのと同じです。このメッセージはオプションでブール値の引数をとります。 `// ウィンドウを最大化する` `obj<<Maximize Window(1)` `// ウィンドウを元に戻す` `obj<<Maximize Window(0)`
Minimize Window	*obj*<<Minimize Window	ウィンドウを最小化する。ウィンドウの右上隅にある最小化ボタンをクリックするのと同じです。このメッセージはオプションでブール値の引数をとります。 `// ウィンドウを最小化する` `obj<<Minimize Window(1)` `// ウィンドウを元に戻す` `obj<<Minimize Window(0)`
Print Window	*obj*<<Print Window	指定したウィンドウを印刷する。
Get Window Size	obj<<Get Window Size	指定したウィンドウの表示サイズを取得する。幅と高さの値を含むリストを戻します。
Get Window Position	obj<<Get Window Position	指定したウィンドウの表示位置を取得する。縦と横のサイズをリストで戻します。

表 10.2 プラットフォームウィンドウに送れるメッセージ（続き）

メッセージ	構文	説明
Report TopReport	*obj*<<Report Report(*obj*)	プラットフォームウィンドウ内にあるレポートに対する、ディスプレイボックスへの参照を戻す。詳しくは「表示ツリー」の章の「ディスプレイボックスオブジェクトの参照」（384 ページ）の説明を参照してください。TopReport はディスプレイボックスの先頭に対する参照を戻します。この TopReport は、1 つのウィンドウに複数のプラットフォームのレポートが表示されている場合や、By グループごとに分析を行った場合に使うと便利です。
Journal Window	*obj*<<Journal Window	ウィンドウの内容をジャーナルに追加する。

すべてのプラットフォームで使用できるオプション

By

起動コマンド内での By 引数の使用は、ほとんどのプラットフォームでサポートされているので、By についてはプラットフォーム別に説明しません。一般的な説明については、「By グループレポート」（353 ページ）を参照してください。

Title

標準のタイトルを置き換える Title コマンドは、どのプラットフォームでも使用できます。例:

```
Bivariate(X(:Name(" 身長 ( インチ )")),Y(:Name(" 体重 ( ポンド )")), Title(" タイトル "));
```

Axes

グラフを含むプラットフォームでは、プラットフォームスクリプトの中で軸をカスタマイズできます。各プラットフォームのグラフに特有のオプションがリストされています。

Automatic Recalc

自動再計算がオンの場合、wait(0) コマンドを使ってこれを発動し、再計算を実行します。

追記

スプライン曲線

スプライン曲線を求める関数も用意されています。以下のいずれの関数でも、*x*は説明変数のベクトル、*y*は応答変数のベクトル、*lambda*は平滑化パラメータです。*lambda*の値が大きくなるほどなめらかなスプライン曲線になります。

```
coef = Spline Coef(x, y, lambda);
```

次の5列で構成される行列を戻します。

```
knots||a||b||c||d
```

節点（*knots*）は*x*の一意な値です。スプライン曲線による予測値は、後ほどSplineEval関数で説明する方法により、係数（a、b、cおよびd）を使って計算することができます。

```
yhat=Spline Smooth(x, y, lambda);
```

上のようにに指定されたSplineSmooth関数は、スプライン曲線のあてはめから計算される予測値を戻します。

```
yhat=Spline Eval(x, coef);
```

SplineEval関数は、SplineCoef関数から戻されたのと同じ形のcoef行列（つまり、knots||a||b||c||d）を使って、スプライン曲線の予測値を計算します。*x*引数は、SplineEval関数の場合は、スカラーでも行列でもかまいません。coefの列数は、1より大きい任意の値をとることができ、各列は次に大きなべき乗に使われます。*x*のべき乗は、knotの値で中心化されます。たとえば、coefがknots||a||b||c||dである場合、*j*を、knots[j]が*x*より小さいものの最大値となるような整数とし、xx = x-knots[j]とすると、予測値は次のように計算されます。

```
result = a[j]+xx*(b[j]+xx*(c[j]+xx*d[j]))
```

変形すると、次のようになります。

```
result = a[j] + b[j]*xx + c[j]*xx^2 + d[j]*xx^3
```

モデルのあてはめの効果

モデルの効果には複数の列のリストを指定でき、そのための特別な構文があります。

```
Effect( 効果のリスト、効果のマクロのリスト、またはその両方のリスト)
```

1つの列名、複数の列の交差にはアスタリスク（*）、枝分かれする列には添え字の括弧[ ]を使って効果を指定します。追加の効果オプションは、アンパーサンド（&）文字の後に指定できます。例をいくつか示します。

```
A,                  // 1つの列名単独で1つの主効果
A*B,                // 交差効果、交互作用、または多項式
A[B],               // 枝分かれ
A*B[C D],           // 交差および枝分かれ
```

```
effect&Random,       // 変量効果
effect&LogVariance, // 対数分散効果
effect&RS,           // 応答曲面の項
effect&Mixture,      // 配合の効果
effect&Excluded,     // モデル引数を持たない効果
effect&Knotted,      // 節点スプライン効果
```

効果のマクロは以下のとおりです。

```
Factorial( columns ),   // 完全実施要因
Factorial2( columns ),  // 最大 2 次までの交互作用のみ
Polynomial( columns ),  // 2 次の多項式のみ
```

起動ウィンドウを表示すると同時にモデルのあてはめを実行するには、スクリプトに Run Model を含めます。起動ウィンドウは表示するが、すぐにはモデルのあてはめを実行しない場合は、Run Model の代わりに Add Script を使用します。

注：「モデル」というスクリプトが保存されているデータテーブルで［**分析**］>［**モデルのあてはめ**］を選択すると、スクリプトで指定されている列の役割が起動ウィンドウに自動入力されます。

MANOVA の応答と効果

個別の応答関数（Response Function）の分析を指定するには、添え字付きの Response を使います。

```
manovaObj << (Response[1] << { 応答オプション });
manovaObj << (Response[" 対比 "] << { 応答オプション });
```

各応答関数で次のオプションが使用できます。

```
Custom Test( matrix, <Power Analysis( ... )>, <Label( "..." )> )
```

行列（*matrix*）の各行は、モデル内のすべての引数の係数を示します。

個別の効果の検定を指定するには、名前または番号の添え字をつけた Effect を使います。

```
manovaObj << (Response[1] << (Effect[" モデル全体 "] << { 効果オプション }));
manovaObj << (Response[1] << (Effect[i] << { 効果オプション }));
```

効果には次のように番号が付けられます。

- 切片には 0
- 標準の効果には 1、2、3、...
- 「モデル全体」の検定には *n*+1（*n* は切片を**含まない**効果の数）

これらの番号は各応答関数内の各効果で使用できます。行列（*matrix*）の各行は、効果のすべての水準に対する係数を示します。

```
Test Details( 1 ),
Centroid Plot( 1 ),
Save Canonical Scores,
Contrast( matrix, <Power Analysis(...)> )
```

以下に、「Dogs」データテーブルで動作するテストスクリプトを示します。

```
manObj = Fit Model(
   Y(Name("log(ヒスタミン0)"),Name("log(ヒスタミン1)"),Name("log(ヒスタミン
   3)"),Name("log(ヒスタミン5)")),
   Effects(ヒスタミンの消耗 y or n, 薬剤 , 薬剤 * ヒスタミンの消耗 y or n),
   Personality( MANOVA ),
   モデルの実行
);
manObj << response function( Contrast, Go );
manObj << (response[1] << CustomTest(
      [0 1 0 0,
       0 0 1 0,
       0 0 0 1]
) ); // モデル全体と同じ
manObj << (response["対比"] << (effect["モデル全体"] << TestDetails));
manObj << (response["対比"] << (effect[3] << TestDetails));
```

モデルのあてはめに送るコマンド

個々の応答変数のあてはめにコマンドを送るには、次のような構文を使います。

```
fitObj << (responseName << {オプション , ...});
```

Send コマンド内の Send コマンドは、指定された応答を見つけ、コマンドのリストをその応答に送ります。1 つの Send コマンドでオプションを直接 *fitObj* に送ると、そのオプションはすべての応答に送られます。

個々の効果にコマンドを送るには、Send コマンドをさらに入れ子にします。

```
fitObj << (responseName << ((effectName) << effectOption));
```

DOE（実験計画）のスクリプト

通常、DOE プラットフォーム（実験計画）はインタラクティブな操作に基づいて実行されますが、スクリプトによって実行することもできます。DOE の結果は、「モデル」という名前のテーブルプロパティを含むデータテーブルになります。このテーブルプロパティは、モデルのあてはめを起動し、その計画に適した設定をするためのスクリプトです。

JMP の［スクリプトの索引］で、DOE コマンドをすべて含んだリストを参照することができます。［**ヘルプ**］>［**スクリプトの索引**］を選択し、メニューから［**オブジェクト**］を選択し、DOE を検索します。

調整コマンド

「カスタム計画」における設定を、予め決めておくことができるスクリプトコマンドがあります。以下のスクリプトコマンドについてはここで簡単に説明します。例は、『実験計画 (DOE)』にも記載されています。

DOE Mixture Sum 配合実験において、ある 1 つの成分を一定の割合で常に加える場合には、他の配合成分の合計は 1 未満でなければなりません。因子を定義する際には、定数の配合成分の割合を定数因子として入力します。たとえば、配合因子の定数値が 0.1 のときは、次のコマンドになります。

```
DOE Mixture Sum = 0.9;
```

このコマンドにより、残りの配合因子の合計がデフォルトの 1.0 ではなく、0.9 に制限されます。

DOE Starts 「カスタム計画」で使われるランダム開始点の数です。デフォルトの開始点の数だけでは必要な計画を生成しない場合があります。`DOE Starts` コマンドを使うと、開始点の数を増やすことができます。たとえば、次の JSL ステートメントを実行します。

```
DOE Starts = 100;
```

このステートメントを実行すると、デフォルトの開始点の数は無効になり、開始点の数は 100 に設定されます。

DOE Starting Design 例:

```
DOE Starting Design = matrix;
```

これは、ランダム開始点の計画を指定した行列に置き換えます。開始の計画が指定された場合、その計画だけから検索を開始します。

DOE K Exchange Value 座標交換アルゴリズムは、デフォルトでは、すべての反復における交換に対して、すべての行を考慮します。しかし、行の一部は交換されることがないかもしれません。次に例を示します。

```
DOE K Exchange Value = 3;
```

小さな値（この例では 3）を設定すると、座標交換アルゴリズムは、反復ごとの交換について最も可能性のある、指定された値の行数（この例では 3 行）だけを考慮するようになります。

DOE Bayes Diagonal 例:

```
DOE Bayes Diagonal = vector;
```

このベクトルは、D 最適計画を見つけるための $X'X$ 行列の対角要素を変更するときに使われます。指定されたベクトルは、$X'X$ 行列の現在の対角要素に追加されます。

DOE Sphere Radius 「カスタム計画」の領域を超立方体ではなく超球に制約します。

```
DOE Sphere Radius = n
```

n は、制約領域である球の半径を示します。

三次元散布図のスクリプト

三次元散布図で［**ハードウェアアクセラレーションを使用**］を設定するには、Scene3DHardwareAcceleration コマンドを使用します。例:

```
dt = Open( "$SAMPLE_DATA¥solubility.jmp" );
Scene3DHardwareAcceleration = 0;
dt << Scatterplot 3D(
   Y( :Name( "1- オクタノール " ), : エーテル , : クロロフォルム , : ベンゼン , : 四塩化炭素 , :
   ヘキサン )
);
```

管理図

警告スクリプトの実行

警告スクリプトは、1つ以上のテストで異常が検出された場合に警告をします。警告スクリプトおよびそれに続く JSL スクリプトでは、次の変数を使用できます。

qc_col（列の名前）

qc_test（異常が検出されたテスト）

qc_sample（標本番号）

qc_firstRow（標本の第 1 行）

qc_lastRow（標本の最終行）

例 1: ログへの書き込みを自動化する

警告が自動的に実行されるようにする1つの方法は、作成したスクリプトをデータテーブル内に「**QC Alarm Script**」というテーブルプロパティとして保存する方法です。テストで異常が検出されたときに自動的にログにメッセージが書き込まれるようにするには、次の作業を行います。

1. 下記のスクリプトを実行し、スクリプトをデータテーブルのプロパティとして保存します。
2. 管理図を作成します。
3. 適用するテストを選択します。テストに合格しなかった標本があれば、ログにメッセージが表示されます。

```
CurrentData Table()<<Set Property("QC Alarm Script",
   Write(match(
     QC_Test, 1,"1 点が A ゾーンを超えている ",
            2," 連続した 9 点が C ゾーン以上にある ",
            3," 連続した 6 点が常に増加または
               減少している ",
            4," 連続した 14 点が交互に上がったり
               下がったりしている ",
            5," 連続した 3 点のうち 2 点が A ゾーン
               以上にある ",
```

```
        6,"5 点のうち 4 点が B ゾーン
           以上にある ",
        7," 連続した 15 点が C ゾーンにある ",
        8," 連続した 8 点が
           C ゾーンには 1 点もない " )));
```

例 2: 管理図のテスト結果を音声で通知させる

```
dt = Open("$SAMPLE_DATA/Quality Control/Coating.jmp");
Control Chart(Alarm Script(Speak(match(
      QC_Test,1, "1 点が A ゾーンを超えています ",
      QC Test,2," 連続した 9 点が C ゾーン以上にあります ",
      QC_Test,5," 連続した 3 点のうち 2 点が A ゾーン以上に
               あります "))),
Sample Size( :サンプル), Ksigma(3), Chart Col( :重量 ,
XBar(Test 1(1), Test 2(1), Test 5(1)), R));
```

これらのスクリプトで、JSL 警告コマンドの Speak、Write、Mail を使用することができます。

注：Windows で音声による警告を実行させるには、Microsoft Text-to-Speech エンジンがシステムにインストールされている必要があります。

例 : $\bar{X}$ 管理図と R 管理図

次の例は、「Coating.jmp」データを使っています（出典は、*ASTM Manual on Presentation of Data and Control Chart Analysis*）。分析対象となる品質特性は「**重量**」列で、サブグループの標本サイズは 4 に設定してあります。

次のスクリプトを実行し、XBar 管理図と R 管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Coating.jmp");
Control Chart(Sample Size( :サンプル), KSigma(3), Chart Col( :重量 , XBar, R));
```

サンプル 6 を見ると、工程が統計的に管理された状態にないことが示唆されています。標本の値を調べるには、どちらかの管理図上でサンプル 6 の点をクリックします。すると、対応する行がデータテーブル内で強調表示されます。

例 : サブグループ標本のサイズが異なるときの $\bar{X}$ 管理図と S 管理図

この例も「Coating.jmp」を使いますが、今回、分析対象となるのは「**重量 2**」列です。

次のスクリプトを実行し、XBar 管理図と S 管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Coating.jmp");
Control Chart(Sample Size( :サンプル), KSigma(3), Chart Col( :重量 2, XBar, S));
```

データの「**重量 2**」列にはいくつか欠測値があるため、管理限界の値が一定ではありません。どのサンプルもオブザベーションの数は同じですが、サンプル 1、3、5、7 には欠測値が含まれています。

例 : 個々の測定値と移動範囲の管理図（IR 管理図）

サンプルデータの「Quality Control」フォルダの「Pickles.jmp」データは、ピクルスのバットに含まれた酸度をまとめたものです。ピクルスは酸に敏感な上に大きなバットで製造されるため、酸度が高すぎるとバット全体が不良品になってしまいます。4つのバット内の酸度を、毎日午後1時、2時、3時に計測し、日付、時刻、酸度の測定値をデータテーブルにまとめました。

次のスクリプトを実行し、管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Pickles.jmp");
Control Chart(Sample Label( :日付 ), GroupSize(1), KSigma(3), Chart Col( :酸度 ,
  Individual Measurement, Moving Range));
```

例 : UWMA 管理図

「Clips1.jmp」データテーブルを検討してみましょう。分析対象は、製造された金属製クリップの取っ手の間隔です。工程における平均間隔の変化を監視するために、毎日5つのクリップを選んでサブグループとし、移動平均の範囲を3に設定してUWMA管理図を作成します。

次のスクリプトを実行し、UMWA管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Clips1.jmp");
Control Chart(Sample Size(5), KSigma(3), Moving Average Span(3), Chart Col( :取っ手
  の間隔 , UWMA));
```

第1日の点は、最初のサブグループ標本、つまり第1日に取った5つの値の平均です。第2日の点は、第1日と第2日のサブグループ平均を合わせて計算した平均を表します。第2日以降の点は、その日と前2日間のサブグループ平均を合わせた平均です。

取っ手の間隔の平均値に減少傾向が見られますが、3σ管理限界の外まで落ち込んだ点はありません。

例 : EWMA 管理図

次のスクリプトを実行し、「Clips1.jmp」のEMWA管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Clips1.jmp");
Control Chart(Sample Size(5), KSigma(3), Weight(0.5), Chart Col( :取っ手の間隔 ,
  EWMA));
```

例 : NP 管理図

「Quality Control」フォルダ内の「Washers.jmp」データは、亜鉛メッキが施されたワッシャー15ロット、各400個のうちの不適合品の個数をまとめたものです。検査されたのは、亜鉛メッキが粗い、鉄鋼が露出している、などの仕上げの不良です。仕上げに不良のあるワッシャーは不適合品とみなされます。不適合品の度数は、400個から成る各ロットに何個の不適合ワッシャーがあるかを示します。

次のスクリプトを実行し、**NP**管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Washers.jmp");
Control Chart(Sample Size(400), KSigma(3), Chart Col( :Name("不適合数"), NP));
```

例 : P 管理図

引き続き「Washers.jmp」データを使い、ここでは標本サイズを表す変数を指定します。これにより、異なる標本サイズの使用が可能になります。

次のスクリプトを実行し、**P** 管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Washers.jmp");
Control Chart(Sample Label( :ロット), Sample Size( :ロットサイズ), K Sigma(3),
  Chart Col(Name(" 不適合数 "), P))
```

点は **NP** 管理図と同じですが、*Y* 軸、平均、および限界値は割合をもとに計算されるため、異なっています。

例 : U 管理図

「Quality Control」フォルダ内の「Braces.jmp」データは、自動車を支える留め金具の各ケース内の不適合数を記録したものです。検査単位は 1 ケース分の留め金具です。ユニットサイズ（サブグループ標本のサイズ）は検査されるケースの数（1 日あたり）で、日によって異なります。**U** 管理図を作成すると、ユニットサイズあたりの留め金具の不適合数を監視することができます。上側管理限界と下側管理限界は、ユニットサイズ（検査単位の数）によって異なります。

次のスクリプトを実行し、**U** 管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Braces.jmp");
Control Chart(Sample Label( :日付), Unit Size( :ユニットサイズ), K Sigma(3), Chart
  Col( :Name(" 不適合数 "), U));
```

例 : C 管理図

C 管理図は、サブグループ標本内における不適合数を監視するという点で **U** 管理図と似ています。**C** 管理図では、検査単位あたりの平均不適合数を監視することもできます。

この例ではシャツの品質を取り上げましょう。あるアパレルメーカーは、シャツを 10 枚ずつ箱に詰めて出荷します。出荷の前に、シャツにきずがないかどうかが検査されます。シャツ 1 枚あたりのきずの平均数を調べるため、箱あたりのきずの数を 10 で割った値を記録しました。

次のスクリプトを実行し、**C** 管理図を作成します。

```
Open("$SAMPLE_DATA/Quality Control/Shirts.jmp");
Control Chart(Sample Label( :箱), Sample Size(:Name("1 箱の枚数 ")), K Sigma(3),
  Chart Col( :きずの数 , C));
```

フェーズ

「フェーズ（phase）」とは、データテーブル内の連続するオブザベーションをグループにまとめたものです。たとえば、新しい工程で生産を開始する前と後は異なるフェーズだと定義できます。指定したフェーズ変数の水準ごとに新しいシグマ、限界値のセット、ゾーン、テストの結果が計算されます。

管理図で使用しているフェーズ変数の水準に対し、JSL で限界値を設定するには、専用の構文が必要です。「Diameter.jmp」の各フェーズに対し、限界値を設定する例を次に示します。

```
Control Chart(
      Phase( :フェーズ ),
      Sample Size( :日付 ),
      KSigma(3),
      Chart Col(
            :直径,
            XBar(
                  Phase Level("1", Sigma(.29), Avg(4.3),LCL(3.99),UCL(4.72)),
                  Phase Level("2", Sigma(.21), Avg(4.29),LCL(4),UCL(4.5))),
            R(
                  Phase Level("1"),
                  Phase Level("2"))

));
```

第 11 章

表示ツリー
ウィンドウの作成と使用

スクリプトは、柔軟な操作を行うことができます。この章では、ウィンドウの構成要素を作成し操作する方法について示します。表示（ディスプレイ）に関する以下のようなスクリプトについて説明します。

- レポートウィンドウ内の項目を操作する方法
- 独自に作成した出力や、標準のレポートを組み合わせたレポートウィンドウを構築するための方法

注：ディスプレイボックスの中には作成および使用が可能なものと、作成はできず、使用のみ可能なものがあります。

作成できるディスプレイボックスのリストを見るには、［**ヘルプ**］メニューから［**スクリプトの索引**］>［**関数**］>［**Display**］を選択します。

表示ツリーに表示できるディスプレイボックスのリストを見るには、［**ヘルプ**］メニューから［**スクリプトの索引**］>［**ディスプレイボックス**］を選択します。このリストにあって［**関数**］の中にないディスプレイボックスについては、メッセージを送ることはできますが、スクリプト内で作成することはできません。それらは、通常、他のディスプレイボックスのサブボックスとして作成されます。

2次元プロットおよび3次元プロットに関する詳細については、「スクリプトによるグラフ作成」（461ページ）および「3D シーン」（499ページ）を参照してください。

目次

表示の操作 379
ディスプレイボックスの紹介 379
ディスプレイボックスオブジェクトの参照 384
メッセージを送る 387
ビルトインウィンドウにアクセスするには 397
選択ウィンドウの使用 397
ディレクトリ内のファイル 398
表示ツリーの作成 399
基本事項 400
既存の表示の更新 401
インタラクティブな表示要素 404
モーダルウィンドウと非モーダルウィンドウ 409
作成した表示にメッセージを送る 425
独自の表示を始めから作成する 426
プラットフォームを含むディスプレイボックスの作成 427
カスタムプラットフォームの作成 429
シート 432
ジャーナル 434
Picture 表示タイプ 435
モーダルウィンドウ 435
モーダルウィンドウの作成 436
汎用モーダルウィンドウ 436
廃止された Dialog を New Window に変換する 437
Dialog と New Window の違い 442
ダイアログおよび列ダイアログの作成 448
スクリプトエディタのスクリプト 451
構文リファレンス 452

表示の操作

JMP のレポートは、さまざまな種類の長方形のボックスを入れ子にしたり、並べたりすることにより、階層的な形で作成されます。スクリプトを使ってレポートを操作するには、これらのボックスの動作について多少学習する必要があります。また、以下のような場合は、さらに詳しく学習する必要があります。

- 既存のレポートのディスプレイボックスを操作する。

  この節では、JMP レポートについて概要を紹介した後で、レポートをナビゲートする方法、レポートからデータを抽出する方法、および JSL からレポートを変更する方法を説明します。

- 独自のレポートを構築する。

  まず、この節の概要をよく読んでから、「表示ツリーの作成」（399 ページ）を参照してください。

ヒント：［一般］環境設定の［**無効なディスプレイボックスメッセージを報告**］オプションは、無効なディスプレイボックスメッセージを特定するのに役立ちます。

- このオプションは、デフォルトではオフになっています。このオプションがオフで、スクリプトに無効なディスプレイボックスメッセージが含まれている場合、JMP はそれらのエラーを無視し、スクリプトを実行し続けます。
- この環境設定がオンの場合、無効なディスプレイボックスに達するとスクリプトの実行は停止され、ログにエラーが送られます。

このオプションは、スクリプトの開発中は役立ちますが、実際に使用するにあたっては、無効なメッセージが使われたかどうかは不要な情報であることが多いです。

ディスプレイボックスの紹介

以下に、代表的な JMP ディスプレイボックスのダイアグラムを示します。

表 11.1 代表的なディスプレイボックス

縦方向ボックス（`V List Box`）では、ボックスが縦方向に並べられます。

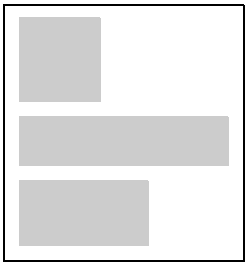


表 11.1 代表的なディスプレイボックス（**続き**）

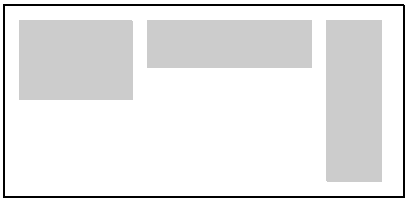
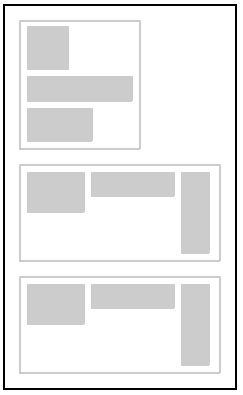
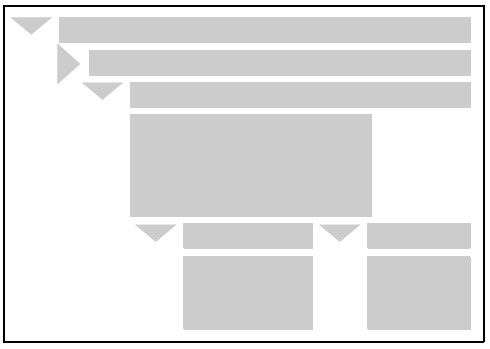
横方向ボックス（H List Box）では、ボックスが横方向に並べられます。	
縦方向ボックス（V List Box）と横方向ボックス（H List Box）を入れ子にすることができます。この例では、1つの縦方向ボックス（V List Box）の中に、1つの縦方向ボックス（V List Box）と2つの横方向ボックス（H List Box）が並べられています。	
テーブルボックス（Table Box）は特殊な横方向ボックス（H List Box）で、文字列および数値列から構成されます。	
アウトラインボックス（Outline Box）では、アウトラインによる階層が作成されます。	

表 11.1 代表的なディスプレイボックス（続き）

ピクチャーボックス（Picture Box）では、軸、フレーム、およびラベルを合わせてグラフが構成されます。	

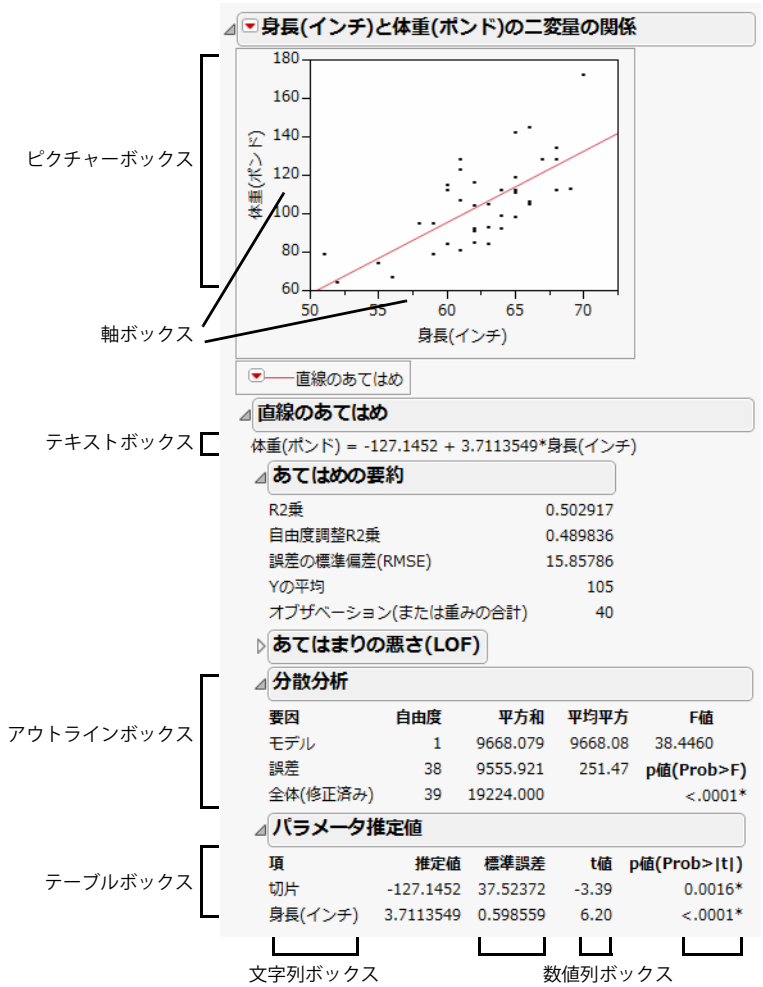
注：JMP で複数の子を持てるコンテナは、横方向ボックス（H List Box）、縦方向ボックス（V List Box）、アウトラインボックス（Outline Box）、パネルボックス（Panel Box）の 4 つのみです。他のコンテナでは、子が 1 つしか使えません。

JSL には、この他に、ボタンボックス（Button Box）、スライダボックス（Slider Box）、範囲スライダボックス（Range Slider Box）、タブボックス（Tab Box）、グローバルボックス（Global Box）などのディスプレイボックスがあり、表示をカスタマイズすることができます。これらについては、「インタラクティブな表示要素」（404 ページ）で説明しています。また、編集可能なテキストボックス（Text Edit Box）と、特性要因図プラットフォームから作成したものと同じツリーを作成するボックス（Hier Box）も用意されています。

各ボックスは、レポートを表示するとき、そのボックスに含まれている各構成要素の大きさを問い合わせ、それらの位置を適切に配置します。

次に示すのは、「Big Class.jmp」の「二変量の関係」レポートです。ディスプレイボックスの構造がわかるように注釈がついています。

図11.1 レポートのディスプレイボックス



ボックスの種類と表示内容の配置方法については、表11.5（453ページ）で説明しています。構文リファレンスの中で「最下位」と記載されたボックスは、内部に他のボックスを含まないボックスです。

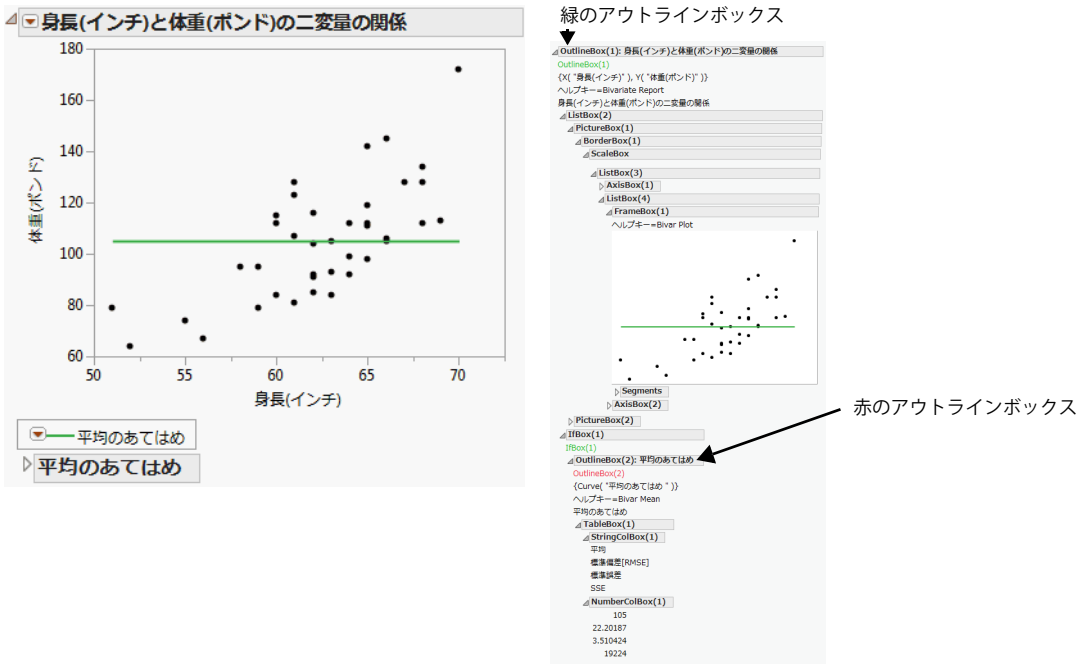
ディスプレイツリーの表示

レポートのディスプレイボックスツリーを表示するには、次の手順に従います。

1. レポートウィンドウのサイズを大きくします。
2. Ctrl キーと Shift キーを押しながら、レポートの空白の部分を右クリックします。

   レポートの右側か下をクリックできます。レポートウィンドウが非常に大きい場合は、レポートのサイズがウィンドウよりも小さくなるまでいくつかのアウトラインノードを閉じます。
3. ［**ツリー構造の表示**］を選択します。

図11.2 ツリー構造の表示

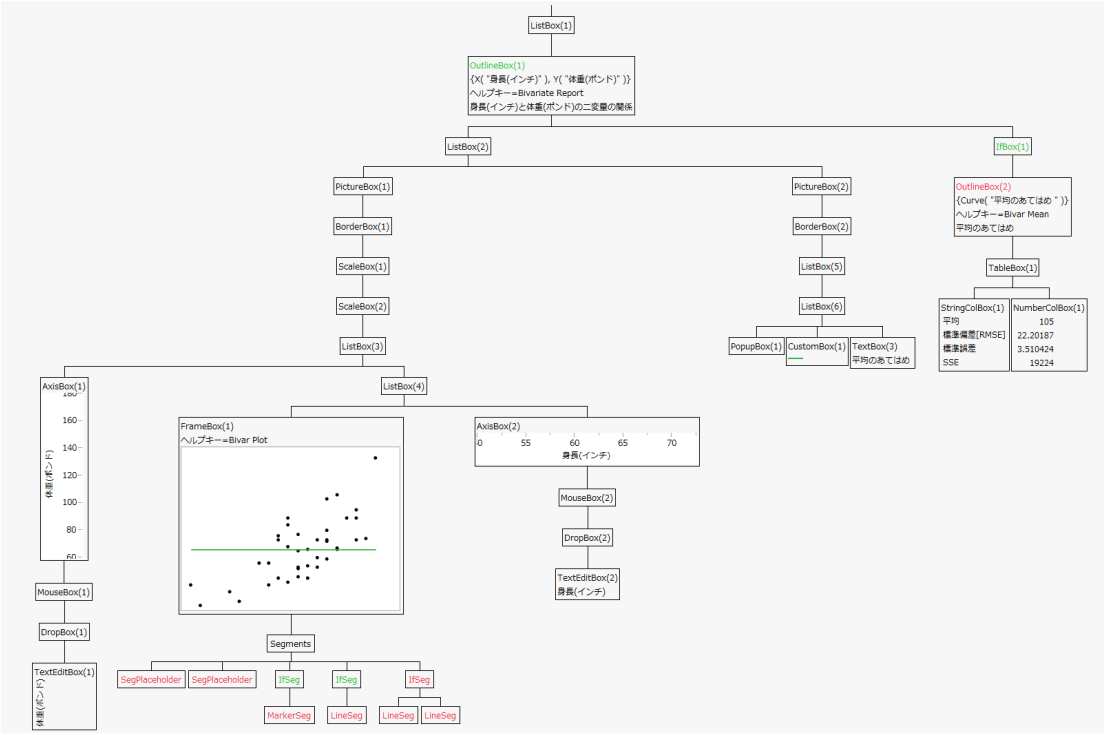


この例では、赤いアウトラインボックスは「平均のあてはめ」アウトラインノード（例では閉じています）に、緑のアウトラインボックスは「身長（インチ）と体重（ポンド）の二変量の関係」アウトラインノードに対応しています。他のボックスは、それぞれレポート上の他のボックスに対応しています。

従来型のツリーを表示するには、Shiftキーを押したまま空白の部分を右クリックして、［ツリー構造の表示］コマンドを表示させます。その後、Shiftキーだけを押しながら、［**ツリー構造の表示**］を選択します。

ディスプレイボックスの上にカーソルを置くと、階層グループのアウトラインが表示されます。

図11.3 従来型のツリー構造



スクリプトを使ってツリー構造を表示することもできます。そのためには、任意のレポートに `<<showtreestructure()` メッセージを送ります。特定の部分のツリー構造を表示する場合は、このメッセージをレポートの該当部分（ディスプレイボックスオブジェクト）に送ります。

ディスプレイボックスオブジェクトの参照

ディスプレイボックスの参照と呼ばれる特殊なJSL値は、ディスプレイボックス（DisplayBox）を参照します（このマニュアル全体を通して、構文のまとめでは、*db*をディスプレイボックスの参照、*dt*をデータテーブルの参照、*obj*をスクリプトオブジェクトの参照のプレースホルダとして使っています）。

`Report` メッセージを使ってディスプレイボックスの参照を作成し、スクリプト可能なプラットフォームに関連付けられた表示ツリーの親にアクセスできます。

```
variable = platform object<<report;
```

たとえば、二変量分析のレポートへの参照を取得するには、次のようにします。

```
dt= Open("$SAMPLE_DATA¥Big Class.jmp");
biv = bivariate(x(:Name(" 身長 ( インチ )")),y(:Name(" 体重 ( ポンド )")),fit mean, fit
  polynomial(4));
rbiv = biv<<report;
```

注：プラットフォームへの参照と**レポート**への参照を混同しないようにしてください。これらのオブジェクトはタイプが異なり、受け取ることのできる JSL メッセージのタイプも異なります。レポートは、たとえば、イメージとしてコピーしたり、ディスプレイボックスを選択したり、アウトラインノードを閉じたりするなどの操作を実行できます。一方、プラットフォームは、検定を実行したり、直線や曲線を描画したり、ウィンドウ全体を閉じたりするなどの操作を実行できます。

添え字の使用

レポートのある部分への参照を作成するには、添え字の演算子を使うのが最も簡単です。以下に示すような方法で参照を作成することができます。これらの例では、指定された文字列に一致したディスプレイボックスを見つけ出します。"text" の部分には、文字列を含む任意の式を指定することもできます。次のような形式で添え字を使った参照を記述します。

```
var = db["text"];
```

上の行は、*text* で指定されたタイトルの *db* 内のディスプレイボックスを探して、それを変数（var）に割り当てます。タイトル（*text*）には、タイトルの一部ではなく完全な文字列を指定する必要があります。

通常、メッセージを簡単に送れるように、レポートの一部を変数に割り当てます。割り当ては、次のような指定で割り当てられます。

```
Variable = rpt ["search string"];
```

たとえば、二変量のレポートで分散分析の表を見つけるには、以下のようにします。

```
r1=rbiv[" 分散分析 "];
```

以下の節で、添え字演算子を使ってディスプレイボックスを参照するさまざまな方法について説明します。

ワイルドカード文字列

文字列の一部とワイルドカード文字（? など）を使って、タイトルの残りの部分と一致させることができます。次の例は、「平均」で始まる文字列を探し、「平均のあてはめ」を見つけ、平均のあてはめのアウトラインを開きます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate( Y( :Name(" 体重 ( ポンド )") ), X( :Name(" 身長 ( インチ )") ), Fit Mean(
  {Line Color( {57, 177, 67} )} ) );
rbiv = biv << report; // 「二変量」レポートへの参照を戻す
out2 = rbiv[" 平均 ?"]; //「二変量」レポート内の「平均のあてはめ」を見つける
out2 << Close( 0 ); //「平均のあてはめ」のアウトラインを開く
```

ボックスタイプとワイルドカード文字列

前述の例と同じ機能を実行する別の方法を紹介します。この方法では、ディスプレイボックスの種類とワイルドカード文字列で目的の部分を絞り込みます。例:

```
out2 = rbiv[Outline Box(" 平均 ?")];
```

上の行は、rbiv レポート内で指定したテキスト文字列を含むアウトラインボックスを見つけ、それを out2 変数に割り当てます。

ボックスタイプとインデックス

ディスプレイボックスを見つける別の方法としては、*boxType* でインデックス番号 *n* を指定する方法があります。

```
db[boxType(n)]
```

上の行は、指定の種類（*boxType*）のディスプレイボックスのうち、*n* 番目のものを見つけます。例:

```
lb6 = rbiv[List Box(5)];
```

上の行は、rbiv レポート内の 6 番目のリストボックスを見つけ、それを lb6 変数に割り当てます。

ヒント：ディスプレイボックスのインデックス番号を特定するには、「入れ子と優先順位」（390 ページ）で説明している方法でツリー構造を表示します。

入れ子になったディスプレイボックス

レポート内で複数の階層下に表示されるディスプレイボックスを参照するには、次の形式を使用します。

```
db[arg1, arg2, arg3, ...]
```

最後の引数に該当するものを、最後から 2 番目の引数のアウトラインノードにあるディスプレイボックス内で見つけ、最後から 2 番目の引数に該当するものは、最後から 3 番目の引数のアウトラインノードにあるディスプレイボックス内で見つけ ... と続きます。これは、アウトラインツリーの階層を徐々に下がりながら入れ子になっているディスプレイボックスを探す方法です。

アウトラインボックス内で複数の層になっている項目を識別するには、前述の方法において、複数の添え字を指定します。JMP は、最初の項目を見つけた後、*その項目内で*次の項目を順次探していきます。

```
db[arg1,arg2,arg3]    /* 最初の項目を見つけ、その後、その位置から始めていく */
db[arg1][arg2][arg3] // カンマで区切った場合と同じ
```

また、異なる種類の添え字を一緒に並べることができます。

以下のコード行では、ノード「多項式のあてはめ 次数=4」の下のアウトラインノード「パラメータ推定値」の最初のテーブルの 3 番目の列を選択しています。

```
rbiv[" 多項式のあてはめ次数 =4"][" パラメータ推定値 "][1][3]<< select << reshow;
```

子のないディスプレイボックス

子のないディスプレイボックスは添え字可能ではありません。たとえば、次のチェックボックスは子を持たないため、添え字の指定をするとエラーとなります。最後の行のように、添え字を指定しないコードが正しいです。

```
New Window( "例",
cb = Check Box( {"One", "Two", "Three"} ) );
cb << Set( 3, 1 ); // 3番目のチェックボックスを選択する
Show( cb[1] << Get( 3 ) ); // これは誤り
Show( cb << Get( 3 ) ); // これが正しい
   cb << Get(3) = 1; // ログ出力
```

注：アウトラインボックス（OutlineBox）に対する番号は、OutlineBoxが開いていても閉じていても変わりません。一方、IfBoxが閉じている場合は、そこに含まれる要素は完全に除去されます。

ワイルドカード

指定する文字列の中には、どんな文字の代わりにもなるワイルドカード文字「?」を使用できます。分析対象の列によってタイトルが異なるような結果に対する汎用スクリプトを書きたいときは、ワイルドカードが重要な役割を果たします。

次のスクリプトは、「身」で始まる任意の列に対する「一変量の分布」の例です。

```
dist = Distribution(Y(Column("身?")));
```

また、「ワイルドカード文字列」（385ページ）に示すように、タイトルの一部が一致するワイルドカード文字を使用してアウトラインボックスを見つけることもできます。

メッセージを送る

表示の参照は、Sendまたは<<演算子を使ってスクリプトを表示要素に送るときに使用できます。たとえば、out2がアウトラインノードへの参照の場合、out2にノードを閉じるように要求できます。

```
out2<<close; // アウトラインノードを閉じる
```

Closeは、ウィンドウの三角形の開閉ボタンをクリックする場合とまったく同様に、アウトラインノードの開閉を交互に切り換えます。「閉じるまたは閉じたまま」にするには1、「開くまたは開いたまま」にするには0を指定します。

```
rbiv["平均のあてはめ"]<<close;     // 切り換え
rbiv["平均のあてはめ"]<<close(1); // 閉じるまたは閉じたまま
rbiv["平均のあてはめ"]<<close(0); // 開くまたは開いたまま
```

Select、Reshow、およびDeselectを使うと、ディスプレイボックス上で選択を示す反転を表示させたり解除させたりすることができます。複数の<<節を一緒に並べて、1つのオブジェクトに複数のメッセージを1行で送ることができます。結果は左から右に読んでいきます。たとえば、次の例の場合、まず、Selectが処理され、次にReshow、その次にDeselect、その次にもう一つのReshowが処理されます。例:

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
biv = Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ), Fit Mean(
   {Line Color( {57, 177, 67} )} ) );
rbiv = biv <<  report;
out2 = rbiv[Outline Box("?Mean")];
```

```
out2 << Close(0);
scbx = rbiv[String Col Box(1)];
for(i=0,i<20,i++,
   wait(.25);
   scbx<<select <<reshow <<deselect <<reshow
   );
```

ディスプレイボックスでできることを調べる

指定したディスプレイボックスの参照が処理できるメッセージを調べるには、以下の例のように、ディスプレイボックスのオブジェクトに Show Properties を使います。例:

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
biv = Bivariate( Y( :Name("" 体重 ( ポンド )"")) , X( :Name("" 身長 ( インチ )"") ), Fit
  Mean({Line Color( {57, 177, 67} )} ) );
rbiv = biv <<  report;
Show Properties(rbiv[frame box(1)]);
```

上のスクリプトは、二変量のレポートを作成し、それを rbiv 変数に割り当てます。指定したディスプレイボックスのメッセージの一覧がログウィンドウに表示されます。以下は、表示されるログウィンドウを一部抜粋したものです。

```
Row Colors [ 色 ](Sets the color of the selected rows.)
Row Markers [ マーカー ](Sets the marker of the selected rows.)
Row Hide and Exclude [ アクション ](Hides and excludes (or unhide or unexcludes) the
  corresponding rows in the data table.)
Row Exclude [ アクション ](Excludes (or unexcludes) the corresponding rows in the
  data table.)
Row Hide [ アクション ](Hides (or unhides) the corresponding rows in the data table.)
Row Label [ アクション ](Labels (or unlabels) the corresponding rows in the data
  table.)
Row Legend [ アクション ](Color the rows according to a data column, and insert a
  legend to the right of this frame.)
Row Editor [Action](Bring up the Row Editor window, starting on the first selected
  point.)
Select Matching Cells [ アクション ](Select points that have similar labels to the
  selected rows)
Name Selection in Column [ アクション ](Label the currently selected rows and save
  the value(label) in a column.)
Select Similar [ アクション ] [ スクリプトの場合のみ ]
Get Background Color [ 色 ] [ スクリプトの場合のみ ]
Set Background Color [ 色 ] [ スクリプトの場合のみ ]
Get Background Fill [ ブール ] [ スクリプトの場合のみ ]
Set Background Fill [ ブール ] [ スクリプトの場合のみ ]
Background Color [ 色 ](Change the graph's background color.)
Background Map [ アクション ]
...
```

一部のメッセージはスクリプトでしか使用できません。ディスプレイボックスに使用できるメッセージがわかれば、指定のボックスに対してそれを送ればいいだけです。

たとえば、この節の最初のスクリプトを実行した後、次の行を rbiv レポートに送ります。

```
fbx=rbiv[frame box(1)];
fbx <<Set Background Color(blue);
```

すると、グラフのフレームボックスの背景色が青に変わります。これは、グラフを右クリックして［**背景色の設定**］から青色を選択するのと同じです。

実際に送ることのできるメッセージは、そのオブジェクトのコンテキストメニューの項目のほかにも、いくつかあります。各種のディスプレイボックスへのメッセージについては、次の節の「表示ツリーの作成」（399 ページ）で、さらに詳しく説明します。

注：Show Properties は、データテーブルやプラットフォームでも動作します。「データテーブル」の章の「データテーブルオブジェクトに送ることができるメッセージ」（260 ページ）および「プラットフォームのスクリプト」の章の「オブジェクトが対応するメッセージ」（359 ページ）を参照してください。

<<演算子の使用

send <<演算子は、すでに作成された表示（ディスプレイ）に対してだけでなく、まだ作成段階の表示に対してもメッセージを送ることができます。<<演算子を使うと、指定されているものが、それに含まれる子オブジェクトなのか、それともオプションなのかを明確に区別することができます。また、グラフ内において、どれがオプションで、どれが実行されるスクリプトであるのかも明確にすることができます。

たとえば、バージョン 4 では、H List Box の引数に指定できるのはボックスだけでした。しかし、バージョン 5 以降では、<<演算子を使って、HListBox の引数としてコマンドを指定できるようになりました。たとえば、H List Box に journal コマンドを挿入するには、次のようにします。

```
New Window("Title",
   HListBox(
      ...
      <<journal,
      ...
   )
);
```

もう 1 つの例として、GraphBox は次のように通常の名前付き引数をとります。

```
New Window("Title", Graph Box(FrameSize(400,400),XScale(0,25),yScale(0,25),
      <<Background Color("Red")));
```

名前付き引数の代わりに、send 演算子を使ってグラフボックス（Graph Box）にコマンドを送ることもできます。

```
New Window("Title",
   Graph Box(
      <<Frame Size(400,400),
      <<XAxis(0,25),
      <<YAxis(0,25),
      <<Background Color("Red")
));
```

入れ子と優先順位

コマンドを組み合わせた場合、左から右へと順にコマンドが評価されます。

```
box<<command1<<command2<<command3;
```

上のスクリプトは、まず command1 を box に送り、次に command2 を box に送り、最後に command3 を box に送ります。次の command が送られる時点で、前のコマンドにより box が変更されている可能性があります。

```
( (box<<command1) <<command2) <<command3
```

上のスクリプトは、command1 を box に送り、結果を取得します。その結果に command2 が送られ、command2 の結果に command3 が送られます。

```
x = box<<command1<<command2<<command3;
```

command3 の結果が変数 x に割り当てられます。最初の 2 つのコマンドは、box を変更した可能性がありますが、x には割り当てられません。

```
x = Text Box( "nothing" );
Print( x << Set Text( "the" )
   << Set Text( "first" )
   << Set Text( x << Get Text() || "thing" )
   << Get Text()
);
   "firstthing"
```

複数のコマンドを組み合わせる場合は、スクリプトが意図したとおりに実行されるように、括弧を使ってコマンドをグループ化します。

ウィンドウをすばやく見つける

zoom window メッセージを送ると、他のウィンドウの下に隠れているウィンドウを見つけることができます。

```
rbiv<<zoom window;
```

レポートのカスタマイズ

レポートの外観をカスタマイズするには、**Send To Report** コマンドの中で **Dispatch** コマンドを使います。たとえば、アウトラインノードの開閉、グラフフレームのサイズ変更、グラフフレーム内の色のカスタマイズなどに、これらのコマンドを使います。

Send To Report と **Dispatch** の例を見るには、いずれかの分析を実行し、レポートのデフォルトの外観を変更します。次に、［**スクリプト**］>［**スクリプトをスクリプトウィンドウに保存**］を選んで、スクリプトを確認します。

Send To Report には、表示ツリーに送られるコマンドのリストが含まれます。次の例では、Send To Report に 2 つの Dispatch コマンドが含まれています。

Dispatch は、コマンドを表示ツリーの特定の部分に送るのに使います。Dispatch には 4 つの引数があります。第1引数は、表示ツリーの目的の部分を見つけるために通過する必要があるアウトラインノードのリストです。第 2 引数と第 3 引数はペアで機能します。第 2 引数は表示要素の名前、第 3 引数は表示要素のタイプです。これら 2 つの引数で、コマンドの送り先である表示ツリーの特定の部分を指定します。送るコマンドは第 4 引数で指定します。

Dispatch コマンドの基本的な機能は、レポート内のアウトラインノード（第 1 引数）、そのアウトラインノードの下にある表示要素（第 2 引数と第 3 引数）、そしてコマンド（第 4 引数）を指定することです。

例として、サンプルのデータセット「Big Class」を開き、付属の「**二変量の関係 (二変量)**」のスクリプトを実行します。これにより、あてはめた直線のレポートが生成されます。次に、「あてはまりの悪さ（LOF）」アウトラインノードを開き、「分散分析」アウトラインノードを閉じます。最後に、［**スクリプト**］>［**スクリプトをスクリプトウィンドウに保存**］を選びます。次のスクリプトが表示されます （見やすいように、スペースと改行を追加しています）。

```
Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Line( {Line Color( {213, 72, 87} )} ),
   SendToReport(
      Dispatch( {"Linear Fit"}, "Lack Of Fit", OutlineBox, {Close( 0 )} ),
      Dispatch( {"Linear Fit"}, "Analysis of Variance", OutlineBox, {Close( 1 )} )
   );
);
```

Send To Report コマンドに 2 つの **Dispatch** コマンドが含まれています。これらのコマンドは、デフォルトのレポートに対する 2 つのカスタマイズに対応しています。最初の **Dispatch** コマンドについて詳しく説明します。

第 1 引数は、「直線のあてはめ」という名前のアウトラインノードを見つけるように指示しています。第 2 引数と第 3 引数は、「直線のあてはめ」アウトラインの下にある「あてはまりの悪さ（LOF）」（Lack Of Fit）という名前のアウトラインボックスを見つけるように指示しています。第 4 引数は、このアウトラインボックスに送るコマンドです。この例では、メッセージは `Close(0)`、つまりノードを開くコマンドです。

注：同じ名前のアウトラインノードが複数ある場合は、添え字が割り当てられます。たとえば、「二変量の関係」の分析に 2 つの 2 次のあてはめがある場合（同じタイトルになる）、2 番目のあてはめにコマンドを適用すると、重複するタイトルに添え字 [2] が追加された Dispatch コマンドが生成されます。

Send to Report と **Dispatch** コマンドをうまく使うには、まずマウスを使ってレポートを実行し、インタラクティブにカスタマイズします。次に、JMP が生成したスクリプトを見てみます。JMP 自身が、実は最高の JSL プログラマーなのです。

プラットフォーム JSL の例

以下に、JSL を使って、分析レポートを最初から最後まで作成するスクリプトを示します。まず、データテーブルを開きます。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
```

ここで、二変量プラットフォームを起動し、それをプラットフォームの参照「biv」に割り当てます。

```
biv=bivariate(y(:Name("体重(ポンド)")),x(:Name("身長(インチ)")));
  // プラットフォームへの参照
```

プラットフォーム自体でできることを調べるには、プラットフォームオブジェクトに Show Properties を使います。なお、以下の出力は環境設定で「英語でスクリプトを保存」をオンにしている場合のものです。この項目をオフにすると、メニューと共通のコマンドは日本語で表示されます。

```
show properties(biv);
   Show Points [ブール] [デフォルトはオン]
   Fit Mean [アクション](Fits a flat line at the mean.)
   Fit Line [アクション](Fits a regression line to the data.)
   Fit Polynomial [アクションの選択] {2,quadratic, 3,cubic, 4,quartic, 5, 6}
   Fit Special [アクション](Fitting with transformations on X or Y, or constraints
      on slope or intercept.)
   Fit Spline [アクションの選択] {1000000, 硬, 100000, 10000, 1000, 100, 10, 1, 0.1,
      0.01, 軟, Other…}(Fitting a flexible curve)
   Fit Each Value [アクション](Fits a line that goes through the mean Y value of each
      set of unique X values.)
   Fit Orthogonal [アクションの選択] {Univariate Variances, Prin Comp, Equal
      Variances, Fit X to Y, Specified Variance Ratio…}(Fitting where both X and Y
      have error.);
   Density Ellipse [アクションの選択] {0.99, 0.95, 0.90, 0.50, Other…}(The bivariate
      normal contour fitted with the correlation.)
   ...
```

上で示している結果は、一部抜粋したものです。

「ログ」ウィンドウの出力を見て、プラットフォームにメッセージを送る際の参考にしてください。(この種のスクリプトの詳細については、「プラットフォームのスクリプト」(347 ページ)の章を参照してください。)

```
biv<<Fit Spline(1000000)<<Fit Mean;
biv<<show points(0); // プロット点を表示しない
biv<<show points(1); // プロット点を再表示する
biv<<fit polynomial(4, 2); // 次数 4、色 2
biv<<fit polynomial(2,4);  // 次数 2、色 4
```

次に、ウィンドウを見やすいサイズにし、目的のグラフを表示するために先頭までスクロールします。

```
biv<<size window(500,700);
biv<<scroll window({0,0});
```

これから、レポートの処理を始めます。まず、参照を作成し、次に、その参照でできることを調べる必要があります。

```
rbiv=biv<<report; // レポートへの参照
show properties(rbiv);
```

ログには、レポートで使用できるメッセージのリストが表示されます（下に示している結果は、一部抜粋したものです）。

```
Close [ ブール ]
GetOpen [ アクション ] [ スクリプトの場合のみ ]
SetOpen [ ブール ] [ スクリプトの場合のみ ]
Horizontal [ ブール ]
GetHorizontal [ アクション ] [Scripting Only]
SetHorizontal [ ブール ] [ スクリプトの場合のみ ]
Open All Below [ アクション ]
Close All Below [ アクション ]
Open All Like This [ アクション ]
Close All Like This [ アクション ]
Close Where No Outlines [ アクション ]
Get Close Orientation [ アクション ] [ スクリプトの場合のみ ]
Outline Close Orientation [ 選択肢 ] {Auto, Horizontal, Vertical}
Set Title [ アクション ] [ スクリプトの場合のみ ]
Get Title [ アクション ] [ スクリプトの場合のみ ]
Get Menu Script [ アクション ] [ スクリプトの場合のみ ]
Set Menu Script [ アクション ] [ スクリプトの場合のみ ]
Set Menu Item State [ アクション ] [ スクリプトの場合のみ ]
Get Menu Item State [ アクション ] [ スクリプトの場合のみ ]
Get Submenu [ アクション ] [ スクリプトの場合のみ ]
Set Submenu [ アクション ] [ スクリプトの場合のみ ]
Set Scriptable Object [ アクション ] [ スクリプトの場合のみ ]
Get Scriptable Object [ アクション ] [ スクリプトの場合のみ ]
Append Item [ サブテーブル ]
  Add Text Item [ アクション ]
  Add Outline Item [ アクション ]
  Add Window Reference [ アクション ]
  Add File Reference [ アクション ]
  Add Directory of Files [ アクション ]
  Add All Open Files [ アクション ]
  Add URL Reference [ アクション ]
  Add Script Button [ アクション ]
...
```

アウトラインの「平均のあてはめ」ノードを開きます。

```
rbiv[" 平均のあてはめ "]<<close(0);
```

実際に、いくつかの結果を選択してみます（個別の結果を見るには、各行を単独で実行）。

```
rbiv[" あてはめの要約 "]<<select;
rbiv[" パラメータ推定値 "]<<select;
rbiv[" 分散分析 "]<<select;
// アウトラインツリーの下位層にある要素を選択
rbiv[" 多項式のあてはめ 次数 =2"," パラメータ ?", columnbox(" 推定値 ")]<<select;
rbiv<<deselect;
```

2 番目の分散分析項目を選択するには、次のようにします。

```
rbiv[" 多項式のあてはめ 次数 =2", " 分散分析 "]<<select;
```

ここで、4 次多項式のパラメータ推定値レポートにおいて、列の表示形式を変更してみましょう。

```
pe=rbiv[" 多項式のあてはめ 次数 =4"," パラメータ ?"];
ests=pe[columnbox(" 推定値 ")];
ests<<set format(12,6);
```

Set Format の第 1 引数では、全体の列幅を表示する文字数で指定します。第 2 引数では小数点以下の桁数を指定します。

図 11.4 レポートに変更を適用

パラメータ推定値

項	推定値	標準誤差	t値	p値(Prob>|t|)
切片	-17.60008	84.82175	-0.21	0.8368
身長(インチ)	1.9521428	1.346006	1.45	0.1559
(身長(インチ)-62.55)^2	-0.220179	0.29996	-0.73	0.4678
(身長(インチ)-62.55)^3	0.0703948	0.035633	1.98	0.0561
(身長(インチ)-62.55)^4	0.0073899	0.004231	1.75	0.0895

変更前

パラメータ推定値

項	推定値	標準誤差	t値	p値(Prob>|t|)
切片	-17.600085	84.82175	-0.21	0.8368
身長(インチ)	1.952143	1.346006	1.45	0.1559
(身長(インチ)-62.55)^2	-0.220179	0.29996	-0.73	0.4678
(身長(インチ)-62.55)^3	0.070395	0.035633	1.98	0.0561
(身長(インチ)-62.55)^4	0.007390	0.004231	1.75	0.0895

変更後
<<Set Format(12,6)

テーブルから単一の数値を取得することもできます。たとえば 3 次の項の推定値を取得するには、次のように記述します。

```
terms=pe[columnbox(" 項 ")];
for(i=1, i<10 & (terms<<get(i))!="( 身長 ( インチ )-62.55)^3", i++, 0);
estimate = ests<<get(i);
```

0.070394822744608

このスクリプトが、どのように動作するかを説明します。Forによる反復処理は、目的の項までの行数を数えるために使われています。Forの2番目の引数が条件であることを思い出してください。条件のテストが真である限り、ループは続きます。この場合は、「項の列が"(身長(インチ)-62.55)^3"でなく、10番目の行に到達していない」という条件がテストされるので、文字列が**一致した**時点でループは終了し、*i*の値は一致した行の番号になります。Forによる反復処理のあとに、*i*を、推定値列に対するGetの引数として用いています。

この方法を使うと、取得した結果を次のテストの引数として使用できます。プロセス制御の場合は、特定の結果を監視し、その結果が一定の範囲を超えたら携帯電話に電子メールメッセージを送るようにすることができます。

```
resume = Expr( /* 次の反復を行うためのスクリプト */ );
message="現在の推定値は " || char(estimate) ||" です。日時： " || mdyhms(today());
if(estimate<1, resume, //else:
   mail("john.doe@company.com", "推定値が限界を超えました", message));
```

また、ボックスから値を行列で取得し、その値を使って、次の計算を行ったり、データテーブルを作成したりすることができます。データテーブルを直接作成することもできます。

```
myMatrix=rbiv[tablebox(4)] << get as matrix;

myVector=rbiv[tablebox(4)][columnbox("平方和")] << get as matrix;
dt<<new column("平方和",values(myVector));

rbiv[tablebox(4)] << make data table("分散分析表");
```

ここで、軸のスケールを調整します。

```
rbiv[axisbox(1)]<<min(70)<<max(170); // Y軸の調整
rbiv[axisbox(2)]<<min(50)<<max(70);  // X軸の調整
```

続いて、レポートの一番上にあるグラフをコピーします。グラフが含まれているピクチャーボックスを選択する必要があります。グラフだけを選択すると、軸が欠落してしまいます。

```
rbiv[PictureBox(1)]<<Copy Picture;
```

ピクチャーは、GIF、PNG（データ損失がなく、GIFより圧縮度が高い）、JPEG、JPG（写真向き）、WMF（フォントを維持する）の各形式でも保存できます。

```
rbiv[PictureBox(1)]<<save picture("myGraph.png",png);
rbiv[PictureBox(1)]<<save picture("myGraph.jpeg",jpeg);
rbiv[PictureBox(1)]<<save picture("myGraph.jpg",jpg);
```

最後に、レポートのジャーナルを作成します。

```
rbiv<<journal window;
```

Set Function と Set Script

<<Set Script メッセージを使って、ディスプレイボックスのコントロール（Button Box、Combo Box、Radio Box など）がクリックされたときにスクリプトを実行させることができます。例:

```
New Window( "Set Script の例 ",
   ex = Button Box( " 押してください " )
);
ex << Set Script( Print( " 押されました " ) );
```

上のスクリプトは、ボタンボックスがクリックされたときに、次のテキストをログに出力します。

" 押されました "

または、<<Set Function メッセージを使うと、ディスップレイボックスのコントロールに、ある特定の関数を実行させることができます。この関数の最初の引数はそのディスプレイボックスを表します。例:

```
New Window( "Set Function の例 ",
   Button Box( " 押してください ",
      <<setFunction(
         Function(
            {this /* functions specified with Set Function get 'this' display box
   */
            },
            this << setButtonName( " ありがとう " )
         )
      )
   )
);
```

上のスクリプトは、「押してください」という名前のボタンボックスを作成します。このボタンがクリックされたとき、Set Function によって呼び出された関数がボタンの名前を「ありがとう」に変更します。

注：<<Set Script と <<Set Function は同時に使用できません。特定のディスプレイボックスを参照する場合は、<<Set Function を使用してください。

Window

Window は、ウィンドウのタイトル（*"title"*）でウィンドウを見つけ出し、そのウィンドウへの参照を戻します。たとえば、「分析結果」というタイトルのウィンドウを閉じるには、次のコードを実行します。

```
wdw=Window(" 分析結果 ");
wdw<<closeWindow();
```

- 引数なしの Window() は、JMP で開いているウィンドウすべてのリストを戻します。
- Window(n) は *n* 番目のウィンドウを戻します。
- Window("title") は、指定したタイトルのウィンドウを戻します。

注：*n*番目のウィンドウまたは指定したタイトルのウィンドウが存在しない場合、空のリストが戻されます。

ビルトインウィンドウにアクセスするには

ビルトインウィンドウをスクリプトで開くことができます。データテーブルの指定やプラットフォームの起動をユーザに指定させたい場合に便利です。JMP ウィンドウを呼び出すには、単に JSL の呼び出しコマンドの引数を省略するだけです。例をいくつか示します。

```
dt=Open();           // ［ファイル］>［開く］で表示されるウィンドウ
dt<<Summary();       // ［テーブル］>［要約］で表示されるウィンドウ
myFit=Fit Y by X(); // ［分析］>［二変量の関係］で表示されるウィンドウ
myChart=Chart();     // ［グラフ］>［チャート］で表示される起動ウィンドウ
```

Open() などのモーダルウィンドウの場合、それ以降のスクリプトは、ウィンドウが閉じてから実行されます。プラットフォームを起動するスクリプトで、列の指定が必要なのに指定されていない場合、列を指定するためのプラットフォーム起動ウィンドウが表示されますが、スクリプトの実行は直ちに継続されます。つまり、Open() などのモーダルダイアログボックスとは異なり、ユーザからの入力を**待ちません**。プラットフォームの起動の詳細については、「プラットフォームのスクリプト」（347 ページ）を参照してください。

選択ウィンドウの使用

Pick Directory コマンドを使って、ユーザにディレクトリの選択を促すことができます。このコマンドは、プラットフォーム固有のウィンドウを表示し、ユーザにフォルダを選択させます。Windows の場合、「フォルダの参照」ダイアログの最上部にオプションのプロンプト文字列が表示されます。

```
path = Pick Directory ("ディレクトリを選択してください。");
```

ユーザにファイルを選択させるには、Pick File() コマンドを使用します。

```
path = Pick File(
   <"prompt message">, <"initial directory"> <{filter list}>,
   <first filter>, <save flag>, <"default file">,
   <multiple>);
```

"prompt message" はウィンドウのタイトルとして使用されます。"initial directory" は、最初に表示するフォルダを指定します。ディレクトリを空の文字列として定義した場合は、デフォルトのディレクトリが使用されます。

Open() ウィンドウに使用する {filter list} を定義し、特定のタイプのファイルだけを表示することもできます。このリストには、次の構文を使用します。

```
{"Label1|suffix1;suffix2;suffix3", "Label2|suffix4;suffix5"}
```

例：

```
{"JMP Files|jmp;jsl;jrn", "All Files|*"}
```

各引用符付き文字列が、Open() ウィンドウの **File name** リストのエントリになります。Label は、各メニューオプションに表示されるテキストを定義します。次の接尾辞のリストは、対応するラベルが選択されている場合に表示されるファイルの種類を定義します。"*" を使用すると、すべての種類のファイルがウィンドウに表示されます。

```
Pick File(
   "JMP ファイルを選択する ",
   "$SAMPLE_DATA",
   {"JMP ファイル |jmp;jsl;jrn",
   " すべてのファイル |*"},
   1, 0, "newJmpFile.jmp");
```

ヒント：すべての引数はオプションですが、位置は固定です。つまり、省略が可能なのは、スクリプトの末尾のオプションのみとなります。それ以外のオプションを省略したい場合は、空の文字列を使用します。

下のスクリプトは、デフォルトのディレクトリもデフォルトのファイルも設定しません。

```
Pick File(
   "JMP ファイルを選択する ",
   "",
   {"JMP ファイル |jmp;jsl;jrn",
   " すべてのファイル |*"},
   1, 0, "");
```

<first filter> 引数には、デフォルトの選択肢を指定します。n はリスト項目のインデックスです。上のスクリプトでは、<first filter> はリストの最初の項目です。

<Save Flag> が偽の場合、ウィンドウは「ファイルを開く」（File Open）ウィンドウで、<Save Flag> が真の場合、ウィンドウは「ファイルの保存」（Save File）ウィンドウとなります。<Save Flag> が偽の場合、"Multiple" 引数は 1 つのウィンドウで複数のファイルを開きます。

```
Pick File(
   "JMP ファイルを選択する ",
   "",
   {"JMP ファイル |jmp;jsl;jrn",
   " すべてのファイル |*"},
   1, 0, "", "multiple");
```

注：コンピュータの物理メモリ内のバッファサイズは、ユーザが開くことのできるファイルの数に影響します。

ディレクトリ内のファイル

特定のディレクトリのファイル名リストを取得するには、Files In Directory コマンドを使います。

```
names = Files In Directory(path, <recursive>);
```

Macintoshの場合は、ファイル名のみが戻されます。Windowsの場合は、次の例のように、ファイル名とサブディレクトリ名の両方が戻されます。

```
names = Files In Directory("$SAMPLE_DATA");

   { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design Experiment",
      "Detergent.jmp", ... }
```

Design Experimentサブディレクトリ内のファイルは含まれず、ルートディレクトリの$SAMPLE_DATAにあるファイルしかリストされていません。ファイル名すべてのリストを戻すには、Files In Directoryにオプションの引数recursiveを追加します。

```
names = Files In Directory("$SAMPLE_DATA", recursive);

   { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design Experiment/2x3x4
      Factorial.jmp", "Design Experiment/Borehole Factors.jmp", ... }
```

完全なパス名を取得するには、ディレクトリ内を再帰的に参照し、ファイルパスとファイル名を連結します。次の例は、$SAMPLE_DATAディレクトリとサブディレクトリの中の各ファイルを繰り返し処理します。各ファイル名にファイルパスが連結されます。

```
names = Files In Directory("$SAMPLE_DATA", recursive);
   For( i = 1, i <= N Items(names), i++,
      names[i] = Convert File Path("$SAMPLE_DATA") || names[i] );
names;

   {"/C:/Program Files/SAS/JMP/11/Samples/Data/2D Gaussian Process Example.jmp",
   "/C:/Program Files/SAS/JMP/11/Samples/Data/Abrasion.jmp", ... }
```

Files in Directoryコマンドでは、ネイティブ形式のパス、POSIXパス、およびパス変数を使用できます。パスの操作の詳細については、「データタイプ」の章の「パス変数」（120ページ）を参照してください。

表示ツリーの作成

表示ツリーを構成する関数を使って、独自の表示項目を作成し、ウィンドウ内に配置できます。この節では、複数の表示項目を同じ画面に配置し、それらにメッセージを送る方法を説明し、最後にいくつかの例を示します。

基本事項

まず、New Windowにタイトルとウィンドウ内に表示する項目をリストする必要があります。ディスプレイボックスを構成する関数にはすべて、最後に「Box」が付いています。各種のディスプレイボックスについて調べるには、「ディスプレイボックスの紹介」（379ページ）を参照してください。Display Segs（セグメント）と呼ばれる、グラフフレーム内に存在する同様のオブジェクトがあり、それらの末尾には「Seg」が付いています。JMPにあるさまざまなDisplay BoxとDisplay Segのうち、一部はJSLでの作成が可能です。必要に応じて、ディスプレイボックスを入れ子にすることができます。

割り当てを使って、新しいウィンドウの参照を作ることにより、メッセージを送れるようになります。（このマニュアル全体をとおして、構文のまとめでは、*db*をディスプレイボックスの参照、*dt*をデータテーブルの参照、*obj*をスクリプトオブジェクトの参照の各プレースホルダとして使っています。）

表示オブジェクトは、JSLで作成または参照されると、別のディスプレイボックスにコピーされるか、ウィンドウが閉じて表示されなくなるまで、自由に共用できる参照です。表示オブジェクトを別の表示ツリーに入れると、JMPは、そのコピーを作成し、所有者を新しいボックスにします。

例

以下の例では、OutlineBox、H List Box、およびMatrix Boxを使って、行列用関数のクイックリファレンスを作成します。

```
a=[0 1 2,3 4 5,6 7 8]; b=[8 7 6,5 4 3,2 1 0];
c=identity(3); d=j(3,3,0); e=1::4; f=[+ -,- +];

mcs=New Window(" 行列用関数シート ",
   Text Box(" このスクリプトは、行列作成用関数のクイックリファレンスです。"),
      Outline Box(" 行列の開始 ",
      H List Box(
         Outline Box("a",Matrix Box(a)),
         Outline Box("b",Matrix Box(b)))),
   Outline Box(" 作成用関数 ",
      Outline Box("c=Identity(3)",Matrix Box(c)),
      Outline Box("d=J(3,3,0)",Matrix Box(d)),
      Outline Box("e=1::4",Matrix Box(e)),
      Outline Box("f=[+ -,- +]",Matrix Box(f)),
      Outline Box("Diag(a)", Matrix Box(diag(a))),
      Outline Box("VecDiag(a)", Matrix Box(vecdiag(a))),
      Outline Box("a||b", Matrix Box(a||b )),
      Outline Box("a|\b", Matrix Box(a|/b)),
      Outline Box("a`", Matrix Box(a`))));
```

図11.5 行列の例

ここで、*mcs*に保存されているディスプレイボックスの参照にメッセージを送り、表示形式を整えます。

```
for(i=1,i<12,i++,mcs[matrixbox(i)]<<set format(2,1));
```

図11.6 整頓された行列の例

項目をきれいに配置するには、Outline Box内にMatrix Boxを配置するのが便利だということがわかるでしょう。このパターンに従って、アウトラインの枝を増やし、トピックを追加することができます。

既存の表示の更新

作成されるレポートに表示するディスプレイボックスの数がわからない場合があります。たとえば、1つまたは複数の変数を分析してレポートする一般的なスクリプトを作成する場合などです。スクリプトを実行するたびに変数の数が変わる可能性があるので、必要なディスプレイボックスの数はわかりません。

以下の節では、Append、Prepend、Delete、およびSib Appendを使用して、レポートのディスプレイボックスを追加したり削除したりする方法を説明します。

Append

Appendメッセージを使うと、既存の表示の末尾にディスプレイボックスを追加できます。スクリプトで、空のボックスを1つ作成し、<<Appendを使って分析の各変数用にそのボックスを追加します。

次のコード例では、変数effectsList内に有効な名前のリストがあり、その名前が行列varprop内の列に対応していることを前提としています。つまり、effectsList[1] は varprop[0,1] のラベルに、effectsList[2]はvarprop[0,2]のラベルにというように順次対応します。

```
varprop=[0 1 2,3 4 5,6 7 8];
effectsList={"いち", "に", "さん"};
```

まず、H List Boxを含む空のOutline Box（アウトラインボックス）が作成されます。内部の空のコンテナにはhbという名前が付けられます。

```
New Window( "H List Boxの例",
   Outline Box( "分散比率",
      hb = H List Box()
   );
);
```

次に、effectsListの項目数分だけforループを繰り返し、effectsListの各要素に対応したNumber Col Box（数値列ボックス）を追加します。

```
for(i=1, i<=NItems(effectsList), i++,
 Eval(substitute(
   expr(hb << append(Number Col Box(effectslist[i], varprop[0,i]))),
   expr(i), i)
 );
);
```

Prepend

PrependメッセージはAppendと同様に動作しますが、項目をディスプレイボックスの最後にではなく先頭に追加します。ディスプレイボックスが追加不能なタイプの場合、追加コマンドは追加可能な子ディスプレイボックスに転送されます。追加コマンドは表示ツリーの最上部に適用するのが最もいい方法です。

たとえば、次のスクリプトは最上部にボタンボックスのある「二変量」レポートを作成します。

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate( Y( :Name("身長(インチ)") ), X( Name("体重(ポンド)") ), Fit Line
  );
(biv << report)(Outline Box( 1 )) <<
Prepend(
   Button Box( "曲線はここをクリック", biv << Fit Polynomial( 2 ) )
);
```

「**曲線はここをクリック**」ボタンをクリックすると、グラフに2次曲線が追加されます。

ツリーの最上部にこのボタンを追加しても、同じことが行えます。

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate( Y( :Name("身長(インチ)") ), X( Name("体重(ポンド)") ), Fit Line
  );
(biv << report) << Prepend(
   Button Box( "曲線はここをクリック", biv << Fit Polynomial( 2 ) )
);
```

Delete

Deleteメッセージは、指定されたディスプレイボックスとその子をすべてレポートから削除します。表示を完全に動的にする場合は、このメッセージとAppendおよびPrependメッセージを一緒に使うと便利です。次の例では、テキストボックスを別のテキストボックスに置き換えます。この場合、スクリプトはSet Textを使う方法もありますが、大半のディスプレイボックスは内容を変更できません。

```
x=New Window ("X",
   list = vListBox(
   t1 = Text Box("t1"),
   t2 = Text Box("t2")));

t1 << Delete;
list << Append(t1 = Text Box ("t1new"));
```

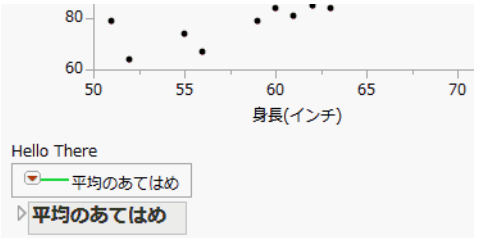
Sib Append

Sib Appendメッセージを使うと、既存のツリーに要素を追加できます。「ディスプレイツリーの表示」(382ページ)のディスプレイボックスツリーを参照してください。ListBox(2)の下に、2つのピクチャーボックスツリーがあります。PictureBox(1)には、二変量の関係の散布図が含まれています。これは、**身長**と**体重**に対応する2つの軸ボックスがあることから判断できます。Picture Box (2)には、緑の線と「**平均のあてはめ**」というテキストボックスが含まれており、これらは「平均のあてはめ」メニューに対応しています。

これらの2つのボックスの間にテキストボックスを挿入したいとします。Picture Box (1)と同じレベルにボックスを追加する必要があるので、Sib Appendメッセージを送ります。

```
Open("$SAMPLE_DATA¥Big Class.jmp");
biv=Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ), Fit Mean(
  {Line Color( {57, 177, 67} )} ) );
(biv<<report) [PictureBox(1)] <<SibAppend(Text Box("Hello There"));
```

図 11.7 同じレベルにテキストボックスを追加



数値列の更新

ディスプレイボックステーブルの数値列を更新するには、Set Values コマンドを使います。

```
Number Col Box<<Set Values ([matrix])
```

行列引数（matrix）は、テーブルの新しい数値を指定します。

文字列の列の更新

ディスプレイボックステーブルの文字列の列を更新するには、Set Values コマンドを使います。

```
String Col Box<<Set Values (<{list}>)
```

<{list}>引数には、テーブルの新しい文字列をリストで指定します。

テキスト改行の制御

通常、Text Box 内ではテキストは自動的に改行されます。ただし、Set Wrap (*n*) メッセージを使うと、デフォルトの改行ポイントを無効にすることができます。この *n* は、改行までのピクセル数です。

テキストボックス内の箇条書き

箇条書きにするには、Bullet Point(1) メッセージを使います。

```
win = New Window( " 箇条書きのリスト ",
   textOne = Text Box( "Lorem ipsum dolor sit amet, consectetur adipisicing elit,
   sed do eiusmod tempor incididunt ut labore et dolore magna aliqua." )
);
textOne << Bullet Point( 1 );
```

このメッセージをテキストボックスに送ると、テキストの前に中黒が挿入され、テキストボックス内の後続の行はインデントされます。

インタラクティブな表示要素

通常の JMP プラットフォームでは使われていない特殊な種類のディスプレイボックスもあります。Button Box、Slider Box、Global Box、Global Box の 4 つです。これらは、対話的なグラフを備えたウィンドウを、ユーザが独自に作成する場合に便利です。

注：「スクリプトによるグラフ作成」（461 ページ）の章では、Handle と MouseTrap が、グラフの内部でどのように動作するのかを説明しています。

ボタン、スライダ、および編集フィールドのコントロールをグラフの外側に配置することもできます。各ボックスタイプはそれぞれ独立したディスプレイボックスを作成するので、「表示ツリーの作成」（399 ページ）で説明しているルールに従い、新しいウィンドウの中でそれらを組み合わせる必要があります。基本的な効果は、ここで示した例のパターンで十分なので、詳しい説明は後にします。

Slider Box

Slider Box は、最小値（min）と最大値（max）で指定された範囲内で、変数の値を指定するスライダコントロールを描きます。スライダを移動するたびにスライダの現在の位置による値がグローバル変数に関連付けられ、それに伴ってグラフが更新されます。つまり、Slider Box を用いることにより、ユーザが指定した値をグラフに設定することができます。

```
Slider Box(min, max, global variable, script, <set width(n)>, <rescale slider(min,
  max)>);
```

以下はその例です。

```
ex = .5;
New Window( "スライダ",
   tb = Text Box( "Value: " || Char( ex ) ),
   sb = Slider Box( 0, 1, ex, tb << Set Text( "Value: " || Char( ex ) ) )
);
```

Range Slider Box

値の範囲を指定する 2 つのコントロールを含むスライダを描画するには、Range Slider Box を使用します。

```
Range Slider Box(min, max, low_val, high_val, script);
```

簡単な例を示します。

```
low = .5;
high = .75;
New Window( "範囲スライダ",
   tb1 = Text Box( "Low: " || Char( low ) ),
   tb2 = Text Box("High: " || Char(high)),
   sb = Range Slider Box( 0, 1, low, high,
      tb1 << Set Text( "Low: " || Char( low ));
      tb2<< Set Text( "High: " || Char( high ));
   )
);
```

Button Box

Button Box は、指定された名前付きのボタンを描きます。ボタンをクリックするたびに、スクリプトが実行されます。ボタンは常にアクティブで、ウィンドウが存在する間、使用できます。Button Box とスライダを組み合わせて使ったり、データ状態の変化に合わせてグラフを更新したりすることができます。また、click() コマンドをボタンオブジェクトにいつでも送ることができます。

```
Button Box("text",<script>);

New Window(" ボタン ",
   hello = Button Box(" こんにちは ", Print(“ こんにちは ")));
hello<<click();
```

Global Box

Global Box は、JSL グローバル変数の名前と現在の値を表示します。ユーザは、ウィンドウで値を直接編集し、Enter キーまたは Return キーを押して変更を確定することで、新しい値をグローバル変数に割り当てることができます。そのグローバルボックスを使うグラフは、自動的に新しい値に更新されます。Sqrt(4) などの式を指定すると、グローバルボックスは、まず、その式を評価してから、結果の 2 を保存して表示します。

```
ex = Sqrt(4);
New Window( " グローバル ", Global Box( ex ) );
```

上のスクリプトは、「グローバル」という名前の新しいウィンドウを作成し、次の結果を表示します。

ex=2

注：Global Box を使用すると、変数が変化するたびにウィンドウの更新が発生します。このため、Global Box オブジェクトの数によってはウィンドウの更新に時間がかかってしまう可能性があります。最終版のスクリプトでは、グローバルボックスの数が多くならないようにすることをお勧めします。グローバルボックスではなく、必要なときに手動で更新できるテキストボックスを使用する方法もあります。

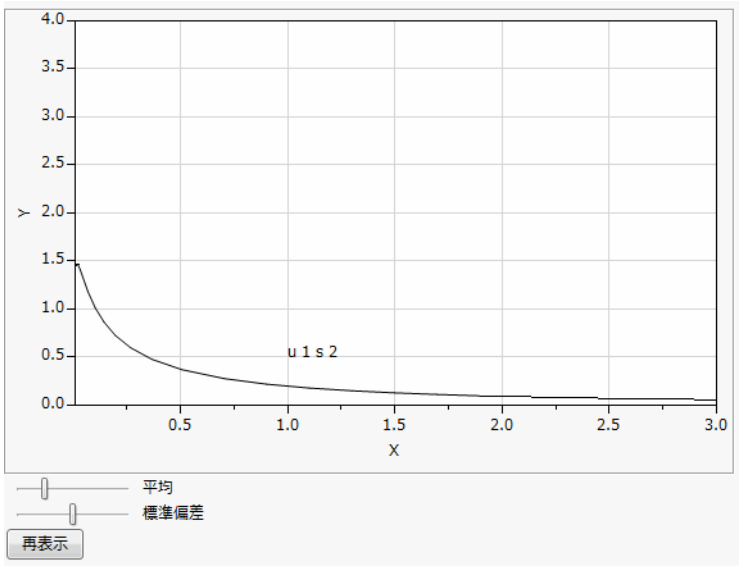
例

以下の例では、グラフボックス、2 つの横方向のボックス、1 つのボタンを縦方向のリストボックス内に貼り付けることによって、2 つのスライダと 1 つのボタンを伴ったグラフを作成しています。

```
// スライダ機能を使った対数正規分布
lU=1; lS=2;
New Window(" 対数正規密度 ",
   V List Box(
      gr=Graph Box(Frame Size(500,300),X Scale(0.01,3), Y Scale(0,4),
         Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
         YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
         text({1,.5},"u= ",lU," s= ",lS)),
      H List Box(Slider Box(0,4,lU,gr<<reshow),Text Box("Mu")),
      H List Box(Slider Box(0,4,lS,gr<<reshow),Text Box("Sigma")),
      Button Box(" 再表示 ",gr<<reshow)));
```

```
show(gr);
gr<<reshow;
```

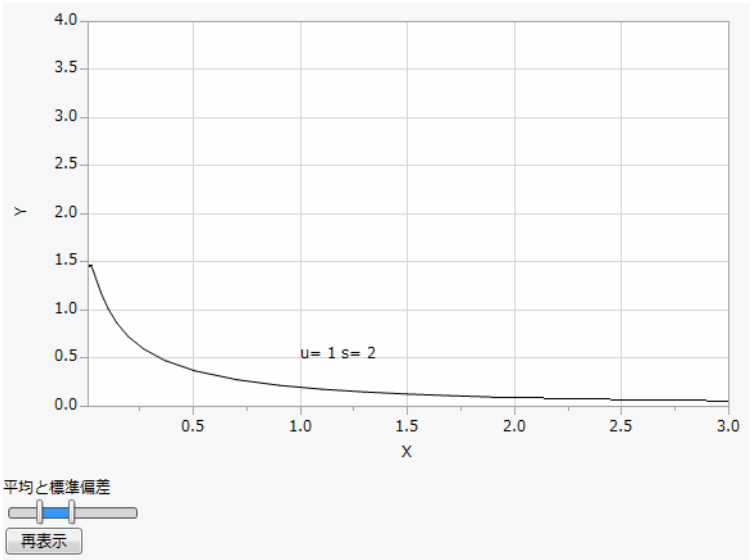
図11.8 レポートウィンドウ内のスライダとボタンの例



次のスクリプトは、Slider Box()の代わりにRange Slider Box()を使って、同じ機能を実行しています。

```
// 範囲スライダ機能を使った対数正規分布
lU=1; lS=2;
New Window(" 対数正規密度 ",
   V List Box(
      gr=Graph Box(Frame Size(500,300),X Scale(0.01,3), Y Scale(0,4),
         Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
         YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
         text({1,.5},"u= ",lU," s= ",lS)),
      V List Box(Text Box(" 平均と標準偏差 "),Range Slider Box(0,4,lU,
   lS,gr<<reshow))),
      Button Box(" 再表示 ",gr<<reshow));
show(gr);
gr<<reshow;
```

図 11.9 範囲スライダボックスの使用例

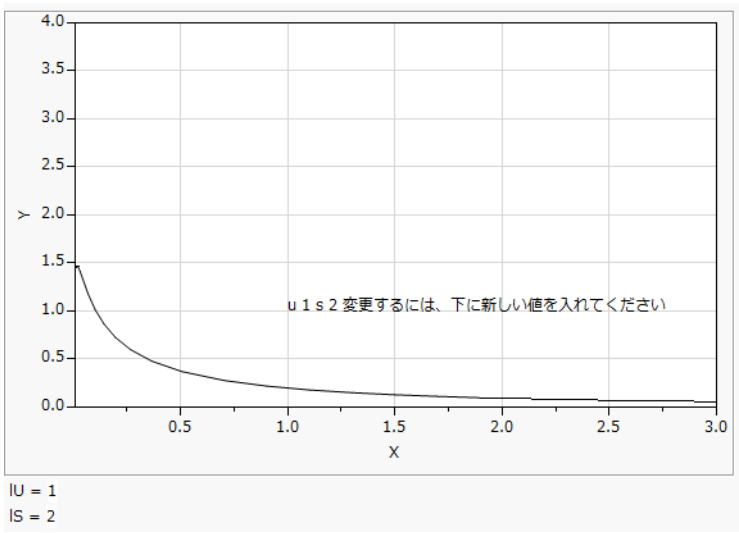


注：Range Slider Boxのコントロールの場合、高値の変数は低値のものより大きくなければいけません。また、低値の変数は高値の変数を超えてはなりません。

下のスクリプトは、Slider Boxのコントロールではなく、Global Boxを編集可能なテキストボックスとして使用しています。

```
// グローバルボックスを使った対数正規分布
lU=1; lS=2;
New Window(" 対数正規密度 ",
   V List Box(
      gr=Graph Box(Frame Size(500,300),X Scale(0.01,3), Y Scale(0,4),
         Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
         YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
         text({1,1},"u= ",lU," s= ",lS," 変更するには、下に新しい値を入れてください ")),
      H List Box(Global Box(lU)),
      H List Box(Global Box(lS))));
```

図11.10 スライダの代わりにグローバルボックスを使用した例



「スクリプトによるグラフ作成」の章の「Drag 関数」（496 ページ）の例で、Button Box の別の使い方を示しています。

モーダルウィンドウと非モーダルウィンドウ

JMP で使われているプラットフォーム起動ウィンドウのような非モーダルウィンドウは、レポートや表示リストと同じです。以下に示すすべての要素は、非モーダルのウィンドウにも組み込めます。

注：モーダルウィンドウを作成する Dialog() 関数は廃止され、今後のバージョンの JMP では使用できなくなる可能性があります。Dialog() 関数の代わりに、Modal 引数を指定した New Window() 関数を使用してください。

New Window() の使用方法については、「モーダルウィンドウ」（435 ページ）を参照してください。構文の詳細については、『スクリプト構文リファレンス』を参照してください。

New Window() では、次のコントロールが使用できます。

Border Box

```
Border Box (Left(pix), Right(pix), Top(pix), Bottom(pix), Sides(int), displaybox)
```

displaybox 引数の周りにスペースを追加します。Left（左）、Right（右）、Top（上）、および Bottom（下）で、*displaybox* 引数のどこにどれだけのスペースを追加するかを指定します。Sides は、表 11.2 にあるように、ボックスの周りに枠を描きます。Sides では、その他の効果も適用できます（表 11.3 を参照）。効果と枠の両方を追加するには、2 つの数値を足します。

たとえば、次のコードは、上下に枠のあるテキストボックスを作成し（枠の位置の値は5）、白の背景色を追加します（効果の値は32）。

```
New Window( "枠",
   Border Box( Sides( 37 ), Text Box( "Hello World!" ) )
);
```

表11.2 枠ボックスのSides引数の値

番号	枠の位置
0	なし
1	上
2	左
3	上と左
4	下
5	上と下
6	左と下
7	上、左、下
8	右
9	上と右
10	左と右
11	上、左、右
12	下と右
13	上、下、右
14	左、下、右
15	上、左、下、右

表11.3 枠ボックスのSides値の追加効果

上記の値に加算	追加の効果
16	枠を環境設定で定義した強調色にする。デフォルトは青です。
32	ボーダーボックスの背景を白にする。
64	ボーダーボックスのコンテナの背景を消去する。

Button Box

```
Button Box("text", <script>)
```

上のスクリプトは、*text*を表示したボタンを描きます。

ボタンにツールヒントを追加するには、次のようにSet Tip メッセージを送ります。例:

```
Button Box( "Hello", <<Set Tip( "World" ) );
```

上のスクリプトは、Helloという名前のボタンを作成し、ボタンの上にカーソルを置いたときにWorldというツールヒントが表示されるように設定します。

その他のButton Boxのメッセージには、<<Set Button Name("string")および<<Get Button Nameがあります。

また、次のアウトラインボックスを開くOpen Next Outlineメッセージをスクリプトコマンドとしても送ることができます。ボタンボックスに複数のメッセージを送る場合、このメッセージを最初のコマンドとしてリストします。

メニューを含んだボタンを作成することもできます。

```
New Window( " テスト ",
   bb = Button Box( " 文字を選択 ",
      choice = bb << Get Menu Choice;
      Show( choice );
   )
);
bb << Set Menu Items( {"a", "b", "-", "c"} );
```

この例の"-"は、メニューの区切りを作成します。区切り記号はリスト内の項目として数えられます。メニューからcを選んだ場合、3ではなく、4が戻されます。

Check Box

```
Check Box ({"item 1", "item 2", ...}, <script>)
```

チェックボックスの項目は、同時にいくつでも選択できます。

Col List Box

```
Col List Box (<Data Table (<name>), <All>, <width(n)>, <maxSelected(n)>,
   <nlines(n)>, <script>, <MaxItems(n)),<MinItems(n)>, <character | numeric>,
   <onChange(expression)>)
```

Allは、現在のデータテーブル内のすべての列を含めることを表します。Width（幅）の単位はピクセル数です。MaxSelectedは、リストボックスで選択できる項目の最大数を表します。Nlinesは、ボックスに表示する行数を表します。

Col List BoxにGet Itemsメッセージを送ると、選択されている列のリストを取得できます。Get Itemsを使用したスクリプト例を紹介します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Get Items のデモ ",
   H List Box(
      chooseme = Col List Box( All, width( 100 ), nlines( 6 ) ),
      Lineup Box(
         N Col( 1 ),
         Spacing( 3 ),
         Button Box( " 列の追加 >>",
            listocols << Append( chooseme << GetSelected );
            // Get Items を Col List Box に送る
            Chosen Columns = listocols << GetItems; ),
         Button Box( "<< 列の削除 ",
            listocols << Remove Selected;
            // Get Items を Col List Box に送る
            Chosen Columns = listocols << GetItems;
         ),
      ),
      // listocols は Col List Box
      listocols = Col List Box( width( 100 ), nlines( 6 ) ),
   ),
   Text Box( " " ),
   // Get Items が戻す項目を表示
   stuff = Global Box( Chosen Columns )
);
```

Col Span Box

Col Span Box を使用すると、テーブルボックス内で複数の列にまたがる列見出しを作成できます。最上部の列見出しは、2 つの子列ヘッダにまたがります。JMP での一例は、「信頼限界」の見出しで、これは「上側限界」と「下側限界」の 2 つの列にまたがります。

```
Col Span Box( title, children )
```

Combo Box

```
Combo Box({"item 1", "item 2", ...}, <script>)
```

コンボボックスの項目は、ドロップダウンメニューの中に表示されます。

Journal Box

Journal Box 関数は、Box で終わる他の関数と同様に、ウィンドウにディスプレイボックスを表示します。このディスプレイボックスは、他のディスプレイボックスと組み合わせて表示できます。

使い方は次のとおりです。

```
box = Journal Box("journal text")
```

ここで、*"journal text"*（ジャーナルテキスト）は、ジャーナルファイルから抽出されたテキストです。

ジャーナルテキストについては、他のボックスと組み合わせ可能なボックスについて細かいルールがあります。したがって、ジャーナルテキストを取得する場合は、領域を強調表示し、ジャーナルコマンドを使って目的の項目だけを含むジャーナルを作成して、保存することをお勧めします。その後で、保存したファイルをテキストエディタで開き（ファイルの拡張子を変更する必要がある場合があります）、Journal Box の引数としてスクリプトに貼り付けます。この際、"\[...]\" という引用符を使うことを強くお勧めします。このように指定すると、ジャーナルテキスト内で二重引用符を使用することができます。

また、ディスプレイボックスに <<GetJournal を送って、ジャーナルテキストを取得することもできます。

次に、モザイク図を作成する例を示します。

```
New Window(" モザイク図 ",
TextBox(" モザイク図があります "),
JournalBox("\[ // 引用符の指定の仕方に注目
PictureBox(sub(
BorderBox(top(12), left(5), bottom(5), right(7), sides(143), options(0), xmin(0),
  ymin(0), sub(
ScaleBox(ID(2), axis(scaleType(0), scaleOrig(0), scaleWidth(1), widthMajor(0.25),
nbin(4), nminor(0), timeCode(0), ndec(3), ndecSpec(0),
MinInit(0), MaxInit(1), LinearInit, MajorInit(0.25), MinorInit(0), NObsInit(0),
options(showMajorTicks, showMinorTicks, showLabels, fixMinimum, fixMaximum)),
  length(180), sub(
ScaleBox(ID(1), length(200), sub(
ListBox(horizontal, near, sub(
ListBox(horizontal, near, sub(
CenterBox(vert, sub(
TextEditBox(" 年齢 ", left))),
AxisBox(side(R), size(33, 180), locked(false), scales(0, 2, 0, 2), ))),
ListBox(vertical, near, sub(
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0), ymin(0),
  sub(
FrameBox(size(200, 180), border(0), flags(0), markerSize(-1), help name("Conting
  Mosaic"), scales(1, 2, 1, 2), seg(
MosaicSeg(
num x(2), num y(6), totals(18, 22),
cross tabs(0.277777, 0.4444444, 0.72222, 0.83333, 0.944444, 1, 0.136366, 0.31818,
  0.63636, 0.863636,
0.909090909090909, 1), sum weight(40), ycolors(5, -2142812212, -2138140980,
  -2134077810, -2134096057, 3), vertical )))))),
NomAxisBox(size(200, 36), sizeID(1, 0), num labels(2), labels(F, M), value(18,
  22), total(40), horizontal, left),
CenterBox(horiz, sub(
TextEditBox(" 性別 "))))),
NomAxisBox(size(29, 180), sizeID(0, 2), num labels(6), labels("12", "13", "14",
  "15", "16", "17"), value(8, 7, 12, 7, 3, 3), total(40), vertical, left),
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0), ymin(0),
  sub(
```

```
FrameBox(size(10, 180), border(0), flags(0), markerSize(-1), help name("Conting
  Mosaic Single"), scales(0, 2, 0, 2), seg(
MosaicSeg(
num x(1), num y(6), totals(40),
cross tabs(0.2, 0.375, 0.675, 0.85, 0.925, 1),
sum weight(40), ycolors(5, -2142812212, -2138140980, -2134077810, -2134096057, 3),
  vertical)
))))))))))))))
]\") //引用符を閉じる
)
```

Line Up Box

```
Line Up Box (NCol(nc), <Spacing(pixels)>, displaybox args)
```

引数で指定した *displaybox* が、*nc* に指定された列数に並べられて描かれます。オプションとして、列の間隔をピクセル単位で指定することができます。

List Box

```
List Box({"item 1", "item 2", ...}, <width(n)>, <max selected(n)>, <nlines(n)>,
  <script>)
```

Width（幅）の単位はピクセル数です。Max selected は、リストボックスで選択できる項目の最大数を表します。Nlines は、ボックスに表示する行数を表します。デフォルトは 3 です。

Append または Insert を使用して、項目をリストボックスに追加できます。

```
New Window( "テスト", lb = List Box( {"a", "e"} ) );
lb << append( {"f", "g"} ); // 結果は a、e、f、g
lb << Insert( {"b", "c", "a", "d"}, 1 ); // 結果は a、b、c、d、e、f、g
```

Append は、常に、リストボックスの最後にリストを追加します。Insert は、指定した位置の後ろにリストを追加します。

一方のボックスで選択された項目を他方のボックスで選択すると、最初のボックスの選択が解除されるといった、互いに排他的な 2 つのリストボックスを作成できます。選択を解除するには、Clear Selection を使用します。

```
New Window( "互いにクリアし合うボックス",
   window:la =List Box({"ブロッコリ", "ほうれん草", "ピーマン"},
      <<Set Script(window:lb << Clear Selection)
   ),
   window:lb =List Box({"アボカド", "かぼちゃ","トマト"},
      <<Set Script(window:la << Clear Selection)
   );
);
```

リストボックスには、イメージを含めることもできます。次の例では、ユーザのコンピュータにあるイメージが 1 番目のリスト項目内に表示されます。2 番目のリスト項目には、JMP の名義尺度アイコンが表示されます。

```
New Window( "例",
   List Box(
      {{"first", "c:\photo.gif"}, {"second","nominal"}},
      width( 200 ),
   );
);
```

Number Col Edit Box

```
Number Col Box("title", numbers)
```

指定されたタイトル（title）をつけた列を作成し、リストまたは行列の形で与えられた数値を挿入します。たとえば、次のように使用できます。

```
x = y = z = 0;
New Window( "例",
   <<Modal,
   Return Result, // 値を取り出す
   Table Box(
      neb =
      Number Col Edit Box(
         "values",
         {x, y, z}
      )
   );
);
   {neb = {2, 4, 6}, Button( 1 )} // 結果
```

Number Edit Box

```
Number Edit Box(value)
```

value引数で指定された数値が入力された数値ボックスを作成します。例:

```
New Window( "例",
   <<Modal,
   neb = Number Edit Box( 5 )
);
x = neb << get;
```

[OK] ボタンのみがウィンドウに表示されます。これはNew Window()のデフォルトの動作です。

Panel Box

```
Panel Box ("title", displaybox args)
```

*dispalybox*引数をラベルつきの枠で囲みます。

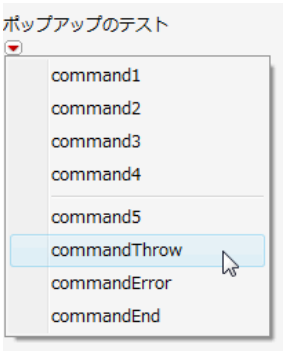
Popup Box

```
Popup Box({"command1", script1, "command2", script2, ...})
```

赤い三角形ボタンのメニューを作成する。次の例では、コマンドが含まれたリストを変数に格納し、Popup Box でこのリストを表示します。

```
commandList = {
"command1", print("command1"),
"command2", print("command2"),
"command3", print("command3"),
"command4", print("command4"),
"", empty(), //区切り線を挿入
"command5", print("command5"),
"commandThrow", throw("commandThrow1"),
"commandError", sqrt(1,2,3),
"commandEnd", print("commandEnd")};

New Window("ポップアップのテスト",
  Text Box("ポップアップのテスト"),
  Popup Box(commandList);
);
```

図 11.11 赤い三角ボタンのメニュー例



<<enable(boolean) を使用することによっても、メニューの有効／無効を切り替えることができます。引数が 1 の場合、メニューが有効になり、引数が 0 の場合は無効になります。先ほどの例を使用して、ポップアップボックスを変数に割り当て、その変数に enable メッセージを送ります。

```
New Window("ポップアップのテスト",
  Text Box("ポップアップのテスト"),
  mymenu = Popup Box(commandList);
);

mymenu << enable(0);
```

これで、赤い三角形は表示されますが、メニュー自体は無効となります。

Radio Box

```
Radio Box({“item 1”, “item 2”, ...}, <script>)
```

ラジオボックスの項目は、一度に1つしか選択できません。

Slider Box

```
Slider Box(min, max, global variable, script, <set width(n)>, <rescale slider(min, max)>)
```

最小値（min）と最大値（max）で指定された範囲内で、変数の値を指定するスライダコントロールを描きます。スライダを移動するたびにスライダの現在の位置による値がグローバル変数に関連付けられ、それに伴ってグラフが更新されます。つまり、Slider Boxを用いることにより、ユーザが指定した値をグラフに設定することができます。

```
ex = .5;
New Window( "例",
   tb = Text Box( "Value: " || Char( ex ) ),
   sb = Slider Box( 0, 1, ex, tb << Set Text( "Value: " || Char( ex ) ) )
);
sb << Set Width( 100 ) << Rescale Slider( 0, .8 );
```

複数のスライダボックスを作成する

1つひとつに固有のグローバル変数に値を格納することなく複数のスライダボックスを作成するには、次のようにします。

```
// 最小値と最大値だけを指定してスライダボックスを作成する
sb1 = Slider Box( 1, 10 );
sb2 = Slider Box( -10, 10 );
// スクリプトまたは関数を設定する
sb1 << Set Function( Function({this},{}, Show(this << get, sb2 << get ) ) );
sb2 << Set Script( Show(sb2 << get, sb1 << get) );
// スライダボックスをウィンドウに入れる
New Window( "title", sb1, sb2 );
```

String Col Edit Box

```
String Col Edit Box("title", {strings})
```

リストされた文字列（{strings}）を含んだ列を、指定された名前（title）とともに表の中に作成します。この関数によって作成された列の文字列は、編集できます。例:

```
a = b = c = "";
New Window( "例",
   <<Modal,
   Return Result,
   Table Box(
      seb =
      String Col Edit Box(
```

```
            "names",
            {a, b, c}
            )
         )
);
   {seb = {"do", "re", "me"}, Button( 1 )}
```

Tab Box

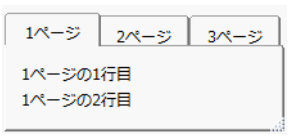
```
Tab Box("page title 1", contents of page 1, "page title 2", contents of page 2,
   ...)
```

タブ付きのウィンドウペインを作成します。

次の例に示すように、ボックス内にタブを表示することもできます。

```
New Window(" タブのテスト ",
   tb = TabBox("1 ページ ",
      vlistBox(
         textBox("1 ページの 1 行目 "),
         textBox("1 ページの 2 行目 ")
      ),
   "2 ページ ",
      vlistBox(
         textBox("2 ページの 1 行目 "),
         textBox("2 ページの 2 行目 ")
      ),
   "3 ページ ",
      vlistBox(
         textBox("3 ページの 1 行目 "),
         textBox("3 ページの 2 行目 ")
      );
));
```

図 11.12 タブボックス



あらかじめタブを選択しておくには、<<SetSelected(*n*) を送ります。*n* はタブの番号です。

<<Set Style メッセージにより、タブボックスの外観を選択できます。デフォルト値は tab（タブ）です。その他のオプションには、次のようなものがあります。

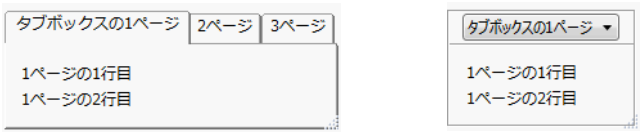
- combo はコンボボックスを作成します。
- outline はアウトラインノードを作成します。

- `vertical spread`はタブのタイトルを縦に表示します。
- `horizontal spread`はタブのタイトルを横に表示します。
- `minimize size`はタブのスタイルがタイトルの幅に応じて決まります。次のスクリプトは、1番目のタブのタイトルを長めに設定します。

```
New Window(" タブのテスト ",
   tb = TabBox(" タブボックスの１ページ ",
      vlistBox(
         textBox("1 ページの１行目 "),
         textBox("1 ページの２行目 ")
      ),
   "2 ページ ",
      vlistBox(
         textBox("2 ページの１行目 "),
         textBox("2 ページの２行目 ")
      ),
   "3 ページ ",
      vlistBox(
         textBox("3 ページの１行目 "),
         textBox("3 ページの２行目 ")
      );
));
tb << Set Style( minimize size );
```

タブボックスのサイズを最小化すると、タブボックスがコンボボックスに変換されます。図11.13は、デフォルトのタブボックスと最小化したタブボックスです。

図11.13 デフォルトのタブボックス（左）と最小化したタブボックス（右）



ヒント：`Set Style()` は、そのままの語と引用符で囲んだ語の両方に対応します。ただし、変数にスタイルを割り当て、それを引数として渡すことはできません。

Text Box

```
Text Box("text")
```

編集不可能なテキストボックスを描きます。他のコントロールのラベルとして使用されるのが一般的です。

テキストには、HTML タグで書式を設定することができます。たとえば、次のスクリプトは、テキストを太字にします。

```
w = New Window(" 書式付きテキスト ",
   Text Box(" これは <b> 太字 </b> のテキストです。",
   <<Markup) );
```

入れ子になったタグは、次のように正しく閉じることが重要です。

```
" これは <b><i><u> 太字斜体 </u></i></b> のテキストです "
```

注：回転後のテキストボックスは、Markup テキストに対応しますが、複数の行や複数の書式、行端揃えがある大きなテキストボックスはサポートしません。

Text Edit Box

```
Text Edit Box ("text")
```

編集可能なテキストボックスを描きます。

スクリプトを追加するには、Text Edit Box にスクリプトメッセージを送ります。これは、ボックスの作成時に実行するのが最も簡単です。スクリプトメッセージを最後の引数として追加するだけです。

たとえば次のスクリプトは、テキスト編集ボックスが変更されるたびにログにメッセージを出力します。

```
New Window(" テキスト編集ボックス ",
TextEditBox(" ここを変更してください ", <<Script(Print(" 変更されました ")))
);
```

Text Edit Box オブジェクトに参照を割り当てると、その内容が使用できるようになります。次のスクリプトは、テキスト編集ボックスが変更されるたびにその値をログに出力します。

```
New Window(" テキスト編集ボックス ",
teb=TextEditBox(" ここを変更してください ", <<Script(Print(teb<<Get Text)))
);
```

プレースホルダテキスト

テキスト編集ボックスが空のとき、そこにどのような値を入力すればよいかを示すヒントを、プレースホルダテキストとして挿入することができます。プレースホルダテキストは、ヒントとして表示されるだけで、テキストフィールドの値には影響しません。

次のスクリプトは、図 11.14 のテキスト編集ボックスを作成します。Hint() 関数は、"mm/dd/yyyy" を戻し、淡いグレーで表示します。

```
New Window(" 例 ",
   TextEditBox(" 現在の日付 "),
   TextEditBox("", Hint("mm/dd/yyyy"))
);
```

図11.14 プレースホルダテキストが表示されたText Edit Box

パスワード

パスワード入力用のテキスト編集ボックスを作成する場合は、入力文字をアスタリスクに置き換える形式を適用できます。たとえば、次の行は、文字列の値は「a」がアスタリスクとして表示されるテキスト編集ボックスを作成します。

```
q = Text Edit Box( "a", passwordstyle( 1 ), setscript( Print( "変更されました!" ) )
  )
```

ユーザがこのテキスト編集ボックスに新しい文字列をタイプすると、すべての文字がアスタリスクとして表示され、「変更されました!」というメッセージがログに印刷されます。

テキスト編集ボックスに、パスワード形式を使用するか使用しないかを指定するメッセージを送ることもできます。

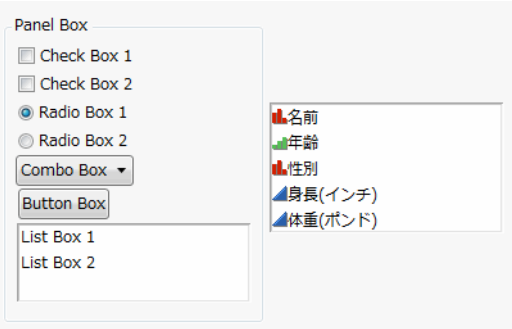
```
q << passwordstyle( 1 ) // テキスト編集ボックスをパスワード形式に設定する
q << passwordstyle( 0 ) // テキスト編集ボックスを標準形式に設定する
```

包括的な例

次のスクリプトを実行すると、上記のコントロールのほとんどを含むサンプルが作成されます。「Big Class.jmp」データテーブルを開きます。

```
New Window(" ウィンドウのコントロール ",
   Line Up Box (NCol(2), Spacing(3),
      Panel Box("Panel Box",
         Check Box({"Check Box 1", "Check Box 2"}),
         Radio Box({"Radio Box 1", "Radio Box 2"}),
         Combo Box({"Combo Box"}),
         Button Box("Button Box"),
         List Box({"List Box 1", "List Box 2"})
      ),
   Col List Box(all)
   );
);
```

図11.15 インタラクティブな表示要素の例



インタラクティブな表示要素の取得と設定の例

<<Set Selected (Item Number, <State>, <Run Script(0|1)>) コマンドを使うと、項目を最初から選択された状態にすることができます。このコマンドは、保存されているディスプレイボックス参照に対して単独で使用するか、リストボックスの<<引数として使用できます。

選択された値を取得するには、<<Get Selectedを使用します。これは、選択された項目の値を戻します。<< Get Selected Indicesは、選択された項目のインデックス番号を戻します。

```
antennaList = {" パラボラ "," ヘリカル "," 極性 "," ラジアントアレイ "};

// 方法 1: ディスプレイボックスの参照
New Window(" リストのテスト ",
 listObj = List Box (antennaList, Print("iList", listObj<<Get Selected,
  listObj<<Get Selected Indices))
);
listObj<<Set Selected(2, 1);

// 方法 2: インライン
New Window(" リストのテスト ",
 listObj = List Box (antennaList,<<Set Selected(2, 1), print ("iList",
  listObj<<Get Selected, listObj<<Get Selected Indices))
);
```

この2つのスクリプトは、どちらも次のテキストをログに出力します。

```
"iList"
{"Helical"}
{2}
```

上記の例では、<<Set Selectedメッセージが完了すると、Print式が実行されます。スクリプトが実行されないようにするには、最後の引数としてRun Script(0)を含めます。Run Script(0|1)は、ディスプレイボックスの変化に反応するスクリプトを<<Setまたは<<Set Selectedメッセージの後で実行するかどうかを制御します。

```
antennaList = {" パラボラ "," ヘリカル "," 極性 "," ラジアントアレイ "};
New Window( " リストのテスト ",
   listObj = List Box( antennaList,
      print( "iList",
      listObj << Get Selected, listObj<<Get Selected Indices ) )
      );
listObj << Set Selected( 2, 1, Run Script( 0 ) );
```

Run Script(1) を指定すると、Set メッセージの完了後にスクリプトが実行されます。値に変化がない場合でも実行されます。(ユーザが同じ値を選択した場合は、スクリプトは実行されません。) Run Script(0) を指定すると、スクリプトは実行されません。

インタラクティブなディスプレイボックスのほとんどでは、Run Script() を使用しない場合、スクリプトは実行されません。ただし、List Box() では、以前の動作と同様、デフォルトでスクリプトが実行されます。

応用例

次に示すコードは、「クラスター分析」プラットフォームの簡易版ディスプレイボックスを描き、指定の引数を使ってプラットフォームを起動します。

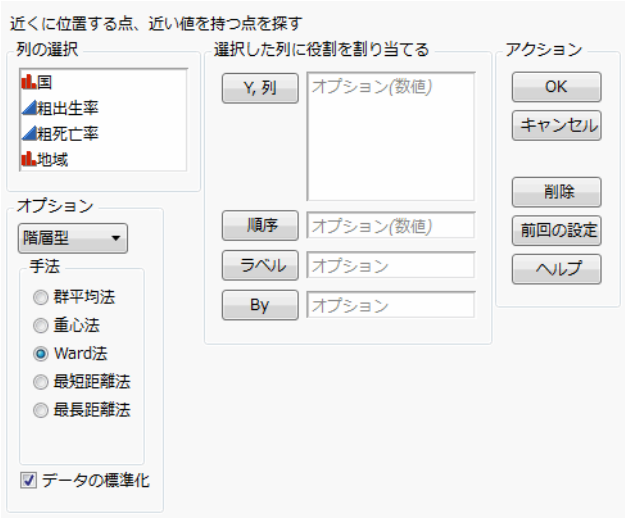
注：機能の一部（Recall=前回の設定と Help=ヘルプ）はスクリプトから実行できないので、クリックすると警告のウィンドウが表示されます。さらに、階層的クラスター法から K-Means クラスター法に切り換えても、ユーザインターフェースで表示されるときと違い、ウィンドウ自体は変化しません。

```
// 「クラスター分析」プラットフォームの起動ウィンドウ
dt = Open( "$SAMPLE_DATA/Birth Death.JMP" );
nc = ncol(dt);
lbWidth = 130;
methodList = {" 群平均法 "," 重心法 ","Ward 法 "," 最短距離法 "," 最長距離法 "};
notImplemented = expr(New Window(" この機能はまだ実行できません ",<<Modal,
  ButtonBox("OK")));

clusterDlg = New Window(" クラスター法 ",<<Modal,
   BorderBox(left(3),top(2),
   VListBox(
      TextBox(" 近くに位置する点、近い値を持つ点を探す "),
      HlistBox(
      VListBox(
         PanelBox(" 列の選択 ",
            colListData=ColListBox(All,width(lbWidth),nLines(min(nc,10)))),
            PanelBox(" オプション ",VListBox(
               comboObj=comboBox({" 階層型 ","KMeans 法 "},<<Set(1)),
               PanelBox(" 手法 ",
                  methodObj=RadioBox(methodList,<<Set(3))
            ),
            checkObj=check box({" データの標準化 "},<<Set(1,1))
         );
```

```
      );
   ),
   PanelBox(" 選択した列に役割を割り当てる ",
      LineupBox(NCol(2),Spacing(3),
         ButtonBox("Y, 列 ",
            colListY<<Append(colListData<<GetSelected)),
            colListY = ColListBox(width(lbWidth),nLines(5),numeric),
         ButtonBox(" 順序 ",
            colListO<<Append(colListData<<GetSelected)),
            colListO = ColListBox(width(lbWidth),nLines(1),numeric),
         ButtonBox(" ラベル ",
            colListL<<Append(colListData<<GetSelected)),
            colListL = ColListBox(width(lbWidth),nLines(1)),
         ButtonBox("By",
            colListB<<Append(colListData<<GetSelected)),
            colListB = ColListBox(width(lbWidth),nLines(1))
         );
      ),
      PanelBox(" アクション ",
         LineupBox(NCol(1),
            ButtonBox("OK",
               if ((comboObj<<Get)==1,
                  HierarchicalCluster(
                     Y(Eval(colListY<<GetItems)),
                     Order(Eval(colListO<<GetItems)),
                     Label(Eval(colListL<<GetItems)),
                     By(Eval(colListB<<GetItems)),
                     Method(methodList[methodObj<<Get]),
                     Standardize(checkObj<<get(1))),
                  KMeansCluster(
                     Y(colListY<<GetItems)
                  );
               );
               clusterDlg<<CloseWindow
            ),
            ButtonBox(" キャンセル ",
               clusterDlg<<CloseWindow),
               textbox(" "),
               ButtonBox(" 削除 ",
                  colListY<<RemoveSelected;
                  colListO<<RemoveSelected;
                  colListL<<RemoveSelected;
                  colListB<<RemoveSelected;
               ),
               ButtonBox(" 前回の設定 ",notImplemented),
               ButtonBox(" ヘルプ ",notImplemented))
```

```
            );
         );
      );
   );
);
```

図11.16「クラスター分析」起動ウィンドウ



作成した表示にメッセージを送る

作成したウィンドウに名前を割り当てると、その名前は、そのディスプレイボックスを所有するウィンドウへの参照になります。その後で、添え字を使って、そのウィンドウ内のディスプレイボックスにメッセージを送ることができます。

たとえば、グラフ部分にインタラクティブな正弦波の作成方法を示します。次の例では、メッセージをウィンドウ内のフレームボックスに送って正弦波を変化させています（*tf*への割り当てに注意）。

```
amplitude = 1; freq = 1; phase = 0;
t = New Window( "揺れ動く波",
   Graph Box(FrameSize(500,300),X Scale(-5,5),Y Scale(-5,5),Double Buffer,
      Y Function(amplitude*Sine(x/freq+phase),x);
      Handle(phase,amplitude,phase=x;amplitude = y);
      Handle(freq,.5,freq=x);
      Text({3, 4},"振幅：",Round(amplitude,4),
         {3, 3.5},"周波数：",Round(freq,4),
         {3, 3},"位相：",Round(phase,4))));
tf = t[framebox( 1 )];
For(amplitude=-4,amplitude<4,amplitude+=.1,tf << reshow);
```

もっと複雑な動きにしたい場合の For ループ使用例

```
amplitude = 1; freq = 1; phase = 0;
for(i=0,i<1000,i++,
   amplitude+=(Random Uniform()-.5);
   amplitude = if(amplitude>4,4,amplitude<-4,-4,amplitude);
   freq += (random uniform()-.5)/20;
   phase+=(Random Uniform()-.5)/10;tf<<reshow; Wait(0);
);
```

独自の表示を始めから作成する

以下のスクリプトは、Summarize 演算子を使って、「Big Class.jmp」の「**身長 (インチ)**」列に関する要約統計量を収集し、ディスプレイボックスを構成する関数を使って見やすく整理した結果をウィンドウに表示します。

```
dt=Open("$SAMPLE_DATA¥Big Class.jmp");
summarize( a=by(:年齢), c=count,
   sumHt=sum(:Name("身長 (インチ)")), meanHt=mean(:Name("身長 (インチ)")),
   minHt=min(:Name("身長 (インチ)")), maxHt=max(:Name("身長 (インチ)")));
sr=New Window("結果の要約",
   TableBox(
      stringColBox("年齢",a),
      NumberColBox("N",c),
      NumberColBox("合計",sumHt),
      NumberColBox("平均",meanHt),
      NumberColBox("最小値",minHt),
      NumberColBox("最大値",maxHt)));
```

これにより、図 11.17 に示す表を含む「結果の要約」というウィンドウが生成されます。

図 11.17 カスタマイズした要約レポートの生成

年齢	N	合計	平均	最小値	最大値
12	8	465	58.125	51	66
13	7	422	60.2857	56	65
14	12	770	64.1667	61	69
15	7	452	64.5714	62	67
16	3	193	64.3333	60	68
17	3	200	66.6667	62	70

ディスプレイボックスに、通常のコマンドを使用できます。

```
show properties(sr);
sr<<journal;
```

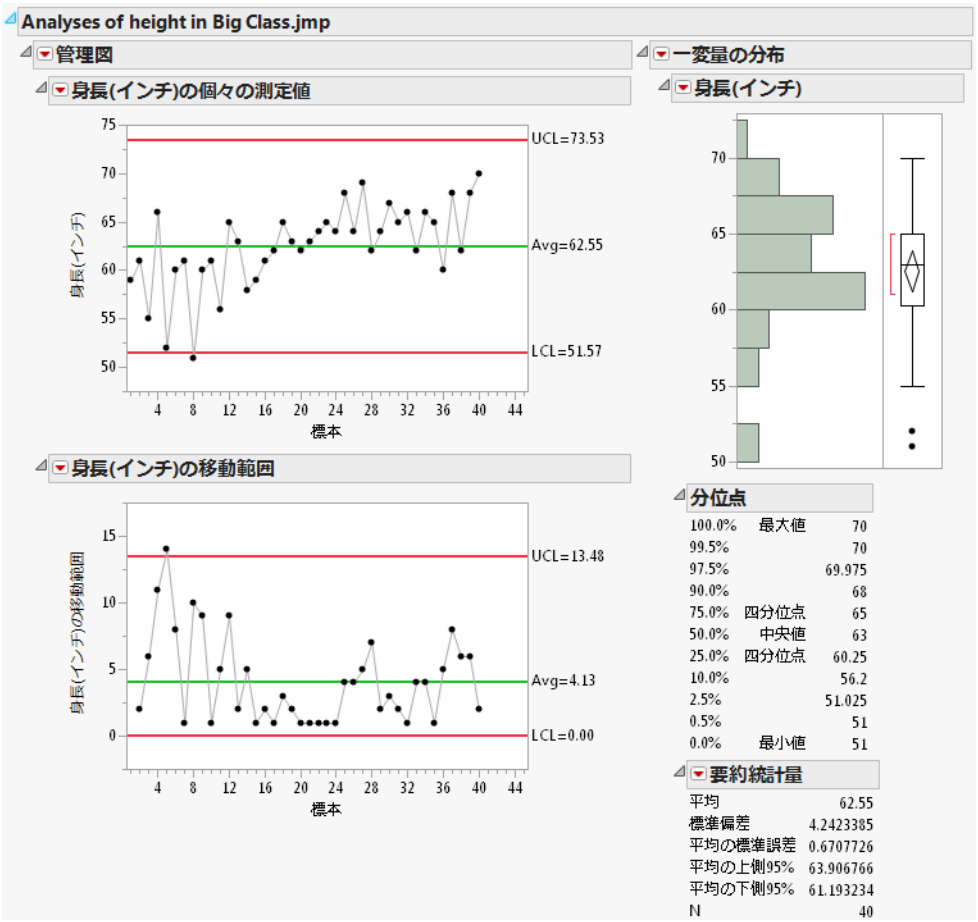
プラットフォームを含むディスプレイボックスの作成

単純にJMPの分析プラットフォームの結果を組み合わせたレポートを作りたいこともあるでしょう。ディスプレイボックス内にプラットフォームのスクリプトを入れるだけで、レポートウィンドウにプラットフォームの結果の入ったディスプレイボックスを組み込めます。さらに、後でメッセージを簡単に送れるように、全体を1つの参照に割り当てます。

この例では、複数のグラフとレポートを1つのウィンドウ内に作成します。

```
dt = Open("$SAMPLE_DATA¥Big Class.jmp");
csp=New Window("Big Class.jmp の身長の分析 ",
   OutlineBox("Big Class.jmp の身長の分析 ",
      HListBox(
         VListBox(cc=Control Chart(chart Col(Name(" 身長 ( インチ )"), Individual
            Measurement,Moving Range),K Sigma(3))),
         VListBox(dist=Distribution(columns(Name(" 身長 ( インチ )")))))));
```

図 11.18 例: 1 つのレポートウィンドウ内に表示された複数のグラフ



先ほどのスクリプトを実行した後、参照 *csp* に対してメッセージを送ると、ウィンドウを処理できます。この例における csp はディスプレイボックスの参照で、プラットフォームに対する Report の機能に似ています。csp に対して複数の添え字を使うと、アウトラインツリー内の特定の項目を見つけることができます。

```
csp[" 管理 ?", "? 移動範囲 "]<<close;
csp["? 分布 "," 分位点 "]<<close;
```

先ほどの例では、ウィンドウ全体を参照（csp）に割り当てるだけでなく、プラットフォーム起動スクリプトをディスプレイボックス内の名前（cc および dist）に割り当てています。これらの参照を使えば、プラットフォームへメッセージを簡単に送れます。ディスプレイボックスを操作する前述の例とは別の方法として、プラットフォームからレポートを取得する方法もあります。次のスクリプトでは、プラットフォームからレポートを取得して、ノードを再度開きます。

```
rcc=cc<<report; rdist=dist<<report;
rcc["? 移動範囲 "]<<close;
rdist[" 分位点 "]<<close;
```

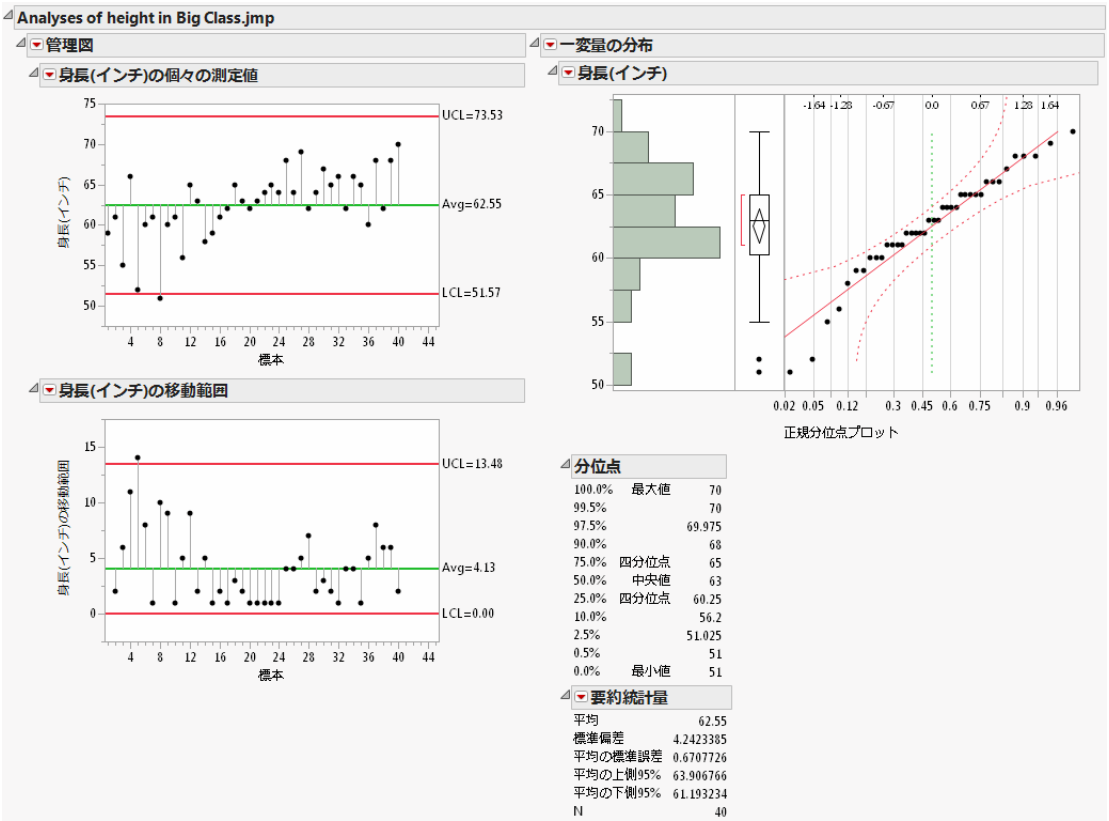
もちろん、メッセージを直接プラットフォームの参照に送ることができます。最初に、show propertiesを使って使用できるオプションを確認してください。

```
show properties(cc);
show properties(dist);
```

これにより、各プラットフォームのポップアップメニューに対応するコマンドがログに表示されます。コマンドをJSLから実行するには、そのコマンドをメッセージとしてプラットフォームの参照に送ります。

```
cc<<needle; dist<<normal quantile plot;
```

図11.19 カスタムレポートの変更



カスタムプラットフォームの作成

「プログラミング手法」の章の「式の操作」（215ページ）の例では、JSLのSubstituteInto演算子を使って、2次式の係数にある値を与えた後、その指定された2次式の解を求める方法を示しています。その例では、2次式の係数をSubstituteIntoの引数として指定しています。

「モーダルウィンドウ」（435ページ）の節では、モーダルウィンドウを介してユーザに係数を入力させる例を示しています。

この節では、これらの例をさらに改善し、完全にカスタマイズされたインターフェースを作成します。最初にウィンドウを表示して係数を入力させます。そして、根を計算した後、独自に作成したグラフとともに結果を表示します。

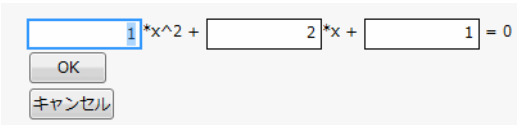
```
// まず、ウィンドウを表示して、ユーザに係数を入力させる
myCoeffs = New Window( "2 次式の根を見つける ",
   <<Modal,
   H List Box(
      a = Number Edit Box( 1 ),
      Text Box( "*x^2 + " ),
      b = Number Edit Box( 2 ),
      Text Box( "*x + " ),
      c = Number Edit Box( 1 ),
      Text Box( " = 0" )
   ),
   Button Box( "OK",
      a = a << get;
      b = b << get;
      c = c << get;
      Show( a, b, c );
   ),
   Button Box( " キャンセル " )
);


/* 次に結果を算出する : 2 次式は x=(-b + - sqrt(b^2 - 4ac))/2a。係数を 2 次式に入れる : */
x = {Expr( (-b + Sqrt( b ^ 2 - 4 * a * c )) / (2 * a) ), Expr( (-b - Sqrt( b ^ 2 -
   4 * a * c )) / (2 * a) )};
// 解のリストを保存する
xx = Eval Expr( x );

/* 3 番目に、実根が見つかったかどうかテストし、適切な表示をする。yes の場合（たとえば、ウィンドウのデフォルトで）は、根とグラフを表示する : */
results = Expr(
   xmin = xx[1] - 5;
   xmax = xx[2] + 5;
   ymin = -20;
   ymax = 20;
   myResult = New Window( "2 次関数の根 ",
      V List Box(
         Text Box( " 方程式の実根 " ),
         Text Box( "    " || Expr( po ) || " = 0" ),
         H List Box( Text Box( "are x=" ), Text Box( xxx ) ),
         Text Box( " " ),                    // 空白行を取得するため
         Graph Box(
            framesize( 200, 200 ),
```

```
            X Scale( xmin, xmax ),
            Y Scale( ymin, ymax ),
            Line Style( 2 ),
            H Line( 0 ),
            Line Style( 0 ),
            Y Function( polynomial, x ),
            Line Style( 3 ),
            Pen Color( 3 ),
            V Line( xx[1] ),
            V Line( xx[2] ),
            Marker Size( 2 ),
            Marker( 0, {xx[1], 0}, {xx[2], 0} )
         );
      );
   );
);
/* no の場合（たとえば、a=3、b=4、c=5 で）は、エラーウィンドウと理解を手助けするグラフを表示する : */
error = Expr(
   New Window( " エラー ",
      V List Box(
         Text Box( " " ),
         Text Box( " 多項式 " || po || " には実根がない。" ),
         Text Box( " " ),
         Text Box( " 原因を知るために、関数のグラフを表示 " ),
         Graph Box(
            framesize( 200, 200 ),
            X Scale( -20, 20 ),
            Y Scale( -20, 20 ),
            Line Style( 2 ),
            H Line( 0 ),
            Line Style( 0 ),
            Y Function( polynomial, x ),
         );
      );
   );
);
/* どちらの場合も、スクリプトは準備のための文字列を必要とする。指定された係数で多項式を書き直す : */
polynomial = Expr( a * x ^ 2 + b * x + c );
// 多項式のこのインスタンスを文字列で保存
po = Char( Eval Expr( polynomial ) );
// 解のリストを文字列で保存
xxx = Char( Eval Expr( x ) );
// テスト準備完了
If( Is Missing( xx[1] ) | Is Missing( xx[2] ),
```

```
    error,
    results
);
```

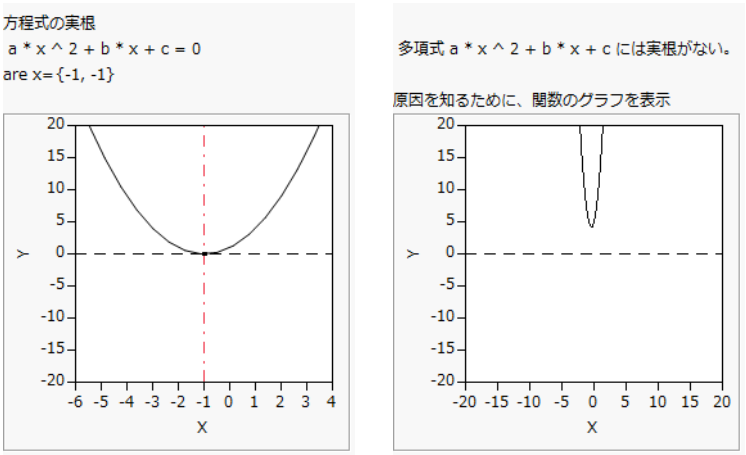
このスクリプトを実行すると、最初に、次のようなウィンドウが表示されます。

図 11.20 例: 独自に作成したウィンドウ



[OK] をクリックすると、根またはエラーメッセージのどちらかを示す、結果のウィンドウが表示されます (図 11.21)。もう一度スクリプトを実行し、フィールドに 5、4、5 とそれぞれ入力して [OK] をクリックします。JMP は実根がないというメッセージを表示します (図 11.21 の右側)。

図 11.21 例: 独自に作成したレポート



シート

Sheet Box を使うと、複数のグラフを縦横に並べることができます。V Sheet Box と H Sheet Box は、そこに含まれているディスプレイボックスを列と行に並べます。まず、どのディスプレイボックスを、どのように配置するかを検討します。そして、H Sheet Box または V Sheet Box を作成し、表示したいグラフに対して Hold メッセージを送ります。最後に、内側に配置する H Sheet Box または V Sheet Box を作成し、それぞれにどのグラフを収納するかを指示します。

次の例では、4 つのグラフ (二変量の散布図、一変量のヒストグラム、ツリーマップ、バブルプロット) を含むシートを作成します。

まず、データテーブルを開き、新しいウィンドウを作成します。

```
Open("$SAMPLE_DATA¥Big Class.jmp");
New Window( "例",
```

V Sheet Boxを使い、ウィンドウを2列に並べることにしましょう。

```
V Sheet Box(
```

このVSheetBoxに、各グラフに1つずつ、合計4つのHoldメッセージを送ります。このとき、順序が重要です。

```
<<Hold(Bivariate( // プロット 1
      Y( :Name("体重 ( ポンド )") ),
      X( :Name("身長 ( インチ )") ),
      Fit Line()
)),
<<Hold(Distribution( // プロット 2
      Continuous Distribution(
         Column( :Name("身長 ( インチ )") ),
         Horizontal Layout( 1 ),
         Outlier Box Plot( 0 )
      );
)),
<<Hold(Treemap(Categories( :年齢 ))), // プロット 3
<<Hold(Bubble Plot( // プロット 4
      X( :Name("身長 ( インチ )") ),
      Y( :Name("体重 ( ポンド )") ),
      Sizes( :年齢 ),
      Coloring( :性別 ),
      Circle Size( 6.226 ),
      All Labels( 0 )
)),
```

最後に、V Sheet Box内に2 つのH Sheet Boxを追加し、どのグラフを保持するかを指示します。各H Sheet Boxに、横に並べて表示する2つのグラフを保持します。H Sheet BoxはV Sheet Boxによって保持されているので、H Sheet Box全体は縦に表示されます。

```
H Sheet Box(
   Sheet Part("",
```

Sheet Partは、Excerpt Boxに指定されたグラフを表示します。第1引数はグラフの番号で、グラフを定義した際の順番を示します。ですから、この最初のH Sheet Boxには、左側に二変量の散布図が、右側に一変量のヒストグラムが表示されます。{Picture Box(1)}の部分は、レポートのどのピクチャーボックスを表示するかを指定しています。通常は1を使用します。

```
      Excerpt Box( 1, {Picture Box( 1 )} )
   ),
   Sheet Part("身長の分布",
      Excerpt Box( 2, {Picture Box( 1 )} )
   );
),
```

```
        H Sheet Box(
           Sheet Part("",
              Excerpt Box( 3, {Picture Box( 1 )} )
           ),
           Sheet Part("ここにタイトル",
              Excerpt Box( 4, {Picture Box( 1 )} )
           );
        );
     );
  );
```

なお、Sheet Partのタイトルについて注意すべき点があります。空の文字列をタイトルとして指定した場合、デフォルトのレポートタイトルになります。たとえば、「身長(インチ)と体重(ポンド)の二変量の関係」などです。特定の文字列で指定すると、その文字列がタイトルに設定されます。この例では、「ここにタイトル」という文字列を指定しています。

注：空の文字列であれ、空でない文字列であれ、Sheet Partのタイトルは必ず指定してください。この引数は必須です。また、タイトルを空白にしたい場合は、1つまたは複数のスペースを指定してください。

ジャーナル

ジャーナルウィンドウに要素を表示するのは、比較的簡単ですが、ジャーナル自体の操作はやや困難です。次に、ジャーナル操作の例をいくつか示します。

まず、「Big Class.jmp」データテーブルからレポートを作成します。

```
biv=bivariate(y(:Name("体重(ポンド)")),x(:Name("身長(インチ)")));
rbiv=biv << report;
```

レポート結果のジャーナルを作成するには、journal windowメッセージを使います。

```
rbiv<<journal window;
```

ジャーナルをファイルに追加するには、次のように指定します。

```
rbiv<<Save Journal("Macintosh HD:users:username:Documents:test.jrn");
```

ジャーナルをHTMLとして保存するには、次のように指定します。

```
rbiv<<Save HTML("Macintosh HD:users:username:Documents:test.htm");
```

ここでは、Macintoshの標準的なファイル名の例をご紹介しています。Windowsの場合は、OSに応じたファイル名に変更するか、すべてのプラットフォームで有効なPOSIXを使ってください。

ジャーナルを保存する際にGZ圧縮形式を使用するよう環境設定を設定するには、次のように指定します。

```
rbiv<<Save Journals GZ Compressed(Boolean);
```

または

```
Preferences(Save Journals GZ Compressed( 1 ));
```

スクリプトにBy変数を使用した場合、各Byグループの分析結果への参照が含まれたリストが戻り値になります。すべてのByグループのレポートをジャーナルにまとめるには、parentメッセージを使用してレポートの先頭に移動する必要があります。

たとえば、次のコードは、Byグループを使用した二変量のレポートを作成し、レポート全体のジャーナルを作成します。

```
biv=bivariate (y(:Name("体重(ポンド)")), x(:Name("身長(インチ)")),by(性別));
((report(biv[1]) << parent) << parent) << save journal("テスト.jrn");
```

Picture表示タイプ

JSLには、JMPの出力や計算式のイメージを保存するときに使うPictureというデータ型があります。ディスプレイボックスの中にあるものをイメージで取り出したり、テキストの計算式を、計算式エディタで見るようなイメージで作成したりできます。

イメージデータを作成するには、*displaybox*にGet Pictureメッセージを送ります。

```
displaybox<<Get Picture;
```

関数Expr As Pictureは、引数を評価し、計算式エディタと同じ表示形式メカニズムを使って、計算式のイメージを作成します。式そのものを引数に指定する場合は、評価した結果ではなく、そのままイメージとして表示されるように、忘れずにExpr()でその式を囲んでください。

例:

```
New Window("計算式", Expr As Picture(expr(a+b*c+r/d+exp(x))));
```

イメージには、次の2つの利用方法があります。

1. ディスプレイボックスを構成する関数と一緒に使って、イメージを表示する。
2. Save Pictureを使って、ファイルに書き込む。

   ```
   picture<<Save Picture("path", type)
   ```

タイプ（*type*）には、WMF(Windows)、EMF(Windows)、PICT(Macintosh)、BMP(Windows)、JPEG（またはJPG)、PNGを指定できます。

モーダルウィンドウ

JSLでは、モーダルと非モーダル両方のウィンドウを作成できます。

- モーダルウィンドウとは、ユーザがすぐに応答しなければならないものを指します。ウィンドウの外側をクリックするとエラー音が鳴り、ユーザがウィンドウに対して応答するまでスクリプトの実行が停止されます。

- 非モーダルのウィンドウとは、JMP レポートなど、すぐに応答する必要のないものを指します。プラットフォームの起動ウィンドウは非モーダルです。

注：モーダルウィンドウを作成する Dialog() 関数は廃止され、今後のバージョンの JMP では使用できなくなる可能性があります。Modal 引数とともに New Window() を使用するか、または Column Dialog() を使用して列を選択してください。

Dialog() に代えて New Window() を使用すると、Dialog() 専用の関数ではなく、ディスプレイボックスを構成する標準の関数を使用できます。

モーダルウィンドウの作成

モーダルウィンドウに関連するスクリプトが送られると、JMP はウィンドウを作成し、ユーザが選択して［**OK**］をクリックするのを待ちます。その後で、変数とその値のリストを保存します。**モーダル**の場合、ユーザはウィンドウとやりとりする以外に選択肢がありません。ウィンドウの外をクリックしてもエラー音が鳴るだけで、ユーザが［**OK**］または［**キャンセル**］をクリックするまではスクリプトの実行が停止されます。

特に、Column Dialog 関数は、ユーザに現在（最前面）のデータテーブルから列を選択させるために使用されます。Column Dialog で作成されるウィンドウも、モーダルウィンドウです。

以下は、スクリプトでモーダルウィンドウを使う際のアドバイスです。

1. 可能ならば、すべてのモーダルウィンドウをスクリプトの冒頭に置きます。そうすることで、ユーザと JMP の間のすべてのインタラクティブなやりとりが一度に行われ、その後はユーザがついていなくても JMP が後の処理を進めます。
2. 作成したモーダルウィンドウで、ユーザに充分な情報を提示できているかどうかを確かめます。数値を入力する場所を示すだけでなく、その数値がどのように使われるかも、ユーザに知らせるようにします。入力内容に制限がある場合はそれを伝えます。

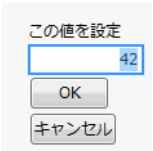
汎用モーダルウィンドウ

最も簡素なモーダルウィンドウの例として、変数の値を 1 つ指定させるものを考えてみます。

```
New Window( " 値の指定 ",
   <<Modal,
   Text Box(" この値を指定 "),
   variablebox = Number Edit Box( 42 ),
   Button Box( "OK"),
   Button Box( " キャンセル ")
);
```

Number Edit Box の引数 42 がその変数のデフォルト値となることに注意してください。また、Button 引数を 1 つも指定しなかった場合、モーダルウィンドウに自動的にボタンが追加されます。

図11.22 モーダルウィンドウの例



[**OK**] がクリックされると、ウィンドウは閉じ、`{Button(1)}` が戻されてスクリプトの実行が継続されます。値を参照するには、`variablebox<<get` を使用します。

[**キャンセル**] がクリックされると、ウィンドウは閉じ、`{Button(-1)}` が戻されてスクリプトの実行が継続されます。スクリプトのキャンセルについての詳細は、「プログラミング手法」の章の「例外のスローとキャッチ」（235 ページ）を参照してください。

注：モーダルウィンドウには、ウィンドウを閉じるためのボタンが少なくとも 1 つ必要です。モーダルウィンドウのボタンのラベルには、[**OK**]、[**Yes**]、[**No**]、または [**Cancel**] が使用できます。モーダルウィンドウにボタンがまったく含まれていない場合、JMP は [**OK**] ボタンを追加します。

廃止された Dialog を New Window に変換する

`Dialog()` は廃止され、代わりに `New Window()` と `<< Modal` メッセージを使用できます。`Dialog()` を含んだ古いスクリプトは現在も使用可能ですが、代わりに `New Window()` を使用することをお勧めします。

以下の節で、`Dialog()` 関数と `New Window()` 関数の違いを示します。

文字列とリスト

`Dialog()` と `New Window()` の大きな違いは、文字列とリストの引数の指定方法にあります。`New Window()` の場合、項目をリストに入れる必要があります。`Dialog()` の場合は、通常、項目をカンマで区切っていました。たとえば、Combo Box の次のような例を比べてみましょう。

```
// モーダルの New Window
New Window( "Combo Box",
   <<Modal,
   cb = Combo Box( {"True", "False"} ), // 項目を中括弧で囲んだリストで指定する
);

// モーダルの Dialog
Dialog(Title("Combo Box"),
   cb = Combo Box( "True", "False" ), // 項目をカンマで区切って指定する
);
```

注：スクリプトは明示的に [**OK**] ボタンを定義していません。モーダルウィンドウの場合、[**OK**] ボタンは自動的に追加されますが、その他のボタンは明示的に定義してください。

さらに、空のテキストを含めるとき、Dialog() では " " という形で指定できますが、New Window() の場合は Text Box (" ") と指定する必要があります。

リストボックスとチェックボックス

New Window() では、H List Box() を使ってボックスを横に並べ、V List Box() を使ってボックスを縦に並べます。Dialog() では、H List および V List を使用していました。

次のスクリプトは、横に並んだ3つのチェックボックスと、1つの [**OK**] ボタンのあるウィンドウを作成します。

```
New Window("H List Box",
   <<Modal,
   H List Box(
      kb1 = Check Box( "a"),
      kb2 = Check Box( "b"),
      kb3 = Check Box( "c")
   ),
);
```

次は、同じウィンドウを Dialog() と H List を使って作成します。

```
Dialog(Title("H List"),
   H List(
         kb1 = Check Box( "a", 0),
         kb2 = Check Box( "b", 0 ),
         kb3 = Check Box( "c", 0)
    ),
);
```

項目の整列

New Window() で、指定の列数に項目を整列させるには、Line Up Box を使用します。Dialog() では Line Up を使用していました。

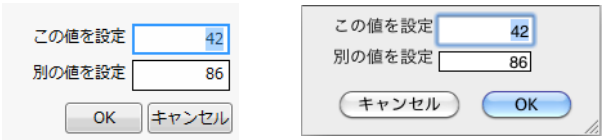
次のスクリプトは、テキストボックスを1列に配置し、数値編集ボックスをもう1列に配置します。

```
New Window("Line Up Box",
   <<Modal,
   V List Box(
      Line Up Box(NCol(2),
      Text Box(" この値を設定 "), var1=Number Edit Box(42),
      Text Box(" 別の値を設定 "), var2=Number Edit Box(86),
      ),
   ),
   H List Box(
   Button Box("OK"),
   Button Box(" キャンセル "))

);
```

［OK］をクリックすると、次の結果がログウィンドウに表示されます。

```
{Button( 1 )}
```

図 11.23 Windows（左）と Macintosh（右）の Dialog によるデフォルトの配置



次は、同じウィンドウを Dialog() と Line Up を使って作成します。

```
Dialog(
   V List(
      Line Up(2,
         " この値を設定 ", variable=Edit Number(42),
         " 別の値を設定 ", var2=Edit Number(86)),
      H List(Button("OK"), Button(" キャンセル "))));
```

［OK］をクリックすると、次の結果がログウィンドウに表示されます。

```
{variable = 42, var2 = 86, Button(1)}
```

注：［OK］と［キャンセル］のボタンの位置は、OS の種類に応じ、ダイアログボックスのスタイルに合わせて調整されます。場合によっては、Button("OK") と Button(" キャンセル ") の配置のために、HList、VList、および LineUp の設定が無効になることがあります。このためダイアログボックスが想定していたものと少し異なる場合もあります。

ラジオボックス

廃止された Dialog() 関数と New Window() 関数のもう 1 つの違いは、Radio Box の使い方です。

New Window() では、Radio Box に Panel Box コンテナを使用する必要があります。

```
New Window("Radio Box",
   << Modal,
   Panel Box( " 選択 ",
      rbox = Radio Box( {"a", "b", "c"} )
   )
);
```

Dialog() では、Radio Buttons は自動的にパネルボックス内に表示されていました。

```
Dialog(Title("Radio Buttons"),
   rb = Radio Buttons( {"a", "b", "c"} ),
);
```

編集可能なテキストボックス

New Window() で、指定の文字列を含む編集可能なボックスを作成するには、Text Edit Box を使用します。Dialog() では、Edit Text を使用していました。

```
New Window("Text Edit Box",
   <<Modal,
   V List Box(
   Text Box(" 文字列 "),
   str1=Text Edit Box("The"),str2=Text Edit Box("quick"),
   str3=Text Edit Box("brown"),str4=Text Edit Box("fox"),
   str5=Text Edit Box("jumps"),str6=Text Edit Box("over"),
   str7=Text Edit Box("the"),str8=Text Edit Box("lazy"),
   str9=Text Edit Box("dog")
    )
);
```

次は、同じウィンドウを Dialog() と Edit Text を使って作成します。

```
Dialog(Title("Edit Text"),
   V List(" 文字列 ",
   str1 = Edit Text( "The" ),
   str2 = Edit Text( "quick" ),
   str3 = Edit Text( "brown" ),
   str4 = Edit Text( "fox" ),
   str5 = Edit Text( "jumps" ),
   str6 = Edit Text( "over" ),
   str7 = Edit Text( "the" ),
   str8 = Edit Text( "lazy" ),
   str9 = Edit Text( "dog" ) )
);
```

指定の文字列を含む編集可能なボックスを作成するもう 1 つの方法は、String Col Edit Box を使用する方法です。たとえば、次のスクリプトは「String Col Box」という名前のウィンドウを作成します。このウィンドウには「文字列」という名前の編集可能なボックスの列があり、各ボックスに指定の文字列が挿入されます。

```
New Window( "String Col Edit Box",
   <<Modal,
   String Col Edit Box(
      " 文字列 ",
      {"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}
   )
);
```

編集可能な数値ボックス

New Window() で、指定の数値を含む編集可能なボックスを作成するには、Number Edit Box を使用します。Dialog() では、Edit Number を使用していました。たとえば、次のように使用できます。

```
New Window( "Number Edit Box",
   <<Modal,
   V List Box(
      Text Box("乱数"),
      num1 = Number Edit Box(Random Uniform()),
      num2 = Number Edit Box(Random Uniform() * 10),
      num3 = Number Edit Box(Random Uniform() * 100),
      num4 = Number Edit Box(Random Uniform() * 1000)
   ),
);
```

次は、同じウィンドウをDialog()とEdit Numberを使って作成します。

```
Dialog(
   Title( "Edit Number" ),
   V List(
      "乱数",
      num1 = Edit Number( Random Uniform() ),
      num2 = Edit Number( Random Uniform() * 10 ),
      num3 = Edit Number( Random Uniform() * 100 ),
      num4 = Edit Number( Random Uniform() * 1000 )
   ),
);
```

同じウィンドウを作成するもう1つの方法は、Number Col Edit Boxを使って指定の数値を含む編集可能なボックスを作成する方法です。たとえば、次のスクリプトは「Number Col Edit Box」という名前のウィンドウを作成します。このウィンドウには「乱数」という名前の編集可能なボックスの列があり、各ボックスに指定した種類の乱数から得られた値が表示されます。

```
New Window( "Number Col Edit Box",
   <<Modal,
   nceb = Number Col Edit Box(
      "乱数",
      {num1 = Random Uniform(), num2 = Random Uniform() * 10, num3 =
      Random Uniform() * 100, num4 = Random Uniform() * 1000}
   ),
);
```

オプションのスクリプトの追加

New Window()を使用するメリットの1つは、ディスプレイボックスにオプションのスクリプトを追加できることです。Dialog()では、次のようなCombo Boxにオプションのスクリプトを含めることはできませんでした。ディスプレイボックスのコントロールのいずれかに関連したアクションが必要なとき、コントロールの最後の引数としてスクリプトを追加する必要がありました。例:

```
New Window( <<Modal,
   comboObj = Combo Box(
      {"True", "False"},
      <<Set( 1 ),
      Print( comboObj << Get )
   )
);
```

ユーザが異なる値を選択したとき、選択された項目の番号（この例では、コンボボックス内の項目数が2なので1または2）がログに出力されます。

Dialog と New Window の違い

次の2つのコード例は、同じウィンドウを、廃止された Dialog() を使用して作成した場合と、推奨する New Window() を Modal オプションとともに使用して作成した場合です。New Window では、より多くの表示オプションを使用でき、また、ウィンドウの内容と機能をより具体的にコントロールできます。

Dialog の例

```
Dialog(
   Title( "Dialog の例 " ),
   H List(
      V List(
         " 分析のパラメータ ",
         Line up( 2,
            " 下側仕様限界 ", lsl = Edit Number( 230 ),
            " 上側仕様限界 ", usl = Edit Number( 340 ),
            " 閾値 ", threshold = Edit Number( 275 )
         ),
         H List(
            V List(
               " ラジオの種類 ",
               type = Radio Buttons( "RCA", "Matsushita", "Zenith", "Sony" )
            ),
            V List(
               " アンテナの種類 ",
               antenna = Radio Buttons( " パラボラ ", " ヘリカル ", " 極性 ",
                  " ラジアントアレイ " )
            );
         ),
         synch = Check Box( " 放射同期 ", 0 ),
         " グラフのタイトル ",
         title = Edit Text( " 分析結果 " ),
         H List(
            " 品質 ",
            quality = Combo Box( " 最優良 ", " 優良 ", " 良 ", " 可 "
```

```
                );
            );
        ),
        V List( Button( "OK" ), Button( "キャンセル" ) )
    );

);
```

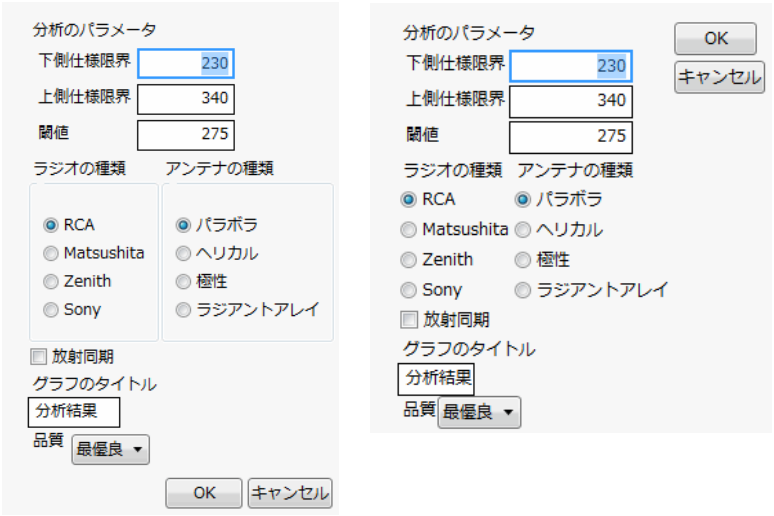
New Windowの例

次の例は、縦と横のリストボックスが複数ある同じウィンドウを作成します。

```
New Window( "New Windowの例",
    <<Modal,
    V List Box(
        V List Box(
            Text Box( "分析のパラメータ" ),
            Line up Box(
                NCol(2),
                Text Box( "下側仕様限界" ),
                lsl_box = Number Edit Box( 230 ),
                Text Box( "上側仕様限界" ),
                usl_box = Number Edit Box( 340 ),
                Text Box( "閾値" ),
                threshold_box = Number Edit Box( 275 )
            ),
            H List Box(
                Panel Box("ラジオの種類",
                      rb_box1 = Radio Box( {"RCA", "Matsushita",
                      "Zenith", "Sony"} )),
                Panel Box( "アンテナの種類" ,
                      rb_box2 = Radio Box( {"パラボラ", "ヘリカル", "極性",
                      "ラジアントアレイ"} ) )
            ),
            cb_box1 = Check Box( "放射同期"),
            Text Box( "グラフのタイトル" ),
            title_box = Text Edit Box( "分析結果" ),
            H List Box( Text Box( "品質" ),
            cb_box2 = Combo Box( "最優良", "優良", "良", "可" ) )
        ),
        H List Box(
            Align(Right),
            Spacer Box(),
            Button Box( "OK",
                lsl = lsl_box << get;
                usl = usl_box << get;
                threshold = threshold_box << get;
```

```
            radio_type = rb_box1 << get;
            antenna = rb_box2 << get;
            synch = cb_box1 << get;
            title = title_box << gettext;
            quality = cb_box2 << get;
         ),
         Button Box( "キャンセル" )
      );
   );
);
```

図 11.24 Dialog（左）と New Window（右）の結果



Dialog() の例で［OK］ボタンをクリックすると、指定したすべての変数が戻されます。

```
{lsl = 230, usl = 340, threshold = 275, type = 1, antenna = 1, synch = 0, title
   = "分析結果", quality = 1, Button( 1 )}
```

New Window() の例で［OK］ボタンをクリックすると、クリックされたボタンだけが戻されます。

```
{Button( 1 )}
```

値の取得

Dialog() と New Window() の最も大きな違いは、値を取り出す際の違いにあると言えるでしょう。Dialog() では、ダイアログが閉じた後に、値がアンロードされます。例:

```
dlg = Dialog(
   rb = RadioButtons( {"a", "b", "c"} )
);
```

```
Show( dlg["rb"] );  // 選択されたものを取得
   dlg["rb"] = 3; // 「c」が選択されたことを報告
```

New Window() では、変数値を取得する方法が2つあります。デフォルトでは、New Window() 内の各変数を取得するには、個々に <<Get 式を使用しなければなりません（「取得の方法2」（445ページ）を参照）。New Window() 内の値を Dialog() と同じ方法で戻すには、<<Return Results オプションを使用します（「取得の方法1」（445ページ）を参照）。

取得の方法1

New Window スクリプトで、[**OK**] がクリックされた後に自動的に結果を戻すよう指定するには、<<Modal の後に <<Return Result オプションを含めます。例:

```
nw = New Window("V List Box",
   <<Modal,
   <<Return Result,
   V List Box(
      kb1 = Check Box( "a"),
      kb2 = Check Box( "b"),
      kb3 = Check Box( "c")
   ),
   Button Box("OK")
);
```

チェックボックス a と b が選択された場合、Return Result オプションの結果は次のようになります。

```
   {kb1 = {1}, kb2 = {1}, kb3 = {}, Button( 1 )}
```

取得の方法2

New Window() の例の出力結果を見るには、各変数に <<Get を追加します。選択肢を見るには、スクリプトの末尾に Show 行を追加します（Return Result オプションを含めずに）。

```
nw = New Window("V List Box",
   <<Modal,
   V List Box(
      kb1 = Check Box( "a"),
      kb2 = Check Box( "b"),
      kb3 = Check Box( "c")
   ),
   Button Box("OK",
      val1=kb1 <<get;
      val2=kb2 <<get;
      val3=kb3 <<get;
      ),
);
Show(val1, val2, val3); // ウィンドウが閉じた後に変数を戻す
```

チェックボックスaとbが選択された場合、結果は次のようになります。

```
val1 = 1;
val2 = 1;
val3 = 0;
```

New Window()関数およびそのオブジェクトの詳細については、「モーダルウィンドウと非モーダルウィンドウ」（409ページ）を参照してください。構文の詳細については、『スクリプト構文リファレンス』を参照してください。

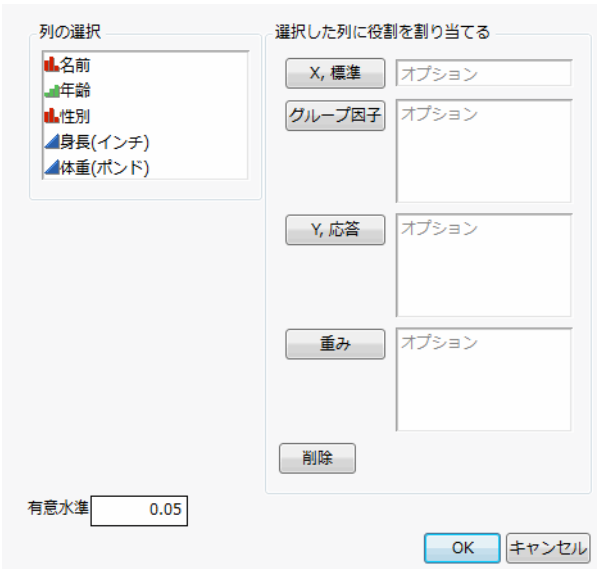
データ列

Column Dialogは、廃止されたDialog関数の変形で、現在のデータテーブルや関連のあるデータテーブル（開いている必要があります）から列を選ぶよう促すことができます。

たとえば、次のような設定が行えます。

```
dt = Open( "$SAMPLE_DATA¥Big Class.jmp" );
r = Column Dialog(
   Col ID = Col List( "X, 標準", Max Col( 1 ) ),
   Group = Col List( "グループ因子" ),
   Split = Col List( "Y, 応答" ),
   w = Col List( "体重(ポンド)" ),
   H List( "有意水準", alpha = Edit Number( .05 ) )
);
```

図11.25 Column Dialog



この例では、ユーザの選択に応じて、次の例のようなリストが戻されます。

```
{Col ID = {}, Group = {}, Split = {}, w = {}, alpha = 0.05, Button( -1 )}
```

これらのリストを得るには、Col List節はColumn Dialogの直接の引数である必要があります（他の引数の中でネストされていてはいけません）。オプションでMaxCol(*n*)引数を指定すると、選択できる列の数を*n*に制限できます。結果のリストには、括弧で囲まれた「名前」のリストが含まれます。リストは**常に**戻されますが、空リストの場合もあります。ColList節は、最大12まで指定することができます。

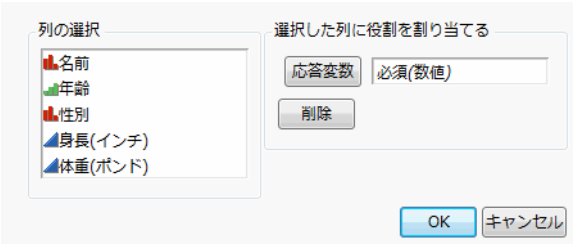
廃止されたDialogコマンドで受け付けられる他の項目はColumn Dialogでも受け付けられ、同じ機能を果たします。[OK]、[キャンセル]、そして［削除］の各ボタンと、選択の対象となる列のリストは、どれも自動的に付加されます。

MinColパラメータとMaxCol引数を使うと、列ダイアログボックスで選択できる列の最大数と最小数を指定できます。選択できる列のデータタイプ（Ordinal、Nominal、またはContinuous）を指定することもできます。「列の選択」リストボックスの幅は、Select List Width(*pixels*)引数を使って設定できます。また、選択された列のリストボックスの幅を設定するには、Col List()関数内でWidth(*pixels*)を使います。

たとえば、次のコードは、1つの数値列だけを選択できる列ダイアログボックスを生成します。

```
rt_Dlg = Column Dialog(
   cv = ColList( "応答変数", MaxCol( 1 ), MinCol( 1 ), DataType( Numeric ) )
);
```

図11.26 選択列の制限



DataTypeとして指定できるのは、Numeric、Character、およびRowStateです。

また、特定の列の選択肢をあらかじめ指定しておくには、Columnsを使います。例:

```
dlg = Column Dialog(
   xCols = Col List( "X, 因子", Columns( :Name("身長(インチ)") )),
   yCols = Col List( "Y, 応答", Columns( :Name("体重(ポンド)"), :年齢 ) )
);
```

このスクリプトでは、役割Xに「**身長(インチ)**」列を、役割Yに「**体重(ポンド)**」列と「**年齢**」列を割り当てています。

結果の取り出し

```
result=New Window(" アンロード ",
   <<Modal,
   <<Return Result,
   V List Box(
      Line Up Box(NCol(2),
         Text Box("Alpha (0-1)"), a=Number Edit Box(0.05),
         Text Box("Sigma (0-5)"), sd=Number Edit Box(1),
         Text Box(" 効果 (0-5)"), eff=Number Edit Box(2),
         Text Box(" 標本 (2-100)"), n=Number Edit Box(2)),
      H List Box(
         Button Box("OK"),
         Button Box(" キャンセル "))
   )
);
   {a = 0.05, sd = 1, eff = 2, n = 2, Button( 1 )}
```

値を使うには、返されたリストからそれらを取り出す必要があります。

```
sd = result["sd"];
eff = result["eff"];
```

リスト内のすべての要素を一度に評価するには、EvalListを使用します。

```
RemoveFrom(result,5); // Button(1) が未定義のため削除
EvalList(result);
   {0.05, 1, 2, 2}
```

ダイアログおよび列ダイアログの作成

表11.4に、Dialog()およびColumn Dialog()に使用されるディスプレイボックス作成用関数を示します。次の点を念頭に置いてください。

- 列ダイアログには、データテーブル内の列のリストが自動的に含まれます。
- どちらのウィンドウも、ボタンが定義されていない場合は、自動的に［**OK**］ボタンが追加されます。
- 列ダイアログには、［**キャンセル**］および［**削除**］ボタンも自動的に追加されます。

表11.4 ダイアログおよび列ダイアログの作成用関数

作成用関数	構文	説明
Dialog	Dialog(*contents of window*)	**注：**JMP 10 で廃止されました。 この表にリストされた任意の項目を含むモーダルウィンドウを作成します。各項目は、セミコロンではなくカンマで区切って指定します。

表11.4 ダイアログおよび列ダイアログの作成用関数（続き）

作成用関数	構文	説明
Column Dialog	Column Dialog(*contents of window*)	役割の指定に使うモーダルウィンドウを作成します。各項目は、セミコロンではなくカンマで区切って指定します。
Col List	*var*=Col List("*role*", <MaxCol(*n*)>, <Datatype(*type*)>)	Column Dialog() の場合にのみ使用可能。 役割（*role*）ボタンを使って選択の対象を作成します。ユーザが選択したものは、*var*={*choice 1, choice 2, ..., choice n*} という形式のリスト項目で戻されます。 列の最小数または最大数を指定する場合は、MaxCol(*n*) または MinCol(*n*) を使用します。 列のデータタイプを指定する場合は、Datatype(*type*) を使用します。*type* に指定できるのは、Numeric、Character、および Rowstate です。
List Box	var=List Box({"item", "item", ...}, width(50), max selected(2), nlines(6));	項目リストを含むディスプレイボックスを作成する。 Dialog() の List Box に指定できる引数は1つだけです（リスト）。複数の引数を指定する場合は、New Window() に List Box を含めます。
HList	HList(*item*, *item*, ...)	項目（*item*）を上端に沿って一定間隔で横一列に並べます。2つの VList を HList の引数にすると、上端に沿って一定間隔に並んだ2つの列ができます。
VList	VList(*item*, *item*, ...)	項目（*item*）を左端に沿って一定間隔で縦一列に並べます。2つの HList を VList の引数にすると、左端に沿って一定間隔に並んだ2つの行ができます。
Line Up	Line Up(*n*, *item_11*, *item_12*, ..., *item_1n*, ..., *item_nn*)	項目（*item*）を *n* 列に並べます。*item_ij* は *i* 行目の *j* 番目の項目です。
Button	Button("OK")、Button("Cancel")	[**OK**] または [**キャンセル**] ボタンを描画します。[**OK**] がクリックされると、Button(1) を戻します。[**キャンセル**] がクリックされると、Button(-1) を戻します。

表 11.4 ダイアログおよび列ダイアログの作成用関数（続き）

作成用関数	構文	説明
string	"*string*"	ウィンドウ内のテキストを描画します。たとえば、Edit Number フィールドの前に文字列のラベルをつけることができます。文字列は引用符で囲む必要があります。
Edit Number	*var*=Edit Number(*number*) *var*(Edit Number(*number*))	*number* をデフォルト値とした、数の編集フィールドを作成します。[**OK**] がクリックされると、フィールドに入力された数を変数に割り当てます。 Column Dialog() の場合は、var(Edit Number (...) を使用します。Dialog() の場合は、どちらの構文も使用できます。
Edit Text	*var*=Edit Text("*string*", <width(*x*)>) *var*(Edit Text("*string*", <width(*x*)>))	*string* をデフォルト値とした、文字列の編集フィールドを作成します。また、ボックスの最小幅をピクセルで指定できます。デフォルトの幅は 72 ピクセルです。[**OK**] がクリックされると、フィールドに入力されたテキストを変数に割り当てます。 Column Dialog() の場合は、var(Edit Text (...) を使用します。Dialog() の場合は、どちらの構文も使用できます。
Radio Buttons	*var*=Radio Buttons("*choice1*", "*choice2*", ...) *var*(RadioButtons("*choice1*", "*choice2*", ...))	指定した選択肢で、縦に並んだ左揃えのラジオボタンのリストを作成します。最初の選択肢がデフォルトです。[**OK**] がクリックされると、選択されているボタンが変数に割り当てられます。選択肢は引用符で囲んだ文字列で指定する必要があります。 Column Dialog() の場合は、var(Radio Buttons(...) を使用します。Dialog() の場合は、どちらの構文も使用できます。

表 11.4 ダイアログおよび列ダイアログの作成用関数（続き）

作成用関数	構文	説明
Check Box	*var*=Check Box("*Text after box*", *<1|0>*) *var*(CheckBox("*Text after box*", *<1|0>*))	[OK] がクリックされると、選択されているチェックボックスの変数には 1 が割り当てられます。選択されていないチェックボックスは、変数に 0 を割り当てます。 オプションで、ウィンドウが開いたときに選択可能（オン）のチェックボックスに 1 を、選択不可（オフ）のチェックボックスに 0 を追加できます。デフォルトの値は 0（オフ）です。 Column Dialog() の場合は、var(Check Box(...) を使用します。Dialog() の場合は、どちらの構文も使用できます。
Combo Box	*var*=Combo Box("*choice1*", "*choice2*", ...) *var*(ComboBox("*choice1*", "*choice2*", ...))	リストされた選択肢を使ってメニューを作成します。最初の選択肢がデフォルトです。選択肢は引用符で囲んだ文字列で指定する必要があります。または、リストで指定します。 Column Dialog() の場合は、var(Combo Box(...) を使用します。Dialog() の場合は、どちらの構文も使用できます。

スクリプトエディタのスクリプト

スクリプトエディタウィンドウも表示ツリーの 1 つなので、JSL スクリプトを記述して、スクリプトエディタウィンドウの内容を変更したり保存したりできます。

New Script というコマンドはありません。代わりに、新しいスクリプトウィンドウを開く場合は、New Window 関数を使い、それがスクリプトウィンドウであることを知らせるメッセージを送ります。

```
ww = New Window(" ウィンドウタイトル ", <<Script, "Initial Contents");
```

最後の引数はオプションです。文字列を指定した場合は、新しいスクリプトウィンドウにその文字列が含まれます。

前述の New Window の例では、ww はウィンドウ全体であるディスプレイボックスへの参照です。スクリプトウィンドウに書き込みを行うには、書き込み先であるディスプレイボックス部分への参照を取得する必要があります。これは Script Box と呼ばれます。

```
ed = ww[scriptbox(1)];
```

このように取得した参照（この例では、ed）を使えば、テキストの追加、削除、取得ができます。

```
ed << get text();
   "Initial Contents"
```

スクリプトウィンドウ内のテキストすべてを設定するには、Set Textを使用します。次のコマンドは、スクリプトウィンドウ内のすべてのテキストをクリアした後、aaa=3;を追加し、リターンを入力します。

```
ed << set text("aaa=3;\!N");
```

スクリプトウィンドウの最後にテキストを追加するには、Appendを使用します。

```
ed << append text("bbb=1/10;");
ed << append text("\!Nccc=4/100;");
```

指定の行番号のテキストを取得するには、Get Line Textコマンドを使用します。指定の行番号のテキストを新しいテキストに置き換えるには、Set Line Textコマンドを使用します。

```
ed << get line text(2);
ed << set line text(2, "bbb = 0.1;");
```

スクリプト内の総行数を取得するには、Get Line Countコマンドを使用します。Get Linesコマンドは、各行のテキストを要素としたリストを戻します。

```
ed << get line count();
ed << get lines();
```

スクリプトの体裁を読みやすく整えるには、Refromatコマンドを使用します。

```
ed << reformat();
```

スクリプトウィンドウ内のスクリプト全体を実行するには、Runを用います。

```
ed << run();
```

スクリプトウィンドウを閉じるには、他のJMPウィンドウの場合と同じく、ウィンドウにClose Windowコマンドを送ります。

```
ww << close window(nosave);
```

構文リファレンス

表11.5に、レポートに表示される共通のディスプレイボックスの概要を示します。作成も可能なディスプレイボックスには、それに使用するJSL関数がリストされています。ディスプレイボックスの多くは、JSLでアクセスできる表示ツリーの一部ではあっても、JSLで作成することはできません。

その他のディスプレイボックスの詳細については、［スクリプトの索引］を参照してください（［**ヘルプ**］メニューから［**スクリプトの索引**］を選択）。

表11.5 ディスプレイボックスと表示のJSL関数

ボックスの種類	JSL関数	説明
Axis Box		目盛り、目盛りラベル、軸ラベル、ラベルの向きなどの軸の設定を含む。
Border Box	Border Box(<Left(pix)>, <Right(pix)>, <Top(Pix)>, <Bottom(Pix)>, <Sides(0)>, db)	1～4つの辺上にスペースを追加するのに使用されるコンテナ。
Button Box	Button Box(title, <<Set Icon("path"), script)	指定のタイトル（*title*）がついたボタンを作成する。このボタンがクリックされたときに指定のスクリプト（*script*）を実行します。
Cat Axis Box		カテゴリカル軸用の軸ボックス。
Cell Plot Box		セルプロットを含む。
Center Box		この中のディスプレイボックスを中央に配置する。
Check Box	Check Box(list, <script>)	1つまたは複数のチェックボックスを表示するディスプレイボックスを作成する。
Check Box Box		チェックボックスの中に1つのチェックボックスを含む。
Col List Box	Col List Box(<Data Table(<name>)>, <all>, <width(pix)>, <maxSelected(n)>, <nlines(n)>, <script>, <MaxItems(n)>, <MinItems(n)>, <character|numeric>, <On Change(expr)>)	データテーブルの列を選択できるリストボックスを表示するディスプレイボックスを作成する。列リストを保持するリストボックスボックスを作成します。
Combo Box	Combo Box(list, <script>)	メニュー付きのコンボボックスを表示するディスプレイボックスを作成する。
Context Box	Context Box(displayBox, ...)	変数や式のスコープを限定するディスプレイボックスを定義する。各コンテキストボックスは、独立して、個別に評価・実行されます。評価コンテンツボックスを作成します。
Crosstab Box		分割表のコンテナ。
Display 3D Box		3次元のScene Boxのコンテナ。

表 11.5 ディスプレイボックスと表示の JSL 関数（続き）

ボックスの種類	JSL 関数	説明
Excerpt Box	Excerpt Box(report, subscripts)	レポートから一部分を抜粋し、その抜粋したディスプレイボックスを戻す。レポートの番号を *report* で指定し、ディスプレイのリストを *subscripts* で指定します。subscripts に指定する番号は、抜粋したディスプレイを除いた後の番号。
Expr As Picture	Expr As Picture(expr(...))	計算式エディタに表示される式と同じ形式で、expr() をイメージに変換する。Pict Box を作成します。
Frame Box		グラフのフレームを含む。
Global Box	Global Box(global)	グローバル変数（*global*）の値が表示されるボックスを作成する。値は直接編集することができ、編集するたびに自動的にグラフが更新されます。
Graph 3D Box	Graph 3D Box(properties)	3 次元散布図を含むディスプレイボックスを作成する。Scene Box を作成します。
Graph Box	Graph Box(properties, script)	X 軸および Y 軸のあるグラフを作成する。Axis Box や Frame Box などのディスプレイボックスを作成します。
H Center Box	H Center Box(display box)	引数 display box に含まれているすべての子ディスプレイボックスを水平方向に中央揃えで配置して戻す。Center Box を作成します。
H List Box	H List Box(display box, ...)	ディスプレイボックスを作成し、その中に別のディスプレイボックスを横に並べて表示する。リストボックスを作成します。
H Sheet Box	H Sheet Box(<<Hold(report), display boxes)	引数に指定された複数のディスプレイボックスを、横方向に配置したディスプレイボックスを戻す。<<Hold() メッセージは、抜粋元となるレポートがどのシートに属するかを指定します。
HierBox	Hier Box("text", Hier Box(...), ...)	テキスト（text）が含まれているツリーのノードを作成する。これは、特性要因図プラットフォームの出力と同じです。Hier Box は別の Hier Boxe を含めることができ、ツリーを作成できます。テキスト（*text*）は、Text Edit Box でもかまいません。

表 11.5 ディスプレイボックスと表示の JSL 関数（続き）

ボックスの種類	JSL 関数	説明
Icon Box	Icon Box(name)	アイコンを含むディスプレイボックスを作成する。引数で指定する名前（name）には、Popup、Locked、Labeled、Sub、Excluded、Hidden、Continuous、Nominal、Ordinal があります。引数には、イメージへのパスを指定することもできます。
If Box	If Box(Boolean, display boxes)	条件によって中身が表示されるディスプレイボックスを作成する。
Journal Box	Journal Box("Journal Text")	引用符付き文字列（*journal text*）から生成されるジャーナルを表示するディスプレイボックスを作成する。手動でジャーナルテキストを生成することはお勧めしません。
Line Up Box	Lineup Box(<NCol(n)>, <Spacing(pixels)>, display boxes, ...)	*n* 列にボックスを整列させて表示するディスプレイボックスを作成する。
List Box	List Box({"item", ...}, <width(pixels)>, <maxSelected(n)>, <nLines(n)>, <script>)	リストボックスを表示するディスプレイボックスを作成する。引数 item には、項目名の文字列を含むリストを指定します。もしくは、項目名の文字列と、尺度または並べ替え順序を示す文字列を含むリストのリストを指定します。後者の場合、リストボックス内の項目の横に、尺度または並べ替え順序のアイコンが表示されます。
List Box Box		Col List Box() によって作成される。
Matrix Box	Matrix Box(matrix)	与えられた行列（*matrix*）を表示する。
Number Col Box	Number Col Box("title", numbers)	指定されたタイトル（*title*）をつけた列を作成し、リストまたは行列の形で与えられた数値を挿入する。
Number Col Edit Box	Number Col Edit Box("title", numbers)	指定されたタイトル（*title*）をつけた列を作成し、リストまたは行列の形で与えられた数値を挿入する。この関数によって作成された列の数値は、編集できます。
Number Edit Box	Number Edit Box(value)	*value* 引数で指定された数値が入力された数値ボックスを作成する。

表 11.5 ディスプレイボックスと表示の JSL 関数（続き）

ボックスの種類	JSL 関数	説明
Outline Box	Outline Box("title", display box, ...)	指定された複数のディスプレイボックスを含んだ、*title* という名前の新しいアウトラインを作成する。
Page Break Box	Page Break Box()	ウィンドウが印刷された場合、改ページを強制するディスプレイボックスを作成する。
Panel Box	Panel Box("title", display box)	指定されたタイトル（*title*）のラベルをもつパネルのディスプレイボックスを作成する。そのパネルには、複数のディスプレイボックスを含めることができます。
Pict Box		ピクチャーオブジェクトを含む。Picture Box() など、レポートウィンドウにピクチャーを配置する関数によって作成される。
Picture Box	Picture Box(Open(picture), format)	ピクチャー（画像）を撮る位置を示すボックスを作成する。このボックスは、（ワープロ文書に出力する時などの）画像イメージとテキストに変換する際の、ピクチャーがとられる位置を示すだけのボックスです。
Plot Col Box	Plot Col Box(title, numbers)	数字（*numbers*）をグラフ化したディスプレイボックスに、引用符付き文字列で指定されたタイトル（*title*）を付けて戻す。数字にはリストまたは行列を指定できます。
Popup Box	Popup Box({"command1", script1, "command2", script2, "",...})	赤い三角形ボタンのメニューを作成する。引数のリストには、コマンドの文字列と実行されるスクリプトをペアで指定します。まず、コマンド文字列を指定し、その後にそのコマンドを選択したときに評価される式を指定します。コマンドが空の場合は、区切り線が挿入されます。
Radio Box	Radio Box({"item", ...}, <script>)	一連のラジオボタンを表示するディスプレイボックスを作成する。ラジオボタンが選択されるたびに、オプションのスクリプトが実行されます。
Range Slider Box	Range Slider Box(min, max, low_val, high_val, script)	範囲を設定できるインタラクティブな変数スライダのあるディスプレイボックスを作成する。

表 11.5 ディスプレイボックスと表示の JSL 関数（続き）

ボックスの種類	JSL 関数	説明
Scene Box	Scene Box(x size, y size)	3D グラフィック用のシーンボックスを作成する。ボックスの横幅は *x size*、高さは *y size* に設定される。
Script Box	Script Box(<script>, <width>, <height>)	引用符付き文字列で指定されたスクリプト（*script*）を含む編集ボックスを作成する。このボックスはスクリプトウィンドウの一種で、JSL を編集・実行できます。
Scroll Box	Scroll Box(<size(h,v)>, <flexible(Boolean)>, displayBox, ...)	スクロールバーを使って表示する、より大きな子ボックスを配置したディスプレイボックスを作成する。
Sheet Box		1 つまたは複数の Sheet Panel Box のためのコンテナ。
Sheet Panel Box		シートレイアウトのパネル 1 つのためのコンテナ。
Sheet Part	Sheet Part(title, display box)	引用符付き文字列 *title* をタイトルにして、指定のディスプレイボックス（*display box*）を含んだディスプレイボックスを戻す。
Slider Box	Slider Box(min, max, global variable, script, <set width(n)>, <rescale slider(min, max)>)	インタラクティブなスライダコントロールを作成する。
Spacer Box	Spacer Box(<size(h,v)>, <color(color)>)	他のディスプレイボックスとの間のスペースを維持するため、または、LineUp Box 内のセルを埋めるために使用されるディスプレイボックスを作成する。
String Col Box	String Col Box("title", {"string", ...})	リストされた文字列（*string*）を含んだ列を、表の中に作成する。
String Col Edit Box	String Col Edit Box("string", ...)	リストされた文字列（*string*）を含んだ列を、表の中に作成する。この関数によって作成された列の文字列は、編集できます。
Tab Box	Tab Box("page title1", contents of page 1, "page title 2", contents of page 2, ...)	タブ付きのウィンドウペインを作成する。引数には、タブページの名前とタブページの内容を交互に同数指定します。Tab List Box を作成します。

表 11.5 ディスプレイボックスと表示の JSL 関数（続き）

ボックスの種類	JSL 関数	説明
Table Box	Table Box(display box, ...)	指定のディスプレイボックス（*display box*）を列としたレポートテーブル（表）を作成する。
Tab List Box		Tab Pane Box を含める。Tab Box によって作成されます。
Tab Pane Box		タブボックスの 1 つのタブペインに表示されるディスプレイボックスを含む。
Text Box	Text Box("text", <arguments>)	引用符付き文字列で指定されたテキスト（*text*）を含んだボックスを作成する。
Text Edit Box	Text Edit Box("text", <arguments>)	引用符付き文字列（*text*）を含む編集ボックスを作成する。Set Script メッセージを送ることによって、テキストにスクリプト（script）を追加できます。
V Center Box	V Center Box(display box)	引数 display box に含まれているすべての子ディスプレイボックスを垂直方向に中央揃えで配置して戻す。Center Box を作成します。
V Sheet Box	V Sheet Box(<<Hold(report), display boxes)	引数に指定された複数のディスプレイボックスを、縦方向に配置したディスプレイボックスを戻す。<<Hold() メッセージは、抜粋元となるレポートがどのシートに属するかを指定します。
Web Browser Box	Web Browser Box(url)	Web ページを表示するディスプレイボックスを作成する。Windows でのみ可能です。

表 11.6 ディスプレイボックスに対する添え字

記号	構文	説明
[ ]	*db*["*text*"]	ディスプレイボックスの参照（*db*）内で指定のタイトル（*text*）を持つアウトラインボックスを見つける。タイトル（*text*）としては、一部だけでなく完全なものが必要ですが、一部だけで検索したいときは「?」をワイルドカード文字として使用できます。たとえば「?推定値」とすれば「パラメータ推定値」を見つけられます。詳細は、「ディスプレイボックスオブジェクトの参照」（384 ページ）を参照してください。

表 11.6 ディスプレイボックスに対する添え字（続き）

記号	構文	説明
	db[Outline Box("*text*")]	指定のテキスト（*text*）が含まれたアウトラインボックスを見つける。
	db[Column Box("*name*")]	指定のテキスト（*name*）が含まれた列ボックスを見つける。
	db[*boxType*(*n*)]	指定の種類（*boxType*）のディスプレイボックスのうち、*n*番目のものを見つける。
	db[*arg1*, *arg2*, *arg3*, ...]	最後の引数に該当するものを、最後から2番目の引数のアウトラインノードにあるディスプレイボックス内で見つけ、最後から2番目の引数に該当するものは、最後から3番目の引数のアウトラインノードにあるディスプレイボックス内で見つけ…と続く。これは、アウトラインツリーの階層を徐々に下がりながら入れ子になっているディスプレイボックスを探す方法です。

第 12 章

スクリプトによるグラフ作成
2次元プロットの作成と編集

グラフを描画するスクリプトを作成することができます。グラフ描画のためのスクリプトは、分析プラットフォームのほとんどすべてのグラフに対して追加できますし、独自の新しいグラフを作成することもできます。どちらの場合でも、JSL スクリプトはグラフ内に保持され、グラフが再表示されるたびに保持されているスクリプトが実行されます。

3次元プロットのスクリプトについては、「3D シーン」（499 ページ）の章を参照してください。

目次

グラフへのスクリプトの追加 463
JSL を使ったグラフィック要素の並べ替え 464
グラフへの凡例の追加 465
独自のグラフを始めから作成する 465
グラフの変更 466
グラフの要素 468
プロット用の関数 468
グラフフレームのプロパティの取得 473
凡例の追加 473
線、矢印、点、図形の描画 474
線 474
矢印 476
マーカー 477
扇形と円弧 479
一般的な図形: 円、長方形、楕円 480
その他の図形: 多角形と等高線 483
テキストの追加 486
色 487
透明度 488
塗りつぶしのパターン 489
線の種類 489
ピクセルを使った描画 490
インタラクティブなグラフ 491
Handle 492
MouseTrap 495
Drag 関数 496
トラブルシューティング 498

グラフへのスクリプトの追加

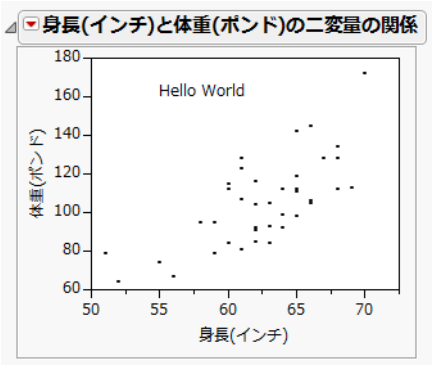
グラフのフレームをコンテキストクリック（マウスの右ボタンをクリック）すると、JSL コマンドの入力や貼り付けができるようになります。通常、スクリプトには、グラフフレームのコンテキスト内で実行される描画コマンドが含まれています。たとえば、次のような設定が行えます。

1. サンプルのデータテーブル「Big Class」を開きます。
2. ［**分析**］>［**二変量の関係**］を選択します。
3. ［X, 説明変数］に「**身長 (インチ)**」列、［Y, 目的変数］に「**体重 (ポンド)**」列を選んで［**OK**］をクリックします。
4. グラフ内を右クリックします。
5. コンテキストメニューから［**カスタマイズ**］を選択します。
6. プラス記号をクリックして、新しいグラフィックスクリプトを追加します。
7. 次のテキストを入力し、［**OK**］をクリックします。

```
text({55,160},"Hello World");
```

これで、グラフの x 座標 55、y 座標 160 の位置にテキストが表示されました。

図 12.1 スクリプトをグラフに追加



デフォルトでは、追加したグラフィックスクリプトによる描画は、散布図のデータ点に上書きされて描かれます。

たとえば、次の行をグラフに追加します。

```
Fill Color("Green"); Rect(57, 175, 65, 110, 1);
```

すると、緑で塗りつぶされた四角形が表示されます。スクリプトはリストの順番どおりに描かれるので、リストの最初の項目が最初に描かれます。長方形を一番後ろに描きたい場合は、一番上に移動します。一番手前に描きたい場合は、一番下に移動します。グラフ上のスクリプトは、すべて並べ替えて任意の順番に描くことができます。新しいスクリプトは、リスト内の選択されている項目のすぐ下に追加されます。

ヒント：スクリプトで列名を参照するときは、参照範囲（スコープ）を明らかにするため、「Column(列名)」、またはコロンを列名の前に置いて「: 列名」と指定してください。

ヒント：この例の手順によって生成されたJSLプログラムは、赤の三角ボタンのメニューから［**スクリプト**］>［**スクリプトをスクリプトウィンドウに保存**］を選択すれば、確認できます。

JSLを使ったグラフィック要素の並べ替え

インタラクティブな方法のほかにも、JSLを使ってグラフィック要素を追加することができます。

```
frame box <<Add Graphics Script(<order>, script)
```

JSLで追加したグラフィック要素は、グラフの最前面にある要素の上に描かれます。オプションの順序（order）引数を使えば、グラフィック要素の描画順を指定できます。orderにはキーワードのBack、または複数のグラフィック要素の描画順を整数で指定します。たとえば、散布図に楕円を追加すると、楕円はマーカーの上に描かれます。キーワードBackを使用した場合、楕円は一番後ろに描かれます。

複数のグラフィック要素の描画順を指定するには、order引数に整数を指定して、それらの描画順を相対的に指定します。次のスクリプトは、まずプロットの点の後ろに青色の楕円を追加し、次に青色の楕円の前で点の後ろに赤色の楕円を追加します。

```
dt = Open( "$SAMPLE_DATA¥big class.jmp" );

op = dt << Overlay Plot(
   X( :Name("身長 ( インチ )") ),
   Y( :Name("体重 ( ポンド )") ),
   Separate Axes( 1 )
);

Report( op )[Framebox( 1 )] << Add Graphics Script(
   1,
   Fill Color( "Blue" );
   Oval( 60, 140, 65, 90, 1 );
);

Report( op )[Framebox( 1 )] << Add Graphics Script(
   2,
   Fill Color( "Red" );
   Oval( 50, 120, 65, 100, 1 );
);
```

グラフへの凡例の追加

Row Legend コマンドを使って、凡例をインタラクティブに追加できます。そのためには、Row Legend メッセージを使用し、凡例の基準にする列と、凡例に色およびマーカーを適用するかどうかを指定します。

「Big Class」を使った以下の例では、「年齢」列に基づいた凡例を表示し、値に応じて色とマーカーの両方を設定します。

```
biv = Bivariate(Y(:Name("身長(インチ)")), X(:Name("体重(ポンド)")));
Report( biv )[Frame Box(1)] << Row Legend( "年齢", color(1), marker(1));
```

color() 引数と marker() 引数はオプションです。デフォルトの動作は、インタラクティブに操作したときと同じです（色はつきますが、マーカーはつきません）。

なお、名義尺度や順序尺度の変数に対して、連続尺度に対する色使いを適用したい場合には、Color(1) に併せて、Continuous Scale(1) を指定してください。

独自のグラフを始めから作成する

独自のグラフを作成したい場合には、New Window コマンド内の Graph Box コマンド内にグラフィックスクリプトを設定します。

```
New Window("ウィンドウタイトル", Graph Box( named arguments,..., script));
```

ウィンドウ名の後には、2通りのキーワードを指定できるオプションが続きます。Editable を指定すると、ジャーナルウィンドウと同じようなウィンドウが作成され、レポート項目をウィンドウ内にドラッグ&ドロップできます。Dialog を指定すると、モーダルウィンドウと同じようなウィンドウが作成され、Macintosh ではグレーの背景色で表示されます。

Graph Box は名前付き引数（named arguments）をとります。

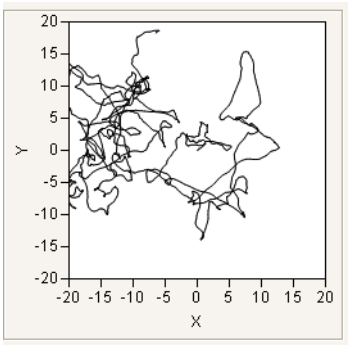
```
FrameSize(horizontal, vertical),          // ピクセル単位のフレームサイズ
XScale(xmin, xmax), YScale(ymin, ymax),   // x、y 軸の範囲
X Axis(messages), Y Axis (messages),      // x、y 軸の設定
Double Buffer                             // アニメーションを滑らかにする
XName, YName                              // x、y 軸の名前
```

以下に、ランダムに走る線を再生するスクリプトを示します。ここでは乱数を使っており、Random Reset で設定される乱数の初期値により、表示される線は異なります。

```
New Window( "ランダムに走る線",
   Graph Box(
      FrameSize( 200, 200 ),
      X Scale( -20, 20 ),
      Y Scale( -20, 20 ),
      x = 0;
```

```
        y = 0;
        xi = 0;
        yi = 0;
        For( i = 0, i < 2000, i++,
           xi = .9 * xi + Random Normal() / 10;
           yi = .9 * yi + Random Normal() / 10;
           xx = x + xi;
           yy = y + yi;
           xx = If(
              xx < -20, -20,
              xx > 20, 20,
              xx
           );
           yy = If(
              yy < -20, -20,
              yy > 20, 20,
              yy
           );
           Line( {x, y}, {xx, yy} );
           x = xx;
           y = yy;
        );
     );
  );
```

図12.2 グラフの作成



グラフの変更

グラフは、スクリプトを使用して変更することもできます。たとえば、次のスクリプトは、グラフを含むウィンドウを作成し、そのレポートへの参照を取得します。

```
Open("$SAMPLE_DATA¥Big Class.jmp");
biv = Bivariate( Y( Name("体重(ポンド)") ), X( :Name("身長(インチ)") ), Fit Line
  );
rbiv = biv<<report;
```

添え字を使えば、そのレポートの任意の部分へメッセージを送ることができます。参照先を確認するには、「**ツリー構造の表示（Show Tree Structure）**」コマンドを使用します。ツリー構造の詳細については、「表示ツリー」の章の「ディスプレイツリーの表示」（382ページ）を参照してください。添え字の使用方法については、「表示ツリー」の章の「添え字の使用」（385ページ）を参照してください。次のスクリプトは、グラフのサイズを400x400ピクセルに変更します。

```
rbiv[frame box(1)]<<frame size(400,400);
```

特定のディスプレイボックスオブジェクトに使用できるメッセージのリストを表示するには、Show Propertiesを使用します。たとえば、以下は、軸に送ることができるメッセージリストの一部です。

```
Show Properties(rbiv[axis box(1)]);
   Axis Settings [アクション](Bring up the Axis window to change various settings.)
   Revert Axis [アクション](Restore the settings that this axis had originally.)
   Add Axis Label [アクション]
   Remove Axis Label [アクション]
   Save To Column Property [アクション](Save the Axis settings as an Axis property
      in the data column associated with this axis.)
   Set Width [アクション] [スクリプトの場合のみ]
   Axis [スクリプト可能] [スクリプトの場合のみ]
     Scale [選択肢] {Linear, Log, Exp Prob, Weibull Prob, Logistic Prob, Frechet
      Prob, Lognormal Prob, Cube Root}
     Min [数値]
     Max [数値]
     Inc [数値]
     Tick Font [アクション]
     Interval [数値] [スクリプトの場合のみ]  {Numeric, Year, Month, Week, Day, Hour,
      Minute, Second}
   ...
```

たとえば、Windows版において、両方の軸におけるラベルのフォントを、12ポイントで斜体（Italic）の「MS P明朝」に変更するには、次のように記述します。なお、軸のラベルはテキスト編集ボックスに保持されています。

```
rbiv[Text Edit Box( 1 )] << set font( "ＭＳ Ｐ 明朝" );
rbiv[Text Edit Box( 1 )] << set font style( "Italic" );
rbiv[Text Edit Box( 1 )] << set font size( 12 );
rbiv[Text Edit Box( 2 )] << set font( "ＭＳ Ｐ 明朝" );
rbiv[Text Edit Box( 2 )] << set font style( "Italic" );
rbiv[Text Edit Box( 2 )] << set font size( 12 );
```

グラフの要素

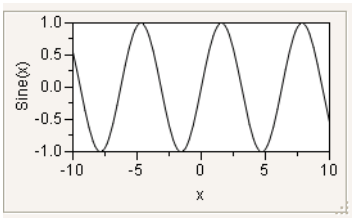
以下は、Graph Boxステートメント内で使用できるコマンドです。この章では、グラフィック専用のJSLを扱いますが、ForやWhileなどの一般的なスクリプトコマンドを使用することもできます。

プロット用の関数

YFunction演算子は、滑らかな連続関数を描くために使います。第1引数はプロットする式で、第2引数は式の中のX変数の名前です。

```
New Window( "正弦関数",
   Graph Box(
      FrameSize( 200, 100 ),
      X Scale( -10, 10 ),
      Y Scale( -1, 1 ),
      xName( "x" ),
      yName( "Sine(x)" ),
      Y Function( Sine( x ), x )
   );
);
```

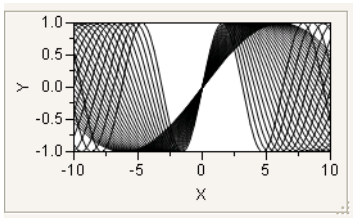
図12.3 正弦波



Forループを使って、複数の正弦波を重ね合わせることもできます。

```
New Window( "重ね合わせた正弦波",
   Graph Box(
      FrameSize( 200, 100 ),
      X Scale( -10, 10 ),
      Y Scale( -1, 1 ),
      For( i = 1, i <= 4, i += .1,
         Y Function( Sine( x / i ), x )
      );
   );
);
```

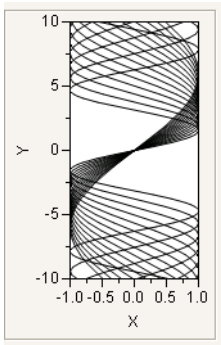
図12.4 重ね合わせた正弦波



同様に、XFunctionを使うと、y変数の値の変化に従ってxの値が変化するプロットを描くことができます。

```
New Window( "重ね合わせた正弦波",
   Graph Box(
      FrameSize( 100, 200 ),
      X Scale( -1, 1 ),
      Y Scale( -10, 10 ),
      For( i = 1, i <= 4, i += .2,
         X Function( Sine( y / i ), y )
      );
   );
);
```

図12.5 X軸に沿って重ね合わせた正弦波

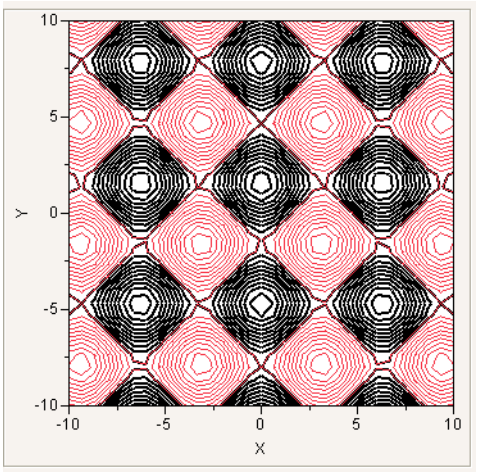


ContourFunctionは、2次元空間で3次元の関数を表現するための方法です。最後の引数には、等高線の値を指定します。単一の値や、::を使って作成した等間隔の値の行列、または任意の値の行列を指定できます。

```
New Window( "卵ケースの鳥瞰図",
   Graph Box(
      FrameSize( 300, 300 ),
      X Scale( -10, 10 ),
      Y Scale( -10, 10 ),
      Pen Color( "black" );
      Pen Size( 2 );
      Contour Function( Sine( y ) + Cosine( x ), x, y, (0 :: 20) / 5 );
```

```
        Pen Color( "Red" );
        Pen Size( 1 );
        Contour Function( Sine( y ) + Cosine( x ), x, y, (-20 :: 0) / 5 );
    );
);
```

図12.6 卵ケース



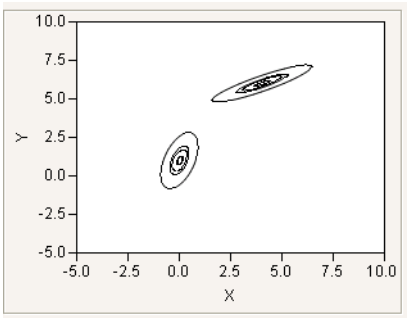
Normal Contourは、2変量正規分布の等高線を描きます。k個の母集団を指定することができます。最初の引数は、等高線の確率を指定するスカラーまたは行列で、後続の引数は、平均、標準偏差、相関係数を指定する行列です。平均と標準偏差の行列の大きさは**k** × 2です。相関係数行列の大きさは**k** × 1で、最初の行は最初の等高線に対応し、2番目の行は2番目の等高線に対応し、以下同様に対応します。最初の列はx、2番目の列はyに対応します。たとえば、次のような設定が行えます。

```
Normal Contour(
    [ prob1,
      prob2,
      prob3, ...],
    [ xmean1 ymean1,
      xmean2 ymean2,
      xmean3 ymean3, ...],
    [ xsd1 ysd1,
      xsd2 ysd2,
      xsd3 ysd3, ...],
    [ xycorr1,
      xycorr2,
     xycorr3, ...]);
```

以下のスクリプトは、2母集団の2変量正規分布の累積確率0.1、0.5、0.7、および0.99における等高線を描きます。最初の母集団は、xの平均0、yの平均1、xの標準偏差0.3、yの標準偏差0.6、相関係数0.5です。2つ目の母集団は、xの平均4、yの平均6、xの標準偏差0.8、yの標準偏差0.4、相関係数0.9です。

```
New Window( "正規確率の等高線",
   Graph Box(
      X Scale( -5, 10 ),
      Y Scale( -5, 10 ),
      Normal Contour( [.1, .5, .7, .99], [0 1, 4 6], [.3 .6, .8 .4], [.5, .9] )
   );
);
```

図12.7 Normal Contour関数



このように、Normal Contourは、2変量正規分布の確率楕円を描きます。2変量正規分布の確率楕円は、「二変量」プラットフォームでも作成できます。「二変量」の確率楕円を確かめるには、サンプルデータの「Football.jmp」を開き、テーブルに保存されているスクリプトを実行してください。

Gradient Function

Gradient Functionの構文は次のとおりです。

```
Gradient Function(expression, xname, yname, [zlow, zhigh], ZColor([colorLow,
   colorHigh]), <XGrid(min, max, incr)>, <YGrid(min, max, incr)>)
```

この関数は、式で指定された色でグリッド上の長方形を塗りつぶします。この関数を実行するには、次の構文を使います。

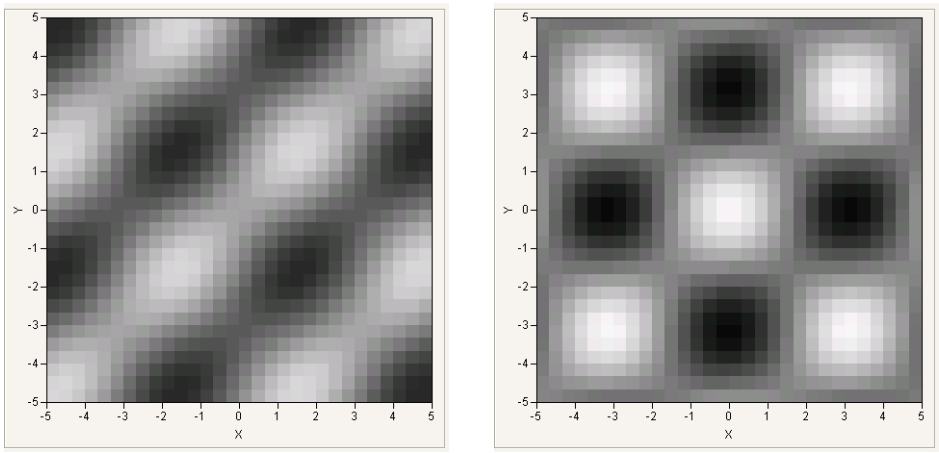
GradientFunction(	
expression	等高線の式。式は、2変数の関数でなければいけません。
xname,yname,	式に含まれている2つの変数の名前。
[zlow, zhigh],	式の上下限値。この2つの値を補間して色が決められます。
ZColor([colorLow, colorHigh])	上限値と下限値それぞれに対応する色
<XGrid(min, max, incr),> <YGrid(min, max, incr)>);	グリッドの指定（オプション）

ZColor 値は名前ではなく数値コードでなければなりません。「色」（487 ページ）に記載されているカラー表の番号を使うことができます（0=black（黒）、1=grey（グレー）、2=white（白）、3=red（赤）、4=green（緑）、5=blue（青）など）。

次のスクリプトは、Gradient Function の使用例です。このスクリプトは動画を作成しますが、次図はその静止画です。

```
phase = 0.7;
New Window( "Gradient Function",
   a = Graph(
      FrameSize( 400, 400 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      DoubleBuffer,
      Gradient Function(
         phase * Sine( x ) * Sine( y ) + (1 - phase) * Cosine( x ) * Cosine( y ),
         x,
         y,
         [-1 1],
         zcolor( [0, 2] )
      );
   );
);
b = a[FrameBox( 1 )];
For( i = 1, i <= 5, i++,
   For( phase = 0, phase < 1, phase += 0.05,
      b << reshow;
      Wait( 0.01 );
   );
   For( phase = 1, phase > 0, phase -= 0.05,
      b << reshow;
      Wait( 0.01 );
   );
);
```

図12.8 Gradient Function



グラフフレームのプロパティの取得

既存のグラフフレームのプロパティを取得するには、以下の関数を使うと便利です。

H Size　グラフフレームの横のサイズをピクセル単位で戻す。

V Size　グラフフレームの縦のサイズをピクセル単位で戻す。

X Origin　グラフフレームの左端の x 値を戻す。

X Range　グラフフレームの左端から右端までの距離を戻す。右端の *x* 値は、XOrigin()+XRange() により得ることができます。

Y Origin　グラフフレームの下端の *y* 値を戻す。

Y Range　ディスプレイボックスの下端から上端までの距離を戻す。

例:

最初の行は右端を計算し、2 番目の行は上端を計算します。

```
rightEdge = X Origin() + X Range();
topEdge = Y Origin() + Y Range();
```

凡例の追加

Row Legend コマンドを使って、凡例をグラフに追加できます。次の例では、サンプルデータファイルの「Fitness.jmp」を使って、「**年齢**」列の値を基準にして色とマーカーを設定し、プロットに凡例を追加しています。

```
biv = Bivariate(Y(:Name(" 酸素摂取量 ")), X(:Name(" 走行時間 "))); // 散布図を生成
Report(biv)[Frame Box(1)] << Row Legend (: 年齢 , color(1), marker(1));
```

marker()引数には、0か1を指定します。1を指定した場合、各点にマーカーが設定されます。

以下のような色の設定ができます。

- Color(0)は凡例の色をオンにします。
- Color(1)は凡例の色をオフにします。

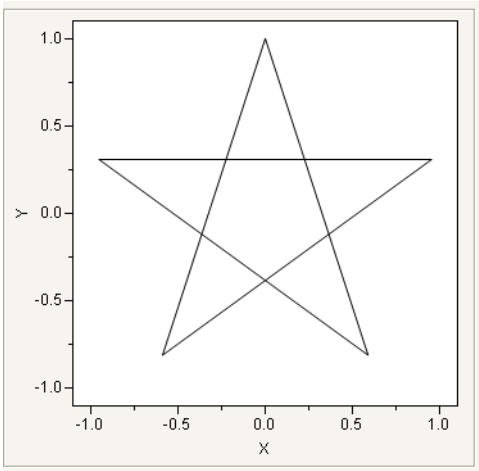
線、矢印、点、図形の描画

線

Lineは点の間に線を引きます。

```
New Window( "星",
   Graph Box(
      framesize( 300, 300 ),
      X Scale( -1.1, 1.1 ),
      Y Scale( -1.1, 1.1 ),
      Line(
         {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )},
         {Cos( 9 * Pi() / 10 ), Sin( 9 * Pi() / 10 )},
         {Cos( 17 * Pi() / 10 ), Sin( 17 * Pi() / 10 )},
         {Cos( 5 * Pi() / 10 ), Sin( 5 * Pi() / 10 )},
         {Cos( 13 * Pi() / 10 ), Sin( 13 * Pi() / 10 )},
         {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )}
      );
   );
);
```

図12.9 線を使って星を描く



点は、上に示したような2項目のリストか、または*x*座標と*y*座標の行列で指定できます。行列の値は、1行目から順番に使われていくので、要素の数が同じであれば、行ベクトルと列ベクトルのどちらでも使用できます。以下のスクリプトはどれも同じになります。

```
Line({1,2}, {3,0}, {2,4}); // 複数の {x,y} リスト
Line([1 3 2],[2 0 4]);     // 行のベクトル
Line([1,3,2], [2,0,4]);    // 列のベクトル
Line([1 3 2], [2,0,4]);    // 行ベクトルと列ベクトルの組み合わせ
```

したがって、五芒星の例は、以下のような方法で描くこともできます。行列の値を式で入力しているので、簡略形の `[ ]` ではなく、正式な `Matrix({...})` 表記を使う必要があります。

```
new window("星",
   graph box(framesize(300, 300), xscale(-1.1, 1.1), yscale(-1.1, 1.1),
      line(
         matrix({ // x座標
            cos(1*pi()/10), cos(9*pi()/10), cos(17*pi()/10),
            cos(5*pi()/10), cos(13*pi()/10), cos(1*pi()/10)}),
         matrix({ // y座標
            sin(1*pi()/10), sin(9*pi()/10), sin(17*pi()/10),
            sin(5*pi()/10), sin(13*pi()/10), sin(1*pi()/10)}))));
```

`HLine`は、グラフ上の指定された*y*値の位置に横線を引きます。同様に、`VLine`は、グラフ上の指定された*x*値の位置に縦線を引きます。どちらのコマンドでも、引数に行列を指定することによって、複数の線を描くことができます。この例は、「MouseTrap」（495ページ）に記載しています。

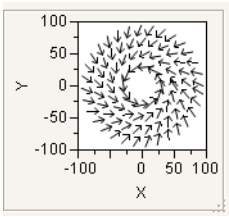
矢印

同様に、Arrowは、最初の点から次の点までの矢印を描きます。デフォルトの矢じりの長さは、(矢印の長さの平方根 + 1)/2です。矢じりの長さを設定するには、オプションの第1引数を使って矢じりの長さをピクセルで設定します。

次のスクリプトは、デフォルトの長さで矢じりを描きます。

```
New Window( "ハリケーン",
   Graph Box(
      FrameSize( 100, 100 ),
      X Scale( -100, 100 ),
      Y Scale( -100, 100 ),
      For( r = 35, r < 100, r += 20,
         ainc = 2 * Pi() * 3 / r;
         For( a = 0, a < 2 * Pi(), a += ainc,
            x = r * Cosine( a );
            y = r * Sine( a );
            aa = a + ainc * 45 / r;
            rr = r - r / 6;
            x2 = rr * Cosine( aa );
            y2 = rr * Sine( aa );
            Arrow( {x, y}, {x2, y2} );
         );
      );
   );
);
```

図12.10 矢印を描く

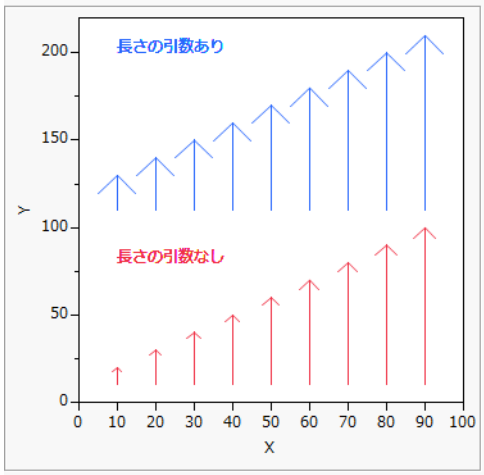


次のスクリプトは、指定された長さ（19ピクセル）とデフォルトの長さで矢じりを描いて比較します。

```
New Window( "矢じり", Graph Box(
   Frame Size( 300, 300 ), X Scale( 0, 100 ), Y Scale( 0, 220 ),

   x = 10; y1 = 10; y2 = y1 + 10;

   For( i = 1, i < 10, i++,
      Pen Color( "Red" );
      Arrow( {x, y1}, {x, y2} );
```

```
        y2 += 10; y1 += 100; y2 += 100;
        Pen Color( "Blue" );
        Arrow( 20, {x, y1}, {x, y2} );

        x += 10; y1 -= 100; y2 -= 100;

        Text Color( "Red" );
        Text( {10, 80}, "長さの引数なし" );

        Text Color( "Blue" );
        Text( {10, 200}, "長さの引数あり" );
    );
));
```

図12.11 矢じりのサイズ



Lineと同様に、上に示したような2項目のリスト、もしくは、*x*座標と*y*座標の行列によって座標を指定できます。

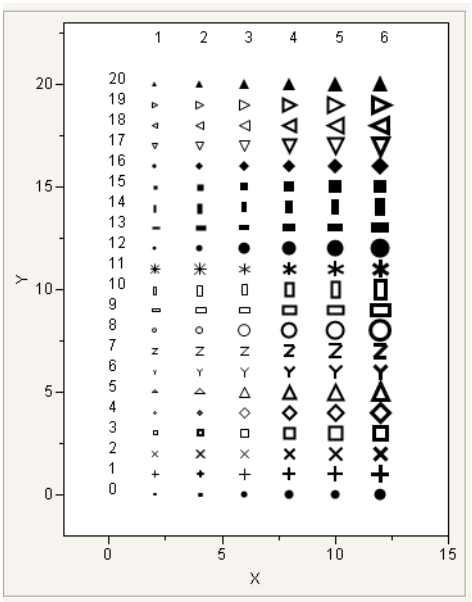
マーカー

Markerは、第1引数で指定されたタイプ（1～15）のマーカーを、第2引数で指定された座標点に描きます。Marker Sizeは、マーカーのサイズを0～6（ドット-XXXL）に設定します。マーカーのサイズを自由に設定するには、-1の値を使用します。

```
ymax = 20;
New Window( "マーカー",
    Graph Box(
        FrameSize( 300, 400 ),
```

```
        X Scale( -2, ymax - 5 ),
        Y Scale( -2, ymax + 3 ),
        For( j = 1, j < 7, j++,
           Marker Size( j );
           For( i = 0, i < (ymax + 1), i++,
              Marker( i, {j * 2, i} );
              Text( {0, i}, i );
              Text( {j * 2, ymax + 2}, j );
           );
        );
     );
  );
```

図12.12 マーカーを描く



マーカータイプの引数の前後、またはその代わりに、行の属性を引数として指定することもできます。Combine Statesを使うと、Markerに複数の行の属性を設定できます。以下に示す例を、前記のグラフスクリプトの中で試してみてください。

```
marker(i, color state(i), {j*2, i});
marker(color state(i), i, {j*2, i});
marker(combine states(colorstate(i),markerstate(i),hiddenstate(i)),{j*2, i});
```

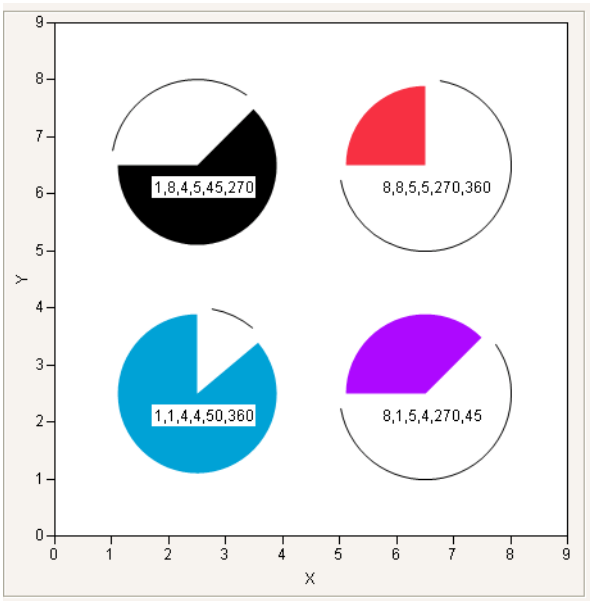
マーカーの座標は、*x*座標と*y*座標の行列で指定することもできます。

扇形と円弧

PieとArcは、それぞれ扇形と円弧を描きます。最初の4つの引数は、*x1*、*y1*、*x2*、*y2*で、内接する長方形の座標です。最後の2つの引数は、度単位の開始角度と終了角度で、時計回りに円弧または扇形が描画されます。このとき、0度は時計の12時の位置とします。

```
New Window( " 扇形と円弧 ",
   Graph Box(
      framesize( 400, 400 ),
      X Scale( 0, 9 ),
      Y Scale( 0, 9 ),
      Fill Color( "Black" ), // 左上
      Pie( 1.1, 7.9, 3.9, 5.1, 45, 270 ),
      Text( erased, {1.75, 6}, "1,8,4,5,45,270" ),
      Arc( 1, 8, 4, 5, 280, 35 ),
      Fill Color( "Red" ), // 右上
      Pie( 7.9, 7.9, 5.1, 5.1, 270, 360 ),
      Text( erased, {5.75, 6}, "8,8,5,5,270,360" ),
      Arc( 8, 8, 5, 5, 370, 260 ),
      Fill Color( "BlueCyan" ), // 左下
      Pie( 1.1, 1.1, 3.9, 3.9, 50, 360 ),
      Text( erased, {1.75, 2}, "1,1,4,4,50,360" ),
      Arc( 1, 1, 4, 4, 370, 40 ),
      Fill Color( "Purple" ), // 右下
      Pie( 7.9, 1.1, 5.1, 3.9, 270, 45 ),
      Text( erased, {5.75, 2}, "8,1,5,4,270,45" ),
      Arc( 8, 1, 5, 4, 55, 260 )
   );
);
```

図 12.13 扇形と円弧を描く



一般的な図形 : 円、長方形、楕円

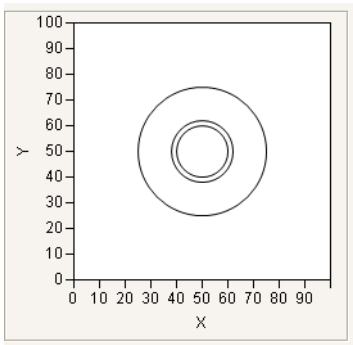
円

`Circle` は、中心点と半径を指定して円を描きます。後続の引数は追加の半径を指定します。

```
New Window( "円",
   Graph Box( framesize( 200, 200 ),
      Circle( {50, 50}, 10, 12, 25 )
   );
);
```

`Circle` には、オプションの最後の引数 `"fill"` もあります。これを指定すると、関数内に定義された円はすべて、現在 fill color で指定されている色で塗りつぶされます。`"fill"` が省略された場合、円は塗りつぶされません。

図12.14 円を描く



グラフの縦横比を変更しても、Circle関数によって描かれた円は常に円のままです。グラフのサイズを変更した際に、それに合わせて円も異なる縦横比に変更されるようにしたい場合には、楕円を描くOval関数を使用してください。

グラフのサイズを変更しても、円のサイズは変更されないようにしたい場合には、半径をピクセルで指定してください。

```
New Window("円",
   Graph Box(framesize(200, 200),
      Circle({50, 50}, PixelRadius(10), PixelRadius(12), PixelRadius(25))));
```

長方形

Rectは、指定された対角座標に基づいて長方形を描きます。座標は順序に従った4つの引数（*左*、*上*、*右*、*下*）か、またはペアのリスト（{*左*,*上*}、{*右*, *下*}）で指定できます。

```
New Window( "長方形",
   Graph Box( framesize( 200, 200 ),
      Pen Color( 1 ); Rect( 0, 40, 60, 0 );
      Pen Color( 3 ); Rect( 10, 60, 70, 10 );
      Pen Color( 4 ); Rect( 50, 90, 90, 50 );
      Pen Color( 5 ); Rect( 0, 80, 70, 70 );
   );
);
```

Rectには、オプションの第5引数 *fill*（塗りつぶし）があります。長方形を塗りつぶさない場合は0を、塗りつぶす場合は1を指定します。長方形は、現在fill colorで指定されている色で塗りつぶされます。*fill*のデフォルト値は0です。

塗りつぶし（fill）の引数が負の値の場合は、塗りつぶしなしの1ピクセルの枠が生成されます。

```
New Window( " 長方形の枠 ",
   Graph Box( framesize( 200, 200 ),
      Rect( 0, 40, 60, 0, -1 )
   );
);
```

楕円

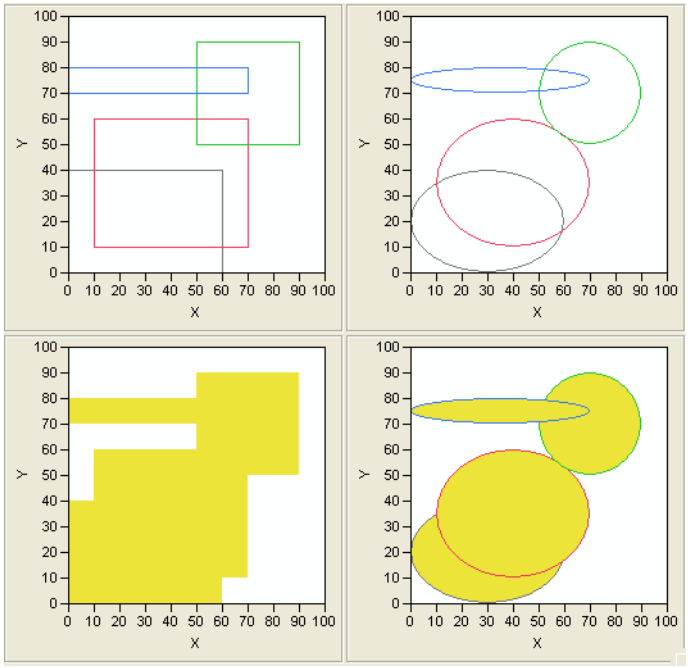
Oval は、引数 *x1*、*y1*、*x2*、*y2* で指定された長方形に内接する楕円を描きます。

```
new window(" 楕円 ",
   graph box(framesize(200,200),
      pen color(1); oval(0,40,60,0);
      pen color(3); oval(10,60,70,10);
      pen color(4); oval(50,90,90,50);
      pen color(5); oval(0,80,70,70)));
```

Oval にもオプションの第5引数 *fill*（塗りつぶし）があります。楕円を塗りつぶさない場合は0を、塗りつぶす場合は1を指定します。楕円は、現在 fill color で指定されている色で塗りつぶされます。*fill* のデフォルト値は0です。

図12.15は、長方形と楕円のグラフです。塗りつぶした場合と、塗りつぶしていない場合の両方を示しています。塗りつぶしを行ったグラフを見ると、長方形には輪郭がなく、楕円には輪郭があります。塗りつぶした長方形に輪郭を描きたい場合は、まず塗りつぶした長方形を描いてから、その後、同じ座標を使用して塗りつぶしていない長方形を描いてください。

図12.15 長方形と楕円（塗りつぶしあり／なし）



その他の図形：多角形と等高線

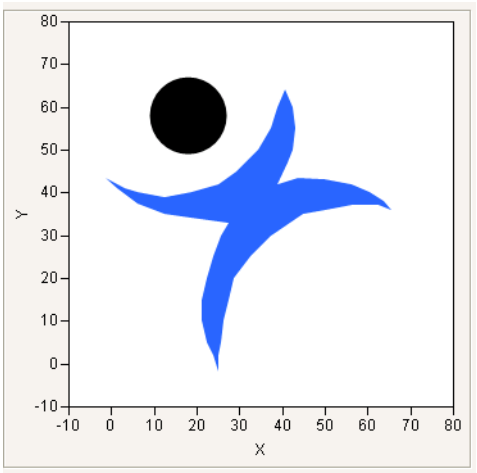
多角形

Polygonは、Lineと同様に点をつなげますが、さらに最後の点と最初の点を結んで多角形を閉じ、多角形の内部を塗りつぶします。各点の座標は、2項目のリストで個々に指定する（前記のMarkerの説明を参照）ことも、*x*座標と*y*座標の行列で指定することもできます。行列の値は、1行目から順番に使われていくので、要素の数が同じであれば、行ベクトルと列ベクトルのどちらでも使用できます。次のスクリプトでは、点の行列を設定した後で、その行列をPolygon()に指定しています。

```
gCoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5,
  6.25, 2.5,
1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75, 40.625,
  42.5, 43.125,
42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50, 45,
  37.5, 32.5,
28.75, 27.5, 26.25, 25.625, 25];
gCoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40,
  39, 40, 42, 45,
50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37, 37, 36, 35, 30,
  25, 20, 15, 10,
5, 2];
```

```
New Window( "JMP マン",
   Graph Box(
      framesize( 300, 300 ),
      X Scale( -10, 80 ),
      Y Scale( -10, 80 ),
      Pen Color( "black" );
      Fill Color( "blue" );
      Polygon( gCoordX, gCoordY );
      Fill Color( "black" );
      Circle( {18, 58}, 9, "FILL" );
   );
);
```

図 12.16 多角形を描く



関連したコマンドの In Polygon は、与えられた点が、指定された多角形の中に入るかどうかを知らせます。次のコードでは、いくつかの点が図 12.16 の JMP マン内にあるかどうかを調べています。

```
In Polygon(0,60, GcoordX,GCoordY);  // 0 を戻す
In Polygon(30,38, GcoordX,GCoordY); // 1 を戻す
```

In Polygon を JMP マンのスクリプトに追加することもできます。以下のスクリプトを実行し、グラフ内のさまざまな位置をクリックして、「ログ」ウィンドウを確認してください。

```
new window("JMP マン",
   graph box(framesize(300,300), xscale(-10,80),yscale(-10,80),
      pen color("black"); fill color("black");
      polygon(gCoordX, gCoordY);
      mousetrap({},print(if(in polygon(x,y,gCoordX,gCoordY),"in","out")))));
```

等高線

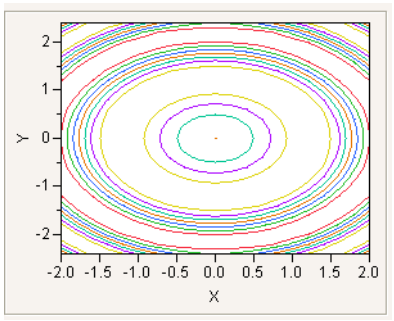
Contourは、座標のグリッドを使って等高線を描きます。構文は、次のとおりです。

```
Contour(xVector,yVector,zGridMatrix,zContour,<zColors>);
```

*n*個、*m*個の値から成る*xVector*および*yVector*に対する*n* x *m*行列の*zGridMatrix*で表される曲線に対し、*zContour*の値で定義された等高線を、*zColors*で定義された色で描きます。

```
// 等高線のテスト

x = (-10 :: 10) / 5;
y = (-12 :: 12) / 5;
grid = J( 21, 25, 0 );
z = [-.75, -.5, -.25, 0, .25, .5, .75];
zcolor = [3, 4, 5, 6, 7, 8, 9];
For( i = 1, i <= 21, i++,
   For( j = 1, j <= 25, j++,
      grid[i, j] = Sin( (x[i]) ^ 2 + (y[j]) ^ 2 )
   );
);
Show( grid );
New Window( "ハット",
   Graph Box( X Scale( -2, 2 ), Y Scale( -2.4, 2.4 ), Contour( x, y, grid, z,
   zcolor ) )
);
```

図12.17 等高線を描く

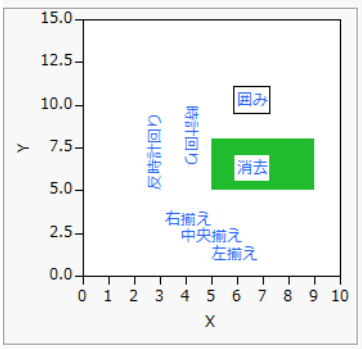


テキストの追加

Textを使うと、任意の位置にテキストを描けます。位置とテキストは、任意の順序でいくつでも指定できます。座標とテキストの引数に加えて、最初の引数に、Center Justified（中央揃え）、Right Justified（右揃え）、Erased（消去）、Boxed（囲み）、Counterclockwise（反時計回り）、Clockwise（時計回り）をオプションで指定できます。Erasedは、グラフ内のテキストを見にくくするものを「消去する」ためのコマンドです。テキストの背後に背景色付きの長方形を作成します。以下の例で、Erasedを指定したテキストが、緑の長方形の上の白いボックス内に、どのように表示されるかを見てください。他の効果は、それぞれの名前が示すとおりです。

```
mytext = New Window( " テキスト ",
   Graph Box(
      framesize( 200, 200 ),
      Y Scale( 0, 15 ),
      X Scale( 0, 10 ),
      Text Size( 9 );
      Text Color( "blue" );
      Text( {5, 1}, " 左揃え " );
      Text( Center Justified, {5, 2}, " 中央揃え " );
      Text( Right Justified, {5, 3}, " 右揃え " );
      Fill Color( 4 );
      Rect( 5, 8, 9, 5, 1 );
      Text( Erased, {6, 6}, " 消去 " );
      Text( Boxed, {6, 10}, " 囲み " );
      Text( Clockwise, {4, 10}, " 時計回り " );
      Text( Counterclockwise, {3, 5}, " 反時計回り " );
   );
);
```

図12.18 グラフボックス内にテキストを描く



text関数は、4つの座標値を引数として、その枠の中で自動的に文字列を改行させることもできます。構文は次のようになります。

```
text( {left, top, right, bottom}, string)
```

色

5つのコマンドで色を制御します。Fill Colorは塗りつぶし領域の色を、Pen Colorは線と点を描くペンの色を、Back Colorはテキストの背景色（前述のErasedにおけるテキスト周りのボックスの色）を、Background Colorはグラフの背景色を、Text Colorは追加するテキストの色をそれぞれ設定します。

塗りつぶしを行った場合、Fill Colorだけが使われて、Pen Colorは使われません。一部の描画パッケージのように両方を同時に使うことはできません。同時に使うには、一方は塗りつぶし、もう一方は塗りつぶさない2つの図形を描きます。色は1つの数値引数、引用符で囲んだ色の名前、RGB値などで指定できます。JMPが提供している標準の色は、番号0～15（0と15は黒）または名前で指定することができます。

表12.1 JMPに用意されている標準の色

番号	名前	試用デモンストレーションスクリプト
0	Black（黒）	colors={"Black", "Gray", "White", "Red", "Green", "Blue", "Orange", "BlueGreen", "Purple", "Yellow", "Cyan", "Magenta", "YellowGreen", "BlueCyan", "Fuchsia"}; ymax=15; mygraph=New Window("JMPの標準色 ", Graph Box(FrameSize(300,300), XScale(0, ymax), yScale(0, ymax+2), for(i=0, i<ymax, i++, If(colors[i + 1] == "White", Fill Color(65); Rect(0, i + 1 + .5, 15, i + 1 - .5, 1)); pen color(colors[i+1]); text color(colors[i+1]); hline(i+1); text({2, i + 1},i); text({5,i+1},colors[i+1]))));
1	Gray（グレー）	
4	White（白）	
3	Red（赤）	
4	Green（緑）	
5	Blue（青）	
6	Orange（オレンジ）	
7	BlueGreen（青緑）	
8	Purple（紫）	
9	Yellow（黄色）	
10	Cyan（シアン）	
11	Magenta（マゼンタ）	
12	YellowGreen（黄緑）	
13	BlueCyan（ブルーシアン）	
14	Fuchsia（赤紫）	

16以上の番号は、同じ色の並びを濃淡の違いで繰り返します。これをデモンストレーションするスクリプトは、「データテーブル」の章の「色とマーカー」（331ページ）にあります。0～84の範囲外にある値は無効です。

表12.2 番号と色のマッピング

番号	結果
16～31	濃い
32～47	淡い
48～63	非常に濃い
64～79	非常に淡い
80～84	グレーの濃淡、淡いから濃いへ

RGB値を使う場合は、赤、緑、青の順でそれぞれの含有率をリストして、各色を指定します。

```
pen color({.38,.84,.67}); // 小鴨色
```

RGB Colorと Color to RGBは、JMPの色番号とRGB（赤、緑、青）値との間の変換を行います。たとえば、JMPの色番号3（赤）のRGB値を調べるには、次のように記述します。

```
Color to RGB(3);
   {0.941176470588235, 0.196078431372549, 0.274509803921569}
```

同様に、HLS Colorと Color to HLSは、JMPの色番号とHLS値（色調、明度、彩度）との間で変換を行います。

Heat Colorは、セルプロットやツリーマップなどで使用されている任意のカラーテーマにおける値に対し、該当するJMPの色番号を戻します。構文は次のとおりです。

```
Heat Color(n,<<"theme")
```

themeメッセージはオプションで、デフォルト値は「青-> グレー ->赤」（Blue to Gray to Red）です。カスタムカラーテーマを含め、任意のカラーテーマを指定できます。匿名のカラーテーマを作ることもできます。例:

```
Heat Color( z, <<{"", {{1, 1, 0}, {0, 0, 1}} })
Heat Color( z, <<{"", {blue, green, yellow} })
```

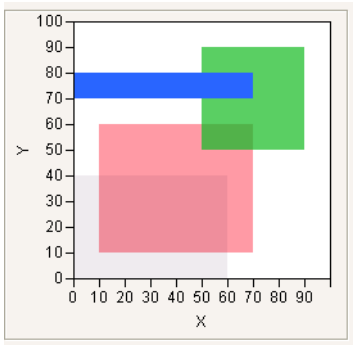
透明度

フレームボックスなどのグラフィックでは、Transparency関数を使って透明度のレベルを指定できます。引数には、0～1の範囲の数値を指定します。値0の場合は透明になり、描画されません。値1の場合は、完全に不透明になります（通常の描画モード）。中間の値を指定すると、すでに描画されているグラフィック上に、半透明の色の層が作成されます。次のスクリプトでは、長方形の透明度を指定しています。

```
New Window( "透明度",
   Graph Box(framesize( 200, 200 ),
      Pen Color( "gray" ); Fill Color( "gray" );
      Transparency( 0.25 );
      Rect( 0, 40, 60, 0, 1 );
```

```
            Pen Color( "red" ); Fill Color( "red" );
            Transparency( 0.5 );
            Rect( 10, 60, 70, 10, 1 );

            Pen Color( "green" ); Fill Color( "green" );
            Transparency( 0.75 );
            Rect( 50, 90, 90, 50, 1 );

            Pen Color( "blue" ); Fill Color( "blue" );
            Transparency( 1 );
            Rect( 0, 80, 70, 70, 1 );
        );
    );
```

図12.19 透明度と長方形



塗りつぶしのパターン

Fill Pattern関数は、サポート対象外になりました。スクリプト内で指定してもエラーにはなりませんが、処理はまったく行われません。

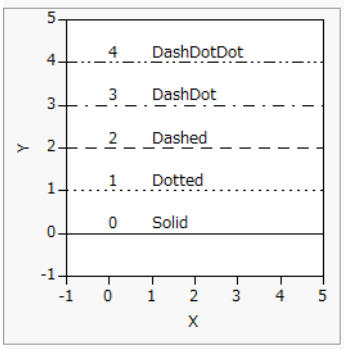
線の種類

Line Style（線種）は、番号（0～4）または名前 (Solid、Dotted、Dashed、DashDot、DashDotDot) で制御できます。

```
linestyles = {"Solid", "Dotted", "Dashed", "DashDot", "DashDotDot"};
New Window( "線種",
    Graph Box(
        FrameSize( 200, 200 ),
        X Scale( -1, 5 ),
        Y Scale( -1, 5 ),
        For( i = 0, i < 5, i++,
            Line Style( i );
```

```
            H Line( i );
            Text( {0, i + .1}, i );
            Text( {1, i + .1}, linestyles[i + 1] );
         );
      );
   );
```

図12.20 線の種類



線の太さを制御するには、Pen Sizeを設定し、線の幅をピクセルで指定します。デフォルトは1で、1ピクセルの線になります。印刷の場合は、Pen Sizeにデフォルトの線の幅を掛けた幅になります。デフォルトの線の幅はプリンタによって異なります。

```
pen size(2); // 2 倍の線幅
```

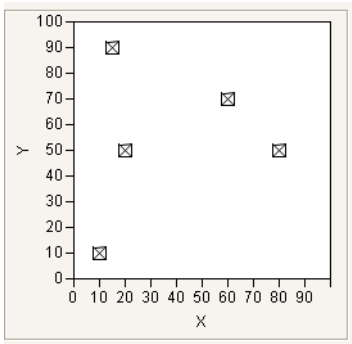
ピクセルを使った描画

ピクセル座標を使って描画することもできます。まず、グラフ座標における基準点をPixel Originで設定し、次に、基準点からの相対的なピクセル座標を、Pixel Move ToコマンドやPixel Line Toコマンドで設定します。Pixelコマンドは、主に、グラフのサイズまたはスケールに左右されないカスタムのマーカーを描くのに使います。マーカーをスクリプトで予め定義しておいて、そのマーカーを任意のグラフから呼び出すこともできます。以下の例では、Functionを使って、引数*x*、*y*をもつPixelコマンドを予め定義しています。

```
ballotBox = Function( {x, y},
   Pixel Origin( x, y );
   Pixel Move To( -5, -5 );
   Pixel Line To( -5, 5 );
   Pixel Line To( 5, -5 );
   Pixel Line To( -5, -5 );
   Pixel Line To( 5, 5 );
   Pixel Line To( -5, 5 );
   Pixel Move To( 5, 5 );
   Pixel Line To( 5, -5 );
);
```

```
New Window( "カスタムマーカー",
   Graph Box(
      framesize( 200, 200 ),
      ballotBox( 10, 10 );
      ballotBox( 15, 90 );
      ballotBox( 20, 50 );
      ballotBox( 80, 50 );
      ballotBox( 60, 70 );
   );
);
```

図12.21 カスタムマーカーを描く



インタラクティブなグラフ

HandleとMouseTrapは、クリックとドラッグに応答するインタラクティブなグラフを作成するための関数です。Handleは、マウスによるドラッグで移動させることができるハンドルのマーカーを追加します。ハンドルが移動されるたびに、その位置の座標を取得し、設定されたスクリプトを実行します。MouseTrapは同様の処理を行いますが、ドラッグ可能なハンドルを使用しないで、クリックした位置の座標を取得します。重要な違いは、Handleは、マウスボタンがハンドル上で押された場合だけ反応しますが、MouseTrapは、マウスボタンがどの位置で押されても反応する点です。

それ以外に、Button Box、Slider Box、Global Boxなどを使って、グラフの外にボタンやスライダなどのコントロールを配置する方法もあります。

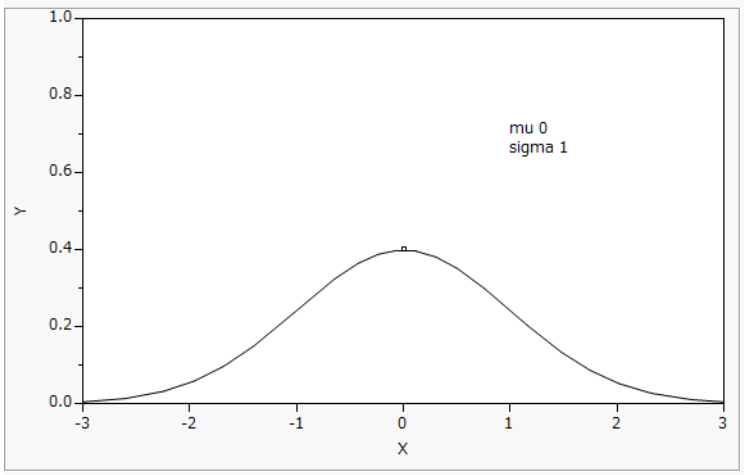
Handle

Handleは、最初の2つの引数の初期値で与えられる座標にハンドルを配置し、引数の初期値を使ってグラフを描きます。その後、そのハンドルは別の位置まで移動できます。初めに指定されたスクリプト（Handle関数の第3引数）は、マウスボタンが押されてハンドルがドラッグされている間、実行されます。2つ目に指定されたスクリプト(Handle関数の第4引数。オプション指定であり、この例では使用していません)は、マウスボタンが放されるたびに実行されます。これらのイベントについては、「MouseTrap」（495ページ）の例を参照してください。

```
// 正規密度
mu = 0;
sigma = 1;
rsqrt2pi = 1 / Sqrt( 2 * Pi() );
New Window( "正規密度",
   Graph Box(
      FrameSize( 500, 300 ),
      X Scale( -3, 3 ),
      Y Scale( 0, 1 ),
      Double Buffer,
      Y Function( Normal Density( (x - mu) / sigma ) / sigma, x );
      Handle(
         mu,
         rsqrt2pi / sigma,
         mu = x;
         sigma = rsqrt2pi / y;
      );
      Text( {1, .7}, "平均 ", mu, {1, .65}, "標準偏差 ", sigma );
   );
);
```

サンプルスクリプトのフォルダに、ベータ分布、ガンマ分布、Weibull分布、および対数正規分布の密度関数を表示するためのスクリプトがあります。正規分布の結果を下に示します。ここではやり方を示しませんが、ご自分で試してみてください。

図12.22 Handleの正規密度の例



エラーを回避するために、必ず、ハンドルの座標の初期値をこの例の1行目のように設定してください。

正規密度の例にあるように、ハンドルの座標に関数を使う場合は、Handleへの引数を調整する必要があります。調整しないと、ハンドルマーカーがグラフの外へ出てしまう場合があります。たとえば、次のような設定が行えます。

```
YFunction(a*x^b,x);
handle(a,b,a=2*x,b=y)
```

マーカーを最初の位置から座標（3,4）の位置までドラッグしたとします。すると、引数*a*に6、*b*に4が設定され、グラフは$y = 6x^4$で再描画され、ハンドルは（6,4）の位置に描かれます。ハンドルなどがグラフの外に出てしまいます。これを修正するには、ハンドルへの最初の引数を、たとえば次のように調整します。

```
handle(a/2,b,a=2*x;b=y)
```

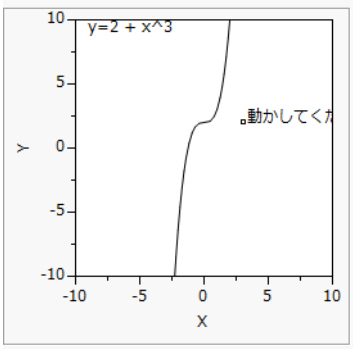
より一般的な状況を考えるために、Handleの引数が a=f(x);b=g(y) と定義されているとします。f(x)=x および g(y)=y の場合は、最初の2つの引数に a、b を指定するだけです。それ以外の場合は、$a = f(x)$をxについて、$b = g(y)$をyについて解いて、適切な引数を求める必要があります。

他の関数を使って、Handleを制限することもできます。以下の例で作成するインタラクティブなグラフは、Round()を使って、指数と切片が整数だけになるようにしています。

```
a = 3; b = 2;
New Window( " 切片とべき乗 ",
   Graph Box(
      FrameSize( 200, 200 ), X Scale( -10, 10 ), Y Scale( -10, 10 ),
      Y Function( Round( b ) + x ^ (Round( a )), x );
      Handle( a, b, a = x; b = y; );
      Text( {a, b}, " 動かしてください " );
      Text( {-9, 9}, "y=", Round( b ), " + x^", Round( a ) );
```

```
   );
);
```

図12.23 Handleの切片とべき乗の例



HandleとForをネストして、複雑なグラフを作成することもできます。

```
a=5; b=5;
New Window(" 指数 ",
   Graph Box(FrameSize(200,200),XScale(-10,10),yScale(-10,10),Double Buffer,
   for(i=0,i<1.5,i+=.2,
      pen color(1+10*i);
      text color(1+10*i);
      YFunction(i*x^round(a),x);
      Handle(a,b,a=x;b=y);
      h=9-10*i;
      text({-9,h},b,"*i*x^",round(a),", i=",i))));
```

また、1つのグラフ内で複数のハンドルを使用できます。

```
amplitude = 1; freq = 1; phase = 0;
NewWindow(" 正弦波 ",
   Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-5,5),Double Buffer,
      YFunction(amplitude*sine(x/freq+phase),x);
      Handle(freq,amplitude,freq=x;amplitude=y);
      Handle(phase,.5,phase=x);
      Text({3, 4}, " 振幅 : ", Round( amplitude, 4 ),
      {3, 3.5}, " 周波数 : ", Round( freq, 4 ),
      {3, 3}, " 位相 : ", Round( phase, 4 ))));
```

MouseTrap

MouseTrapは、マウスをクリックしたときの座標から、グラフ用の引数を取得します。最初のスクリプトは、マウスボタンを押すたびに実行され、2番目のスクリプトはマウスボタンを放すたびに実行され、ハンドルの新しい座標に従って、グラフを動的に更新します。Handleと同様に、MouseTrapの座標に初期値を設定することが重要になります。1つのグラフ内にMouseTrapとHandleの両方を入れる場合は、MouseTrapの前にHandleを指定します。そうすれば、マウスのクリックは、MouseTrapがキャッチする前に、Handleでキャッチされます。

次の例では、MouseTrapとHandleの両方を使って、MouseTrapの座標によって変化する3次元の関数を定めます。Handleの値に基づいて、等高線を1本描きます。

```
x0=0;y0=0;z0=0;
New Window("3 次元関数の等高線 ",
   Graph Box(FrameSize(300,300),XScale(-5,5),yScale(-5,5),DoubleBuffer,
   ContourFunction(exp(-(x-x0)^2)*exp(-(y-y0)^2)*(x-x0),x,y,z0/10);
   handle(-4.5,z0, z0=round(y*10)/10); // z の丸め値をハンドルから取得
   vline(-4.5);text size(9);text(Counterclockwise,{-4.6,-4},
      " ドラッグして等高線の z 値を設定 : z = " || char(z0/10));
   markersize(2);marker(2,{x0,y0});
   mousetrap(x0=x;y0=y); // 基準点をクリックポイントに設定
   text({-4.25,-4.9}," 任意の位置をクリックして関数の中心点を設定 ")));
```

グラフ上で点を視覚的に補間したりするために、MouseTrapを使ってデータテーブルに点を収集することもできます。以下に、（二変量の関係の散布図のような）プロット上でマウスをクリックし、その点をデータテーブルに追加するスクリプトの例を示します。

```
dt = new Table(" データ 1");
Current Data Table(dt);
NewColumn("XX",Numeric);
NewColumn("YY",Numeric);
x=0; y=0;
add point = expr(
   dt<<addRows(1);
   row()=nrow();
   :xx = x;
   :yy = y);
NewWindow(" 点の追加 ",
   Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-5,5),
      for each row(marker({:xx,:yy}));
      MouseTrap({},add point)));
```

この例では、MouseTrapの最初のスクリプトの引数が空であることに注意してください。マウスボタンを押しても、何も起こりません。第2引数のスクリプトadd pointは、マウスボタンを放したときに実行され、データ点が追加されます。つまり、クリックしてドラッグし、マウスボタンを放した場合、データセットに追加される点は、マウスボタンを押した位置の点ではなく、マウスを移動した後の位置の点です。

Drag関数

Handle およびMouseTrap と同様の機能を実行する5つのDrag関数があります。ただし、これらの関数は同時に複数の点を処理します。最初の2つの引数でリストされた行列内の*n*個の座標に関して、次のように動作します。

- Drag Markerは、n個のマーカーを描く。
- Drag Lineは、n個の点を（$n-1$）個の線分でつなぐ。
- Drag Rectは、最初の2つの座標だけを使って他の座標を無視して塗りつぶした長方形を描く。
- Drag Polygonは、n個の頂点を持つ塗りつぶした多角形を描く。
- Drag Textは、座標にテキスト項目を描く。テキスト項目のリストがある場合は、リストのi番目の項目をi番目のx、y座標に描きます。リストの項目が座標のペアより少ない場合は、最後の項目が残りの点に繰り返し使われます。

これらのコマンドの構文は以下のとおりです。

```
dragMarker (xMatrix, yMatrix, dragScript, mouseupScript)
dragLine   (xMatrix, yMatrix, dragScript, mouseupScript)
dragRect   (xMatrix, yMatrix, dragScript, mouseupScript)
dragPolygon(xMatrix, yMatrix, dragScript, mouseupScript)
dragText   (xMatrix, yMatrix, "text", dragScript, mouseupScript)
```

これらすべてには、座標のための左辺値（L-value）の引数が必要です。つまり、行列を値としてもつ変数を引数にとる必要があります。これらの値は、頂点をクリックして新しい位置にドラッグすると変更されます。スクリプトの引数はオプションで、Handleの場合と同様に動作しますが、Handleのように**x**や**y**に座標が含まれることはありません。

Drag演算子は、ユーザが調整でき、その後で調整された値を取得し、そのデータを表示するために使います。前記の、JMPマンを描くスクリプトを思い出してください。Drag Polygonを使うと、編集可能なJMPマンを描くことができます。その時、頂点に対してDrag Markerステートメントを一緒に使えば、ドラッグできる点が見やすくなります。また、Mouse Trapの例と同様に、新しい座標をデータテーブルに保存できます。「:」演算子と「::」演算子によって、同じ名前をもつ行列とデータテーブルの列を明確に区別している点に注意してください。

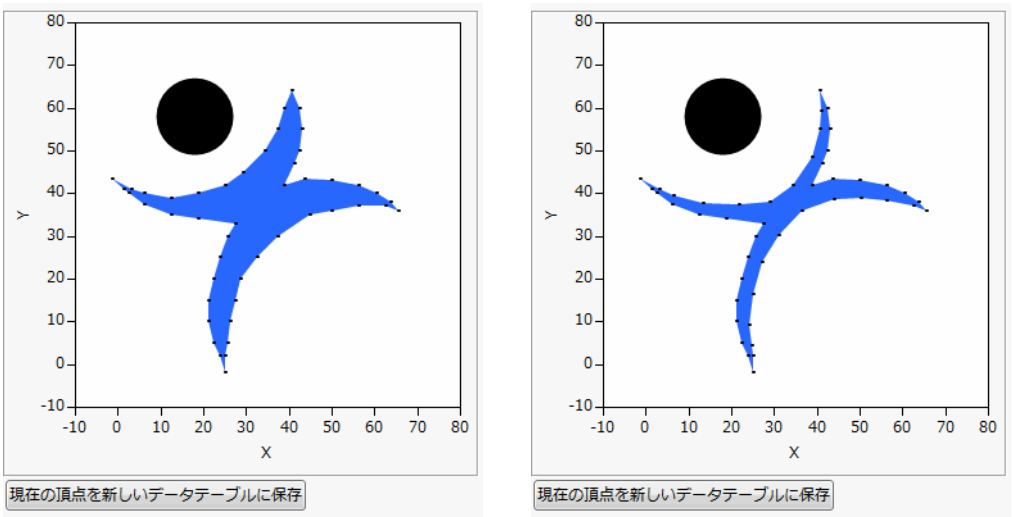
下記のスクリプト例においては、*storepoints*をDrag PolygonまたはDrag Markerの第4引数にする方が簡単ですが、そうすると、ドラッグするたびにデータテーブルが作成されてしまいます。作業が完了した時にだけ、データテーブルを作成したほうが望ましいでしょう。どちらの場合でも、*gCoordX*と*gCoordY*内の値は、ドラッグによって更新されます。

```
::i = 1;
storepoints = Expr(
   mydt = New Table( "My coordinates_" || Char( i ) );
   i++;
   New Column( "GCoordX", Numeric );
   New Column( "GCoordY", Numeric );
   mydt << add rows( N Row( GcoordX ) );
```

```
   :GCoordX << values( ::GcoordX );
   :GCoordY << values( ::GcoordY );
);

 :: GcoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75,
   12.5, 6.25, 2.5,
1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75, 40.625,
   42.5, 43.125,
42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50, 45,
   37.5, 32.5,
28.75, 27.5, 26.25, 25.625, 25];
::GcoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40,
   39, 40, 42,
45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37, 37, 36, 35,
   30, 25, 20, 15,
10, 5, 2];
New Window( "JMP マンを描き直そう！",
   V List Box(
      Graph Box(
         framesize( 300, 300 ),
         X Scale( -10, 80 ),
         Y Scale( -10, 80 ),
         Fill Color( "blue" );
         Drag Polygon( GcoordX, GCoordY );
         Pen Color( "gray" );
         Drag Marker( GcoordX, GCoordY );
         Fill Color( {0, 0, 0} );
         Circle( {18, 58}, 9, "FILL" );
      ),
      Button Box( " 現在の頂点を新しいデータテーブルに保存 ", storepoints )
   );
);
```

JMP マンはもう少しやせた方がいいかもしれません。下の図は、JMP マンの以前の姿といくつかの頂点を慎重にドラッグした後の姿です。この時点でボタンをクリックすると、*storepoints* スクリプトが実行され、JMP マンの新しいスマートな姿の座標が、データテーブルに保存されます。

図12.24 JMPマンを描き直す



この例では、「表示ツリー」の章の「表示ツリーの作成」（399ページ）で説明している次の2つの関数を使っています。

- `Button Box`は、グラフの外にボタンコントロールを作成します。
- `V List Box`は、グラフボックスとボタンボックスを同じグラフウィンドウ内で縦に並べます。

トラブルシューティング

作成したインタラクティブなグラフが期待どおりに動作しない場合は、`Handle`または`MouseTrap`の座標（および、必要な他のグローバル）に初期値を指定していること、および値がグラフに有効であることを確かめてください。

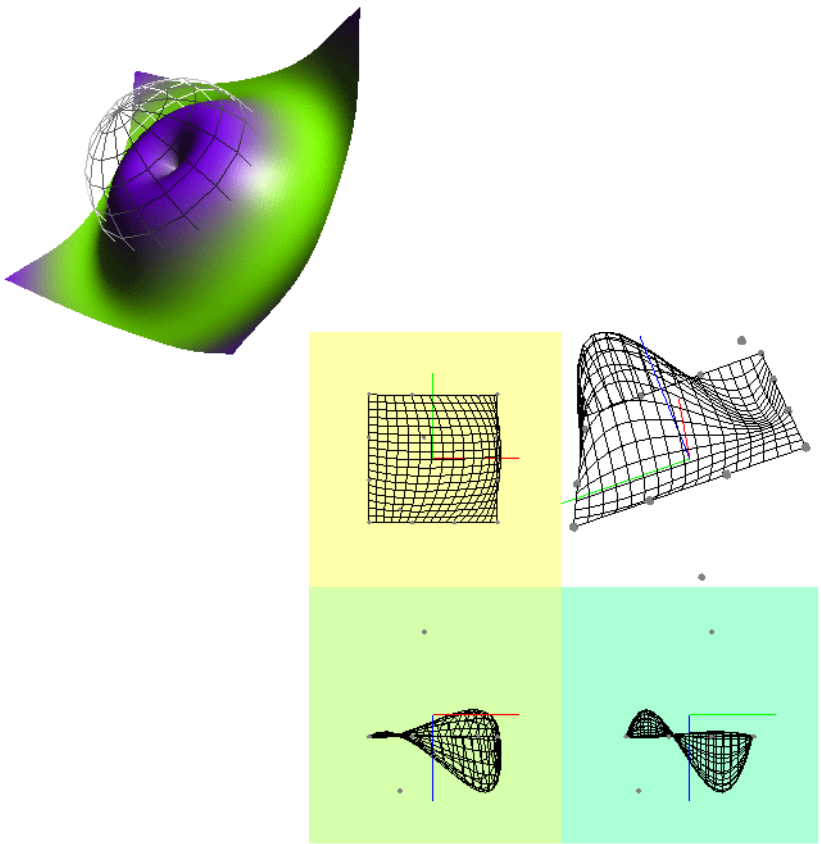
第 13 章

3D シーン

3D シーンのスクリプト

JSL には、OpenGL から派生した 3D シーン（3 次元シーン）を作成するためのコマンドがあります。JSL で提供されている 3D シーンは、OpenGL を完全に実装したものではありませんが、複雑で対話的な 3 次元グラフを作成できます。なお、JMP の「曲面プロット」プラットフォームのプロットは、JSL のシーンコマンド（scene）を使って作成されています。

図 13.1 3D 形状の例



目次

JSL 3D シーンについて 501
JSL 3D シーンボックス 501
表示領域の設定 504
　透視投影シーンのセットアップ 505
　平行投影シーンのセットアップ 506
ビューの変更 507
　Translate コマンド 507
　Rotate コマンド 507
　Look At コマンド 509
　天体球（アークボール） 510
グラフィックの基本要素 511
　基本要素の例 513
　基本要素の外観の制御 515
　Begin および End のその他の用法 520
球、円柱、円盤の描画 520
テキストの描画 522
行列スタックの使用 523
照明と法線 526
　光源の作成 526
　照明モデル 528
　法線ベクトル 528
　シェーディングモデル 529
　材質プロパティ 530
　アルファブレンド 530
　霧 531
　例 531
ベジェ曲線 532
マウスの使用 535
引数 537

JSL 3D シーンについて

JMP の 3D シーン言語は、OpenGL® API のさまざまな機能を拡張、置換、省略して構築されていますが、Silicon Graphics, Inc. による証明またはライセンス許可を受けて実装されているわけではありません。

この章では、3D シーン作成用の JMP の JSL コマンドについて説明します。ただし、OpenGL プログラミングのチュートリアルではありませんので、OpenGL プログラミングに精通していない場合は、補足的な参考書をお読みになることをお勧めします。OpenGL プログラミングに精通している場合でも、この章には特殊な項目が含まれているので、是非お読みください。

JMP の「Sample Scripts」フォルダの「Scene3D」サブフォルダにサンプルファイルが含まれているので、すぐに使用して、使い方をいろいろ工夫できます。スクリプト例の中には、この章で示す例と似ているものもあります。ほとんど完全なアプリケーションと言えるものもあります。

Web サイトの opengl.org には、様々な情報が記載されています。

JMP の 3D シーン言語では、OpenGL API の使用時にユーザが行わなければならないタスクの一部をユーザに代わって実行します。JMP を使うと、テキストの処理が容易になり、組み込みの天体球コントローラを使用できます。モデル表示および投影の行列演算をスタックすることができます。JMP は JMP 自身の表示リスト（ディスプレイリスト）によりシーンを保持し、後でシーンを再生できるようにしています。また、ユーザが作成した JSL コードにコールバックして、シーン内でマウスがどのオブジェクトをポイントしているかを知らせるメカニズムを提供します。ユーザは、余分なプログラミングをする手間を減らせます。現時点では、JMP はテクスチャリングなどの幾つかの機能をサポートしていません。

OpenGL は、Silicon Graphics, Inc. の商標です。

JSL 3D シーンボックス

次のコマンドは、3D シーンのセットアップと設定に必要です。

JMP のすべての表示（詳細については、「表示ツリー」（377 ページ）の章を参照）と同様に、3D シーンもディスプレイボックス（この場合は、シーンボックス）に配置する必要があります。その後で、そのシーンボックスをウィンドウに配置します。したがって、簡単な 3D シーンのスクリプトは、次のようになります。

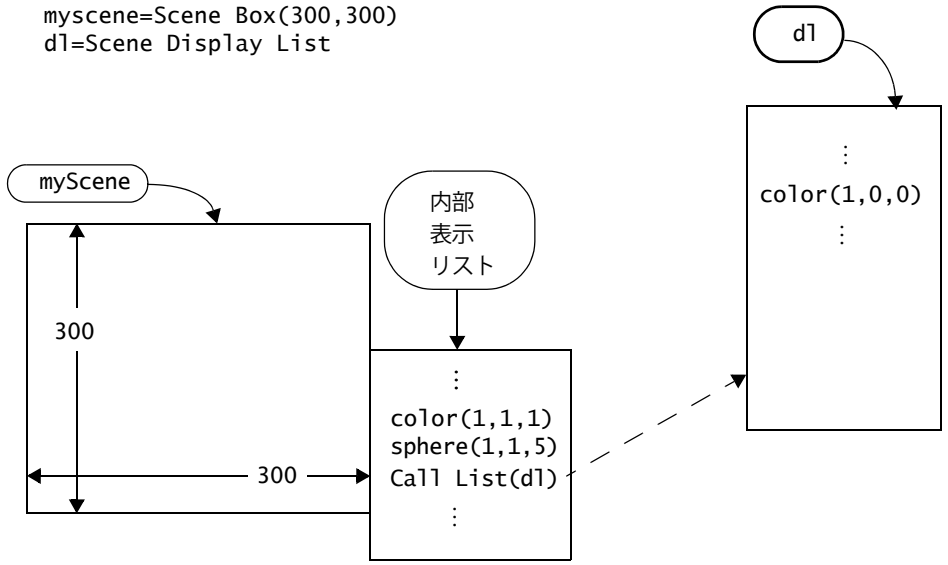
```
myScene=Scene Box(300, 300); //300 x 300 ピクセルのシーンボックスを作成する
...（シーンをセットアップするコマンド）...
New Window ("3D シーン ", myScene); // シーンをウィンドウに描画する
...（シーンを操作するコマンド）
```

シーンの要素を作成するメッセージをシーンに送ることができます。代表的なメッセージとして、視点の変更、物理要素の作成、光源およびテクスチャの操作などがあります。これらのメッセージは表示リスト（ディスプレイリスト）内に保持されます。表示リストは、次のどちらかの方法で操作されます。

- メッセージとしてシーンに送られ、ただちにシーンの内部的な表示リストに追加される。

- メッセージとして、グローバル変数に格納された表示リストに送られ、後でシーンの表示リストにより呼び出される。

図13.2 シーンの作成



次のスクリプトは、シーンの表示リストに直接コマンドを送る例です。各コマンドについては、この章の後半で詳しく説明します。

```
scene = SceneBox( 400, 400 ); // シーンボックスを作成する
New Window( "例 1", scene ); // シーンをウィンドウに配置する
scene << Perspective( 45, 3, 7 ); // カメラを定義する
scene << Translate( 0.0, 0.0, -4.5 ); // (0,0,-4.5) に移動して描画する
scene << color(1,0,0); // テキストの RGB 色を設定する
scene << Text( center, baseline, 0.2, "Hello, World." ); // テキストを追加する
scene << Update; // シーンを更新する
```

最初の2行で、シーンを作成し、ウィンドウに配置します。Perspectiveコマンドは、表示角度と被写界深度を定義します。このコマンドをメッセージとしてシーンに送ると、ただちにシーンの表示リストに追加されます。「Hello World」テキストは原点(0, 0, 0)に描画されるので、Translateコマンドを表示リストに追加し、原点が視野に入るようにカメラを少し後方に移動します。Colorコマンドで色を赤に設定し、テキストを描画し、Updateコマンドでシーンをレンダリングします。これで、コマンドを含む表示リストが描画されます。

図 13.3 Hello World

同様に、表示を作成するコマンドを、グローバル変数に格納されている表示リストに累積させた後で、一度にシーンに送ることができます。グローバル変数を表示リストとして定義するには、Scene Display List関数を使って表示リストを割り当てます。たとえば、グローバルなgreetingを表示リストとして使うには、次のコマンドを実行します。

```
greeting=Scene Display List();
```

この後で、表示コマンドをメッセージとしてgreetingに送ることができます。次の例では、表示リストを使って「Hello World」を描画します。

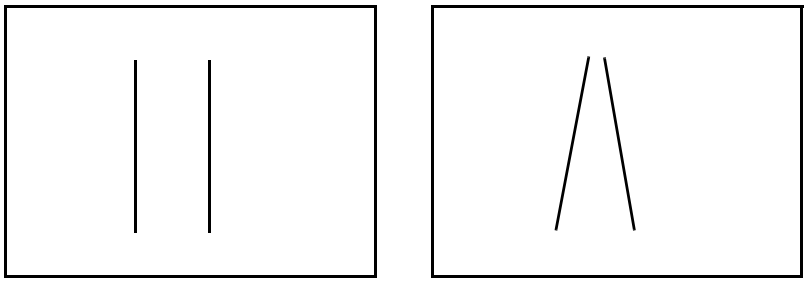
```
// 表示リストを作成してコマンドを送る
greeting = Scene Display List();
greeting << color(1,0,0); // テキストの RGB 色を設定する
greeting << Text( center, baseline, 0.2, "Hello, World." ); // テキストを追加する

// ウィンドウを描画して、保存済みの表示リストを送る
scene = Scene Box( 400, 400 ); // シーンボックスを作成する
New Window( "例 1", scene ); // シーンをウィンドウに配置する
scene << Perspective( 45, 3, 7 ); // カメラを定義する
scene << Translate( 0.0, 0.0, -4.5 ); // (0,0,-4.5) に移動して描画する
scene << Call List(greeting); // 表示リストをシーンに送る
scene << Update; // シーンを更新する
```

どのコマンドが別個に表示リストに移されたか、どのコマンドがシーンに直接適用されたかに注意してください。カメラを操作するコマンド（Perspective およびTranslate）は、シーンに適用されています。オブジェクトを定義するコマンド（Color およびText）は、表示リストに移されました。こうしておけば、その表示リストを何回でも呼び出して、同じオブジェクトを異なる位置に複製できます。

表示領域の設定

3D シーンは、2通りの方法でレンダリングできます。**Orthographic**（平行投影, 正射影）では、どの視点から見ても配置された要素には遠近感がありません。**Perspective**（透視投影）では、視点の位置と連動して遠近感が出るように表示が調整されます。たとえば、線路のような2本の平行線は、平行投影では平行線のままですが、透視投影では、遠くの1点で交わるように表示されます。

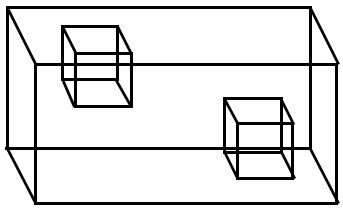
図13.4 平行投影の平行線（左）と透視投影（右）



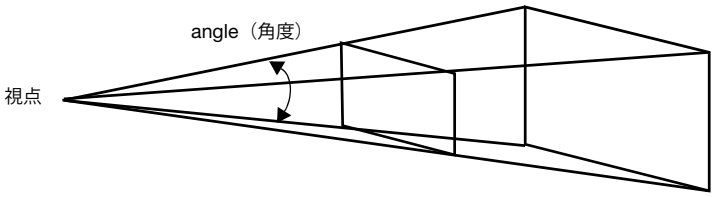
別の例を考えてみましょう。望遠鏡のような筒を真上から見ているとします。平行投影では、筒は厚みのない円として描かれます。透視投影では、円に厚みがあります。つまり、筒の遠端の穴は近端の穴より小さく描かれ、筒の内部が見えます。

したがって、平行投影で表示される領域は直方体であり、透視投影で表示される領域は四角錐台（四角錘の頭部を平面で切り取ったもの）です。

図13.5 投影の比較



平行投影シーン



透視投影シーン

一般に、透視投影は、目やカメラで見る方法を模倣しているので、より現実に近い表示が行われます。平行投影は、建築用のCADプログラムなどのように寸法を保持する必要がある場合に役に立ちます。

透視投影シーンのセットアップ

JSLで透視投影のシーンを構築するには、Perspectiveコマンドを表示に送ります。

```
Perspective (angle, near, far)
```

上の図に示しているように、*angle*は視角、*near*は近い平面までの距離、*far*は遠い平面までの距離です。表示領域を定義するときは、次の2つのことに留意する必要があります。

- 近い（*near*）平面より前方、または遠い（*far*）平面より後方にある要素のように、表示領域の外にある要素は描画されません。それらの要素は表示されません。
- *near* に対する *far* の比は小さくする必要があります。小さい方が、レンダリングエンジンが各要素をシミュレートする時、他の要素の「手前」に描くべき要素を効率よく判定するからです。*near*引数の値はゼロより大きくなければなりません。

「Hello World」を描くスクリプトには、次の行が含まれています。

```
scene << Perspective( 45, 3, 7 ); // カメラを定義する
```

これは、視角が45度、近い平面までの距離は3単位、遠い平面までの距離は7単位であることを定義しています。

視角は、カメラの広角レンズまたは望遠レンズと同じ働きをします。視角が小さければ図が拡大され、大きければ縮小されます。つまり、視角が小さければ、小さい大きさのシーンに3次元空間が写されるので、シーンのなかの要素が大きく見えます。視角が大きければ、大きな大きさのシーンに3次元空間が写されるので、シーンのなかの要素が小さく見えます。したがって、シーンにおける要素の大きさは、`Perspective`関数の*`angle`*引数を使って操作できます。次の図は、角度を45度と90度に設定した透視投影による「hello world」スクリプトの例です。

図13.6 投影法の変更

`Perspective`コマンドの代わりに、`Frustum`コマンドで視野の四角錐台を定義することもできます。

```
Frustum(left, right, bottom, top, near, far);
```

四角錘台における近い方の平面の左下隅と右上隅の(*x, y, z*)座標が、(*left, bottom, near*)と(*right, top, near*)で定義されます。*near*には近いクリップ平面までの距離を、*far*には遠いクリップ平面までの距離を指定します。

平行投影シーンのセットアップ

平行投影のシーンは、透視投影のシーンと同様な方法で指定します。次のコマンドを実行します。

```
Ortho(left, right, bottom, top, near, far)
```

このコマンドは、近い平面の4隅の座標、近い平面までの距離、遠い平面までの距離を指定します。

簡単な2次元環境を処理している場合は、このコマンドで2次元の平行射影シーンをセットアップできます。

```
Ortho2D (left, right, bottom, top)
```

このコマンドは、2次元ビューの4隅を指定します。

ビューの変更

3D シーンを作成する利点の 1 つは、対象をさまざまな角度と位置から簡単に見ることができるということです。Translate と Rotate コマンドを使って、シーンを見る位置を設定できます。

また、ArcBall コマンドを使うと、視角をインタラクティブに変更できます。

Translate コマンド

前述のサンプルスクリプトで、Translate コマンドを実際に確認しました。このコマンドは、シーンの表示基点を設定します。引数は、現在位置からの、*x*、*y*、および z 軸方向への移動量を指定します。

```
Translate (x, y, z)
```

例:

```
Translate( 0.0, 0.0, -2 );
```

これは、原点を z 軸の負の方向に 2 単位移動します。

始めはカメラは原点にありましたが、ここでカメラを z 軸の負の方向に下げたので、カメラが原点を見ることができるようになりました。

Rotate コマンド

Rotate コマンドは、シーンの視る角度を修正する場合に使います。次の形式で指定します。

```
Rotate (degrees, xAxis, yAxis, zAxis)
```

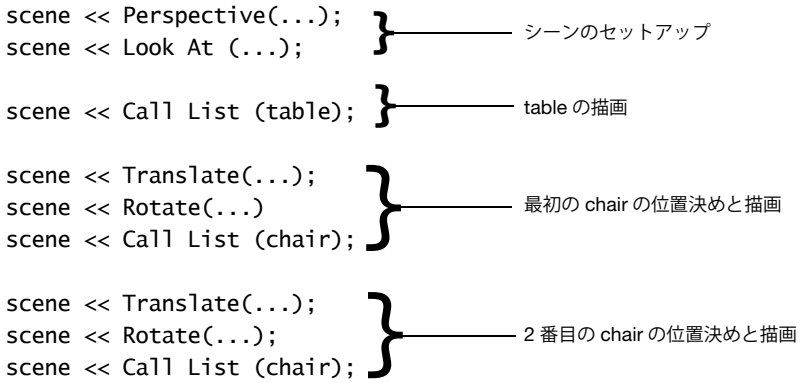
これは、モデルを、ベクトル（*xAxis, yAxis, zAxis*）で指定した軸を中心にして degrees で指定した角度だけ回転します。たとえば、モデルを *x* 軸を中心にして 90 度回転するには、`Rotate( 90, 1, 0, 0 )` を使います。

3 本の軸の値を行列で指定することもできます （たとえば、`Rotate( 90, [1, 0, 0] )`）。

注：JMP の三角関数における角度はラジアン単位ですが、Rotate コマンドでは度単位です。

Translate と Rotate は、オブジェクトを相対的に位置付ける場合にも使われます。一番初めの Translate や Rotate は、後から指定するすべてのオブジェクトのカメラに対する位置を決めるものと考えることができます。後続の Translate コマンドと Rotate コマンドを使って、球、円柱、円盤などのオブジェクトの位置を決め、Call List コマンドや ArcBall コマンドでリストを表示します。たとえば、「table」というリストと「chair」という表示リストがあるとします。シーンは、次のように指定できます。

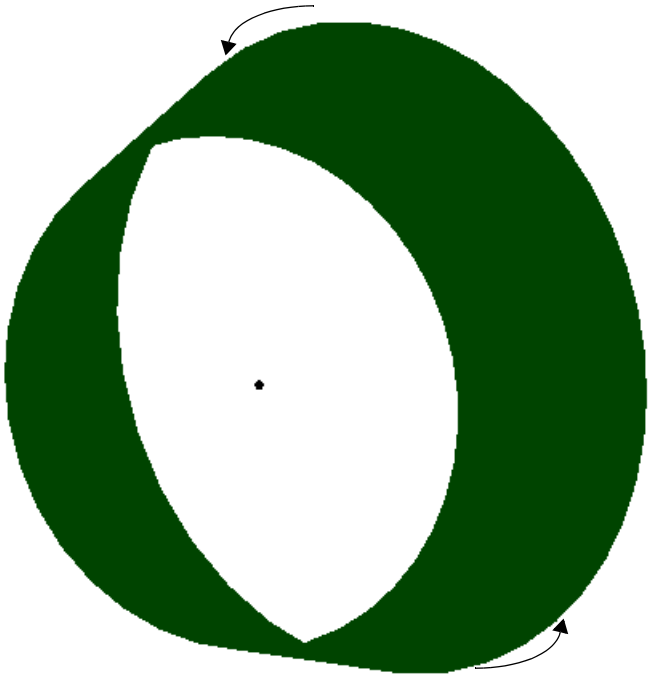
図13.7 TranslateとRotateの使用



次の例では、Rotateコマンドを For ループ内で使って、シーンのカメラアングルを次々に変更します。このスクリプトは、中心点の周りを回転する円柱を描きます。中心点は小さい球で示されます。

```
// OpenGL シーンを保持するシーンボックスを作成する
scene = SceneBox( 600, 600 );

// シーンをウィンドウに配置する
NewWindow( "例 1", scene );

for (i=1, i<360, i++,
   scene << clear;

//  レンズは 45 度、近い距離（near）はカメラから 1 単位、遠い距離（far）は 10 単位
   scene << Perspective( 45, 1, 10 );
   scene << Translate( 0.0, 0.0, -2 );

   scene << Rotate(i,1,0,0);
   scene << Rotate(i*3, 0, 1, 0);
   scene << Rotate (i*3/2, 0, 0, 1);
   scene << Color(0, 1, 0); // 円柱の色は緑
   scene << Cylinder(0.5, 0.5, 0.5, 40,10);
   scene << Color(0, 0, 0); // 球の色は黒
   scene << Sphere(0.01, 10, 5);
   scene << Update;
   Wait(0.01); )
```

図13.8 シリンダの回転



シーンへのメッセージの最後で Update コマンドを使うことに注意してください。このコマンドは、表示される画面と表示リストの現在の状態を一致させるようにJMPに指示します。前の角度と現在の角度がリスト内に混在しないように、先頭でリストをクリアしてから、変更の後でシーンを更新するよう指定します。

Look Atコマンド

Look At コマンドは、カメラの視点を設定する別の方法です。

```
Look At( eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )
```

Look Atコマンドは、カメラを視点（*eye*）座標に配置し、中心（*center*）座標に向けます。*up*ベクトルは、視線上でのカメラの回転方法を指定します。通常、モデルは原点に作成されるので、JMPシーンの始めの方で Look At コマンドまたは Translate コマンドを使ってカメラを原点から離す必要があります。

まず、前のフレームのコマンドのシーンボックスをクリアします。

```
scene<<clear;
```

次に、以下の投影法のいずれかを選択します。

```
scene <<perspective(45,2,10);
scene <<frustum(-.5,.5,-.5,.5,1,10);
scene <<ortho(-2,2,-2,2,1,10);
scene <<ortho2d(-2,2,-2,2);
```

注：投影法をortho2dとした場合は、TranslateまたはLook Atによるカメラ位置の設定は行わないでください。

最後に、TranslateまたはLook Atのいずれかを使用してカメラ位置の設定を行います。

```
scene <<Translate(0.0, 0.0, -4.5);
/* カメラを Z 軸の負の方向に移動し、
   原点 (0,0,0) が視野に入るように移動する */
scene <<LookAt( /* 視点 */3,3,3,    /* 中心点 */0,0,0,    /* アップ */1,0,0 );
/* この指定方法がはるかに簡単 */
```

このような方法でシーンとカメラ位置を設定した後に、モデルを追加していってください。

天体球（アークボール）

マウスの動きに従ってシーンを回転させたい場合があります。JMPの「曲面プロット」プラットフォームは、マウスの動きに従って回転する3Dシーンの例です。

ArcBall（天体球；アークボール）は、3Dシーンの周りに球を設定します。ユーザは、その球の表面をクリックしてドラッグすることにより、シーンを回転させることができます。

CallListコマンドの代わりにArcBallを使って、シーンを天体球（アークボール）に配置します。天体球に張り付けられたシーンは、自動的にマウスのクリック＆ドラッグに対応するようになります。新しいプログラムを作成する必要はありません。ただし、天体球での回転は保存されません（技術的に説明すると、ArcBallは暗黙のPush MatrixとPop Matrixブロックの中にあるので、戻ったときにはその動きは消滅しています。プッシュとポップの詳細については、「行列スタックの使用」（523ページ）を参照してください）。

例として、「基本要素の例」（513ページ）のスクリプトを調べてみましょう。次のような行があります。

```
scene << CallList(shape); // 表示リストをシーンに送る
```

この行を、次のように変更します。

```
scene << ArcBall(shape,2); // 表示リストを天体球に送る
```

このスクリプトは、表示を直径2の天体球に関連付けます。スクリプトを実行してウィンドウが表示されたら、右クリックし、表示されるメニューから［**天体球の表示**］>［**常に**］を選択します。

注：天体球（ArcBall）は、Academic Press発行の『Graphics Gems IV』に掲載された記事（1994年）でShoemakeが使った用語です。

これにより、天体球が常に表示されるように設定されます。シーンを回転するには、天体球をクリック＆ドラッグします。シーンをプラットフォーム、ジャーナル、JSLのどれで表示する場合でも、［**背景色**］、［**ハードウェアアクセラレーションを使用**］、および［**天体球の表示**］の項目があるポップアップメニューが常に使用できます。

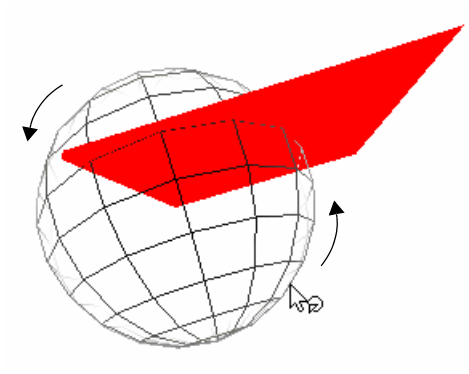
注：天体球を表示しなくても、マウスコマンドは実行されます。ここでは、説明するために表示しているだけです。

Show ArcBall コマンドを使って、JSL で天体球の表示状態を設定することもできます。

```
scene << Show Arcball (state)
```

state には、During Drag、Always、または Never を指定できます。

図 13.9 天体球の表示



グラフィックの基本要素

JSL のすべてのシーンは、少数のグラフィック基本要素から作成されます。これらの基本要素は、複雑なシーンを組み上げるブロックとして機能します。

グラフィックの基本要素はすべて、頂点を指定して実行します。頂点は、単独の点として描かれる場合も、結合されて多角形を構成する場合もあります。基本要素を描くには、基本要素のタイプ、座標、および含まれる頂点のプロパティを指定する必要があります。JSL では、これらは Begin ステートメントと End ステートメントの間に指定します。

```
scene<<Begin(primitive type);
...(頂点とそのプロパティを指定するコマンド)...
scene<<End();
```

頂点の座標を指定するには、vertex コマンドを使います。

```
scene<<Begin(primitive type);
scene<<Vertex(x, y, z);
...
scene<<End();
```

*primitive type*には次のオプションがあります。次の例では、v0、v1などはBeginコマンドとEndコマンドの間に指定されているものとします。

```
scene<<Begin(primitive type);
scene<<Vertex(x0,y0,z0)// 頂点 v0 を指定する
scene<<Vertex(x1,y1,z1)// 頂点 v1 を指定する
...
scene<<Vertex(xn,yn,zn)// 頂点 vn を指定する
scene<<End();
```

表13.1 基本要素の種類

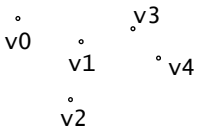
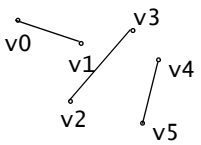
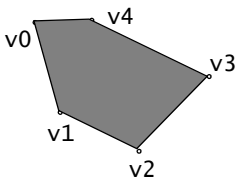
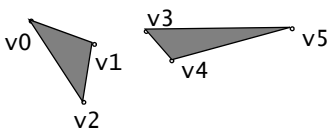
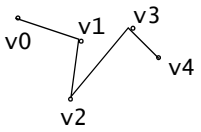
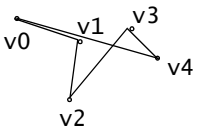
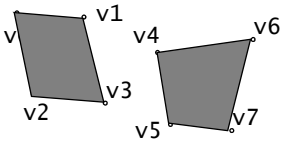
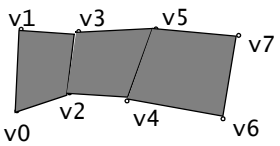
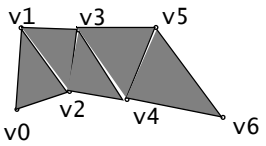
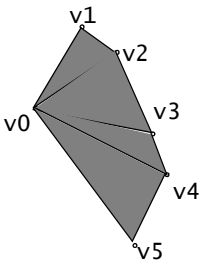
primitive type=POINTS	各頂点に点を描きます。	
primitive type=LINES	複数の線分（連結されていない）を描きます。線分は、v0 と v1、v2 と v3、以下同様に結んで描かれます。n が奇数の場合、最後の頂点は無視されます。	
primitive type=POLYGON	点 v0、...、vn を頂点とする多角形を描きます。3 個以上の頂点がないと、多角形は描かれません。また、指定する多角形は、線が交差しない、凸状の多角形である必要があります。頂点がこれらの条件を満たしていない場合、結果は予期しないものとなります。	
primitive type=TRIANGLES	頂点を v0、v1、v2、次に v3、v4、v5、以下同様に結んで複数の三角形（それぞれ分離している）を描きます。頂点の数が 3 の倍数ではない場合、最後の 1 個または 2 個の頂点は無視されます。	
primitive type=LINE_STRIP	v0 から v1、次に v1 から v2、以下同様に結んで連結する線分を描きます。したがって、n 個の頂点を指定すると n-1 本の線分が描かれます。頂点が 2 個以上ないと何も描かれません。線分を指定する頂点に関する制限はないので、線が交差する場合もあります。	

表 13.1 基本要素の種類（続き）

primitive type=LINE_LOOP	LINE_STRIP とほとんど同じですが、最後の線分は最後の頂点から最初の頂点を結んで描かれ、完全に閉じられます。	
primitive type=QUADS	頂点を v0、v1、v2、v3、次に v4、v5、v6、v7、以下同様に結んで複数の四辺形（4 辺の多角形）を描きます。頂点の数が 4 の倍数でない場合、最後の 1 ～ 3 個の頂点は無視されます。	
primitive type=QUAD_STRIP	頂点を v0、v1、v3、v2、次に v2、v3、v5、v4、その次に v4、v5、v7、v6、以下同様に結んで複数の四辺形（4 辺の多角形）を描きます。頂点の数が少なくとも 4 個ないと何も描かれません。頂点の数が奇数の場合、最後の頂点は無視されます。	
primitive type=TRIANGLE_STRIP	頂点を v0、v1、v2、次に v2、v1、v3（順序に注意）、その次に v2、v3、v4、以下同様に結んで複数の三角形（3 辺の多角形）を描きます。三角形が同じ方向に向かってすべて描かれ、正しい面が形成されるためには、頂点の順序を適切に指定する必要があります。頂点が少なくとも 3 個ないと何も描かれません。	
primitive type=TRIANGLE_FAN	TRIANGLE_STRIP とほとんど同じですが、頂点を v0、v1、v2、次に v0、v2、v3、その次に v0、v3、v4、以下同様に結びます。	

基本要素の例

次の例で、グラフィックの基本要素の使い方を説明します。

```
// 表示リストを作成してコマンドを送る
shape = Scene Display List();
```

```
shape << Color(1,0,0); // テキストの RGB 色を設定する
shape << Begin(POLYGON);
shape << Vertex(0, 0, 0);
shape << Vertex(0, 3, 0);
shape << Vertex(3, 3, 0);
shape << Vertex(5, 2, 0);
shape << Vertex(4, 0, 0);
shape << Vertex(2, -1, 0);
shape << End();

// ウィンドウを描画して、保存済みの表示リストを送る
scene = Scene Box( 400, 400 ); // シーンボックスを作成する
New Window( "基本要素", scene ); // シーンをウィンドウに配置する
scene << Perspective( 90, 3, 7 ); // カメラを定義する
scene << Translate( 0.0, 0.0, -5 ); // (0,0,-5) に移動して描画する
scene << Call List(shape); // 表示リストをシーンに送る
scene << Update; // シーンを更新する
```

このスクリプトの最初のセクションで「shape」という名前の表示リストを作成します。そして、この表示リスト内で、6個の頂点を使って多角形を定義します。

スクリプトの2番目の部分では、シーンボックスと新しいウィンドウを作成しています。そして、Call List関数を使って表示リストをシーンに送っています。

z座標がすべてゼロであること、つまり多角形が1つの平面上にあることを確認してください。多角形が同一平面上にない場合、予期しない結果になります。

スクリプトには、次のような行があります。

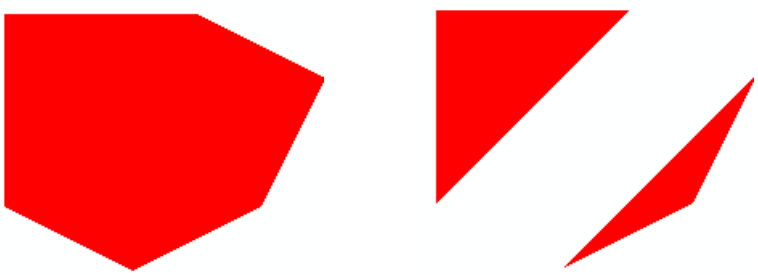
```
shape <<Begin(POLYGON);
```

この行の基本要素を他のタイプに変更してみましょう。たとえば、次のように変更してみましょう。

```
shape <<Begin(TRIANGLES);
```

そうすると、違った図形が描かれます。

図13.10 多角形（左）と三角形（右）



基本要素の外観の制御

JSL には、オブジェクトを描く基本要素の外観を変更するコマンドがあります。線の幅や点描パターン（破線、点線など）を指定することもできます。

サイズと幅

レンダリングされるオブジェクトの点のサイズを設定するには、Point Size コマンドを使います。

```
Point Size (n)
```

n はピクセル数です。このピクセル数は、アンチエイリアシングやハードウェア構成などの他の設定によっては、レンダリングされる実際のピクセル数と異なる場合があります。

線の幅は、Line Width コマンドを使って設定します。

```
Line Width(n)
```

n はピクセル数です。引数 *n* はゼロより大きい必要があり、デフォルトは 1 です。

点描パターン

点描される線を作成するには、Line Stipple コマンドを使います。

```
Line Stipple(factor, pattern)
```

Factor は伸張率です。*Pattern* は、ピクセルをオンまたはオフにする 16 ビットの整数です。効果をオンにするには、Enable(LINE_STIPPLE) を使います。

点描パターンを作成するには、目的の点描パターンを表す 16 ビットのバイナリ数を書き込みます。パターンは右から左に読まれるので、表記がレンダリング方向とは逆に見えてしまうことがあります。バイナリ数を整数に変換し、それを *pattern* 引数として使います。

たとえば、パターン 0000000011111111 の点線を描きたいとします。このバイナリ数は 10 進表記では 255 なので、Line Stipple(1, 255) と指定します。

factor 引数は、各バイナリ桁を 2 倍に引き伸ばして 2 桁にします。この例では、Line Stipple(2, 255) は 0000000000000000111111111111111 になります。

次の例のスクリプトは、それぞれ幅（Line Width コマンド）と点描パターンが異なる 3 本の線を描きます。

```
// OpenGL シーンを保持するシーンボックスを作成する
scene = SceneBox( 200, 200 );

// シーンをウィンドウに配置する
New Window( "Stipples", scene );

scene << Ortho(-2,2,-2,2,-1,1);

scene << color(0,0,0);
   // テキストの RGB 色を設定する
```

```
scene << Enable(LINE_STIPPLE);

scene << Line Width(2);
scene << Line Stipple(1, 255);

scene << Begin(LINES);
scene << Vertex(-2, -1, 0);
scene << Vertex(2, -1, 0);
scene << End();

scene << Line Width(4);
scene << Line Stipple(1, 32767);

scene << Begin(LINES);
scene << Vertex(-2, 0, 0);
scene << Vertex(2, 0, 0);
scene << End();

scene << Line Width(6);
scene << Line Stipple(3, 51);

scene << Begin(LINES);
scene << Vertex(-2, 1, 0);
scene << Vertex(2, 1, 0);
scene << End();
scene << Update;
```

図13.11 点描

注：点描パターンは、回転するモデル上に「這う」ように描かれます。これは、点描パターンがモデルの単位ではなく画面のピクセルで指定され、モデル内の線はモデルの単位に変化がなくても画面上で長さが変化するからです。

塗りつぶしパターン

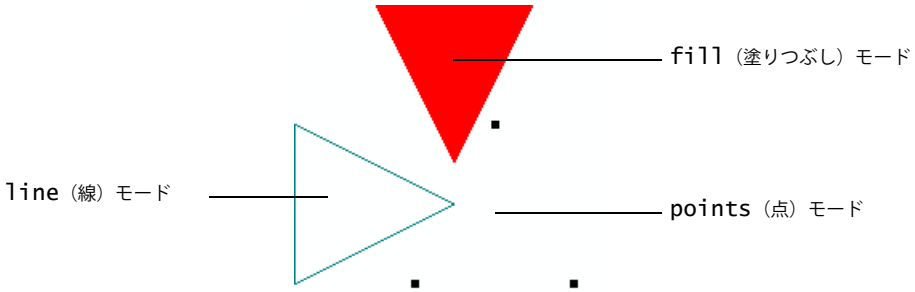
多角形は、前面と背面の両方でレンダリングされ、どちらの側の描画モードもカスタマイズできます。多角形の背面と前面で、異なった描画方法を指定することができます。

多角形の描画モードを設定するには、Polygon Mode コマンドを使います。

```
Polygon Mode (face, mode)
```

face には、FRONT、BACK、または FRONT_AND_BACK を、*mode* には、POINT（点）、LINE（線）、または FILL（塗りつぶし）を指定できます。

図 13.12 点、線、塗りつぶし



次表のスクリプトは、三角形を定義する表示リストを作成します。表示リストにおいて、Translate、Rotate、および Color と組み合わせて 3 回使用し、3 角形を 3 つの位置に描きます。また、Polygon Mode コマンドによって、各三角形の描画モードを変更しています。塗りつぶし（FILL）モードはデフォルトなので、明示的に呼び出していないことに注意してください。

次表には、スクリプトの解説も記載しています。Translate と Rotate コマンドによる操作を、どのように表示リストに累積させているかを説明しています。

表 13.2 Translate コマンドと Rotate コマンド

上記のスクリプトのコード	説明
`shape = Scene Display List();`	表示リストを作成する。
`shape << Begin(TRIANGLES);` `shape << Vertex(0, 0, 0);` `shape << Vertex(-1, 2, 0);` `shape << Vertex(1, 2, 0);` `shape << End();`	三角形の頂点を保持する「shape」という名前の表示リストを作成する。この例は 2 次元上に描画するので、頂点の z 座標はすべてゼロに設定します。
`scene = Scene Box( 200, 200 );` `New Window( "塗りつぶしモード", scene );` `scene << Ortho2d(-2,2,-2,2);`	シーンをディスプレイボックスに配置し、新しいウィンドウを作成する。

表 13.2 Translate コマンドと Rotate コマンド（続き）

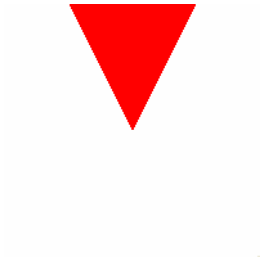
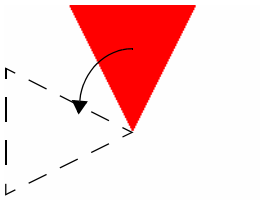
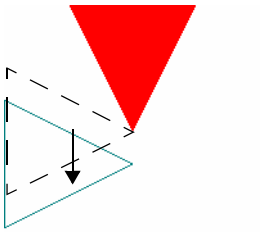
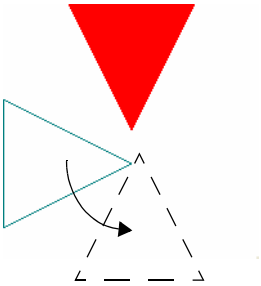
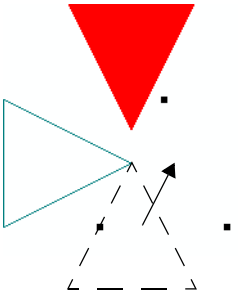
上記のスクリプトのコード	説明
scene << Color(1,0,0); scene << Call List(shape); scene << Update; // シーンを更新して三角形を確認する	最初の三角形を赤で描く。 
scene << Rotate (90, 0, 0, 1); scene << Translate (-0.5, 0, 0); scene << Color(0, 0.5, 0.5); scene << Polygon Mode(FRONT_AND_BACK, LINE); scene << Call List(shape); scene << Update; // シーンを更新して三角形を確認する	2つ目の三角形を緑がかった青で描く。まず三角形を回転します。  それから、移動します。 

表 13.2 Translate コマンドと Rotate コマンド（続き）

上記のスクリプトのコード	説明
scene << Rotate (90, 0, 0, 1); scene << Translate (-0.5, -1, 0); scene << Color(0, 0, 0); scene << Point Size(5); // 見やすいように大きい点を指定する scene << Polygon Mode(FRONT_AND_BACK, POINT); scene << Call List(shape); scene << Update; // シーンを更新して三角形を確認する	3つ目の三角形を黒の点で描く。まず回転します。  それから、移動して、最後の図を描きます。 

塗りつぶした多角形に対して、境界線を描くには、一度、塗りつぶした多角形を描いた後、それに線を重ねて描く必要があります。ただし、レンダリング方法によっては、思ったように線が描画されないことがあります。Polygon Offset コマンドを使うと、この「縫い合わせ」問題を解決できます。

```
Polygon Offset (factor, units)
```

オフセットを有効にするには、目的のモードに合わせて、Enable(POLYGON_OFFSET_FILL)、Enable(POLYGON_OFFSET_LINE)、または Enable(POLYGON_OFFSET_POINT) を使います。実際のオフセット値は、*m**(*factor*)+*r**(*units*) で計算されます。*m* は多角形の深度の最大勾配、*r* はウィンドウ座標の深度値で解像可能な差を生成するのに必要な最小値です。必要に応じて、Polygon Offset(1,1) から始めてください。

Polygon Offset は、「曲面プロット」プラットフォームでも内部的に使われています。曲面とメッシュまたは曲面と等高線を重ねて表示するときに Polygon Offset を使っています。なぜなら、線を視点の方に近づけるか、曲面を視点から遠ざけるかしないと、曲面によって線の一部が隠されてしまうからです。

BeginおよびEndのその他の用法

通常、beginとendステートメントの間には頂点を指定しますが、この他にも有効なコマンドがあります。次のコマンドについては、この章の別の節で説明しています。

- Vertexは、頂点をリストに追加します。
- Colorは、現在の色を変更します。
- Normalは、法線ベクトルの座標を設定します。
- Edge Flagは、エッジの描画を制御します。
- Materialは、材質プロパティを設定します。
- Eval CoordとEval Pointは、座標を生成します。
- Call Listは、表示リストを実行します。

球、円柱、円盤の描画

球、円柱、円盤をすばやくレンダリングできるように、いくつかの定義済みコマンドが用意されています。これらのコマンドは、使いやすいだけでなく、特殊な照明プロパティ（法線）が予め組み込まれています。

作図

円柱、円盤、扇形、および球を作図するには、次のコマンドを使います。

円柱

```
Cylinder( baseRadius, topRadius, height, slices, stacks )
```

*baseRadius*は円柱の底面の半径、*topRadius*は円柱の上面の半径です。*height*は円柱の高さです。

*Slices*は円の精度を指定するもので、円状に近くするためには10以上の値を指定します。QuadricNormals(Smooth)を使うと、外観が良くなります。

*Stacks*には、光の反射に使用できる頂点の数を設定します。*Stacks*の値を大きくすると、それだけ「ホットスポット」が正確になります。

円盤

次のコマンドは、中央に*innerRadius*で指定された穴がある非常に薄い円盤を描きます。

```
Disk( innerRadius, outerRadius, slices, loops )
```

Cylinderコマンドと同様に、*slices*は曲線の精度を制御し、*loops*は光の反射の精度を上げるために頂点数を増やします。

```
Partial Disk( innerRadius, outerRadius, slices, loops, startAngle, sweepAngle )
```

Partial Disk コマンドは、Disk コマンドと同様に動作しますが、円盤の一片が取り除かれます。*startAngle* と *sweepAngle* を使って、表示する円盤部分を指定します。

球

次のコマンドは、指定された半径（*radius*）の球を描きます。

```
Sphere( radius, slices, stacks )
```

slices は経度、*stacks* は緯度と考えることができます。それぞれ 10 前後を指定すると、きれいな球が描けます。

照明

カスタマイズされた曲面の場合とは異なり、球、円盤、および円柱については、法線ベクトルの特別な計算は必要ありません。ただし、次のコマンドを使って、自動的に生成される照明をカスタマイズできます。

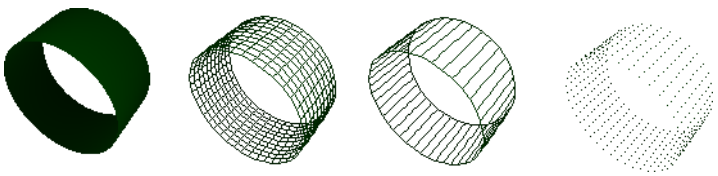
Quadric Normals(*mode*) は、自動的に生成される法線の種類を指定します。*mode* 引数には、None、Flat、または Smooth を指定できます。Flat を指定すると、表面は小平面に分割されます。Smooth を指定すると、各頂点の法線は近接するポリゴンの平均になります。

Quadric Orientation(*mode*) は、法線ベクトルの向きを指定します。*mode* 引数には、Inside または Outside を指定できます。

Quadric Draw Style(*mode*) は、描画モードを指定します。*mode* 引数には、Fill、Line、Silhouette、または Point を指定できます。

JMP は、Quadratic Normals、Quadratic Orientation、および Quadratic Draw Style に設定された値を、それらの設定後に生成された円柱、円盤、球に適用します。

図 13.13 描画の種類



注： 他の OpenGL マニュアルでは、いくつかの 2 次曲面オブジェクトが言及されています。JMP では 1 つだけで、常にそれを使用します。

テキストの描画

前述の「Hello World」の例で説明したように、テキストをシーンに追加するにはTextコマンドを使います。

```
Text( horz, vert, size, string, <billboard>)
```

- *horz*には、Left（左揃え）、Center（中央揃え）、またはRight（右揃え）を指定できます。
- *vert*には、Top（上揃え）、Middle（中間揃え）、Baseline（基準線揃え）、Bottom（下揃え）を指定できます。
- *size*は、モデル座標での大文字Mの高さを指定します。
- *string*は、描画するテキストです。
- *billboard*はオプションの引数で、テキストを回転させます。このオプションを指定すると、テキストは常にユーザに向かって表示されます。

フォントは、常にJMPのテキストフォントが使われます。［環境設定］メニューからテキストのフォントを変更できます。ただし、JMPはシーン用のフォントをキャッシュするので、JMPを再起動しないと、変更は有効になりません。

注：標準のOpenGL定義にはテキストが含まれません。

Text と Rotate および Translate の連動使用

次の例では、TextコマンドをTranslateおよびRotateコマンドと連動させて使っています。

```
/* OpenGL シーンを保持するシーンボックスを作成する */
scene = SceneBox( 600, 600 );

/* シーンをウィンドウに配置する */
NewWindow( "例 2", scene );

scene << Perspective( 45, 3, 7 );
   /* 「レンズ」は 45 度、近い距離（near）はカメラから 3 単位、遠い距離（far）は 7 単位 */
scene << Translate( 0.0, 0.0, -4.5 );
   /* カメラで原点 (0,0,0) が見えるように移動する */
scene << Rotate( 30, 0, 1, 0 );
   /* 最初のテキストを Y 軸（画面の縦方向）を中心に回転させる */
scene << Color( 1, 0, 0 );
   /* 赤色 */
scene << Text( center, baseline, .2, "最初の赤色の文字列" );
scene << Translate( 0.0, 0.0, -2.0 );
   /* 次のテキストをカメラからもっと遠ざける */
scene << Rotate( 30, 0, 1, 0 );
   /* 2 番目のテキストを Y 軸（画面の縦方向）を中心に回転させる */
scene << Color( 0, 1, 0 );
   /* 緑色 */
```

```
scene << Text( center, baseline, .2, "2 番目の緑色の文字列 " );
scene << Update;
   /* 現在の表示リストを使って、ウィンドウ内のディスプレイボックスを更新する */
```

図 13.14 テキスト文字列の回転と移動

緑色の文字列の一部は、遠い方のクリップ平面を超えて後方に位置しています。Perspective の 7 を 10 に変更すると、文字列全体が表示されます。

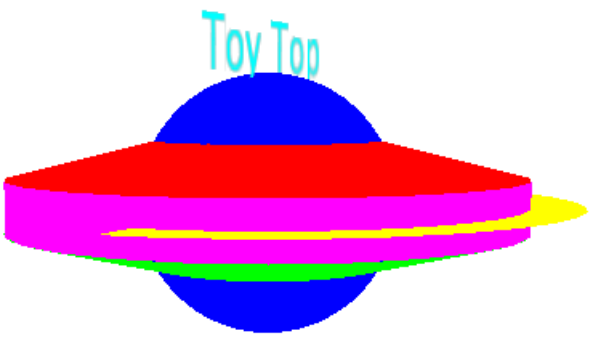
行列スタックの使用

JMP の 3D シーンでは、行列スタックを使って、現在の変換を継続してトラッキングします。スタックは、単位行列に初期化されます。そして、移動、回転、または拡大や縮小のコマンドが指定されるたびに、スタックの最上部の行列が変更されます。

注：多くの OpenGL 実装とは異なり、JMP では転置行列は使いません。

次のプログラム例では、Push Matrix と Pop Matrix を使って「toy top」の各部分の位置を決めています。各部分を指定した後には、原点に戻っています。この方法を使うと、Translate コマンドを 2 回使って元に戻すよりも速く処理できます。

図 13.15 行列スタックを使った描画



```
toyTop = SceneDisplayList();
toyTop<<PushMatrix;
   toyTop<<Translate(0,0,.1);
   toyTop<<Color(1,0,0); // 赤色
   toyTop<<Cylinder(1,.2,.2,25,5);
      /* baseRadius、topRadius、height、slices、stacks を指定する */
toyTop<<PopMatrix;
toyTop<<PushMatrix;
   toyTop<<Translate(0,0,-.1);
   toyTop<<Rotate(180,1,0,0);
   toyTop<<Color(0,1,0); // 緑色
   toyTop<<Cylinder(1,.2,.2,25,5);
toyTop<<PopMatrix;

toyTop<<Color(0,0,1); // 青色
toyTop<<Sphere(.5,30,30);
   /* 半径、スライス数、積み重ね数を指定する */

toyTop<<Color(1,1,0); // 黄色
toyTop<<PartialDisk(1,1.2,25,2,0,270);
   /* 内側半径、外側半径、スライス数、リング数、開始角度、進行角度を指定する */

toyTop<<PushMatrix;
   toyTop<<Translate(0,0,-.1);
   toyTop<<Color(1,0,1); // マゼンタ
   toyTop<<Cylinder(1,1,.2,25,3);
   /* baseRadius、topRadius、height、slices、stacks を指定する */
toyTop<<PopMatrix;

toyTop<<PushMatrix;
   toyTop<<Rotate(90,1,0,0);
   toyTop<<Translate(0,.5,0);
   toyTop<<Color(0,1,1); // シアン
   toyTop<<Text(center,baseline,.2,"Toy Top");
toyTop<<PopMatrix;

/* OpenGL シーンを保持するシーンボックスを作成する */
scene = SceneBox( 600, 600 );

/* シーンをウィンドウに配置する */
NewWindow( "例 3", scene );
scene << Perspective( 45, 3, 7 );
scene << Translate( 0.0, 0.0, -4.5 );
scene << Rotate( -85, 1, 0, 0 );
scene << Rotate( 65, 0, 0, 1 );
scene << CallList( toyTop );
```

```
/* ディスプレイボックスを更新する */
scene << Update;
```

スタック上の現在の行列を変更したい場合は、Load Matrix コマンドを使います。

```
Load Matrix(m)
```

m は、現在の行列スタックにロードされる 4 行 4 列の JMP 行列です。

同様なコマンドに Mult Matrix があります。

```
Mult Matrix(m)
```

Mult Matrix コマンドを実行すると、現在の行列スタックの最上部の行列に *m* が乗算されます。

次に、簡単なコマンドを実行する行列の例を示します。

移動

```
[1  0  0  x
0  1  0  y
0  0  1  z
0  0  0  1]
```

次の回転用行列に対する説明において、c は cos(a)、s は sin(a) です（ここで、a は回転の角度です）。

x 軸を中心にして回転

```
[1  0  0  0
0  c -s  0
0 -s  c  0
0  0  0  1]
```

y 軸を中心にして回転

```
[c  0  s  0
0  1  0  0
-s  0  c  0
0  0  0  1]
```

z 軸を中心にして回転

```
[c -s  0  0
s  c  0  0
0  0  1  0
0  0  0  1]
```

例として、表示リストを移動して回転する 2 通りの方法を示します。初めの例は行列を用いており、2 番目の例は Translate と Rotate を用いています。初めの例は、2 番目の例と反対の方向に移動します。

```
// 行列を使った方法
gl<<Push Matrix;
```

```
xt = identity(4); // これを .75 だけ左方向に移動する
xt[1,4]=-.75;
xr = Identity(4); // これを回転する。cos は度ではなくラジアンで指定する
xr[2,2]=cos(3.14159*a/180);
xr[2,3]=-sin(3.14159*a/180);
xr[3,2]=sin(3.14159*a/180);
xr[3,3]=cos(3.14159*a/180);
yr = Identity(4);
yr[1,1]=cos(3.14159*a/180);
yr[1,3]=sin(3.14159*a/180);
yr[3,1]=-sin(3.14159*a/180);
yr[3,3]=cos(3.14159*a/180);
zr = identity(4);
zr[1,1]=cos(3.14159*a/180);
zr[1,2]=-sin(3.14159*a/180);
zr[2,1]=sin(3.14159*a/180);
zr[2,2]=cos(3.14159*a/180);
gl<<Mult Matrix(xt*xr*yr*zr); // 行列では乗算の順序が重要
gl<<arcball(dl,1);
gl<<Pop Matrix;

//////////////// 関数を使った方法
gl<<Push Matrix;
gl<<Translate(.75,0,0); // これを .75 だけ右方向に移動する
gl<<Rotate(a,1,0,0); // これを度で回転する
gl<<Rotate(a,0,1,0); // ここでは演算子の順序も重要
gl<<Rotate(a,0,0,1);
gl<<Arcball(dl,1);
gl<<Pop Matrix;
```

行列は、表示リストの描画中にだけ保持されているので、以前の変換行列を遡って読み取ることはできません。以前の変換行列を知る必要がある場合には、JSL 変数としてその行列を保持しておき、`Load Matrix` を使ってその行列をスタック上に置きます。

照明と法線

次の方法を使って、図形に、照明、材質、法線ベクトルを追加できます。これらの方法を使うと、モデルに明暗を付けて形状を明確にすることができます。

光源の作成

光源には、色、位置、および方向を指定します。JSL では、`Light` コマンドで最大 8 つ（0 ～ 7）の光源を定義できます。*n* は光源の番号です。

```
Light( n, argument, value, ... value )
```

注：各光源をオンにするには、`Enable (Lighting)` と `Enable (lightn)` コマンドを使います。*n* は光源の番号です。次に、`Light(n, POSITION, x, y, z)` コマンドで、光源をシーン内に移動します。

表 13.3 に、指定できる引数（*argument*）の値を示します。各引数のデフォルト値も示しています。

表 13.3 Light の引数とデフォルト値

引数	デフォルト値	意味
AMBIENT	(0, 0, 0, 1)	環境光（周囲光）の RGBA 輝度
DIFFUSE	(1, 1, 1, 1)	拡散光の RGBA 輝度
SPECULAR	(1, 1, 1, 1)	鏡面光の RGBA 輝度
POSITION	(0, 0, 1, 0)	(*x*, *y*, *z*, *w*) の位置
SPOT_DIRECTION	(0, 0, -1)	スポットライトの (*x*, *y*, *z*) 方向
SPOT_EXPONENT	0	スポットライト指数
SPOT_CUTOFF	180	スポットライトの遮断角度
CONSTANT_ATTENUATION	1	一定減衰率
LINEAR_ATTENUATION	0	線形減衰率
QUADRATIC_ATTENUATION	0	2 次減衰率

注：この表の DIFFUSE と SPECULAR のデフォルト値は Light 0 だけに適用されます。その他の光源については、この 2 つの引数のデフォルト値は (0, 0, 0, 1) です。

最初の 3 つの引数（AMBIENT、DIFFUSE、SPECULAR）は、光源に色を付ける場合に使います。DIFFUSE（拡散）は、光源の物理的な色に最も近く関連付けられる引数です。AMBIENT（環境）は、光がもつ背景光としての特徴に対応したパラメータです。SPECULAR（鏡面）は、表面に反射する光に関連したパラメータです。

POSITION 引数を使って、光源の位置を指定します。4 番目の座標（*w*）に 0 以外の値を指定すると、光源はオブジェクト座標と同じ座標に配置されます。

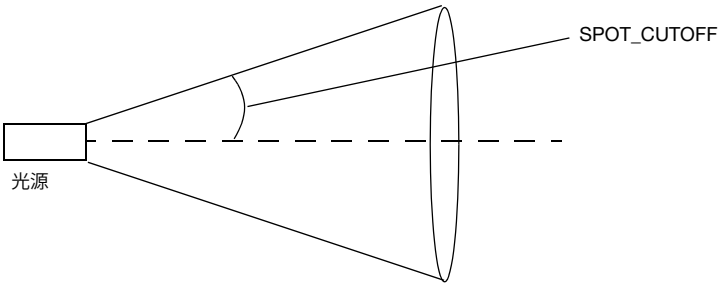
現実世界の光は、光源から離れるに従って暗くなります。ただし、指向性の光に関しては、光源は無限に遠くにあるので、光の輝度の減衰を距離の関数で表すのは意味がありません。JSL では、光源の輝度に、次の減衰率を掛けて光を減衰させます。

$$\text{減衰率} = \frac{1}{c + ld + qd^2}$$

c は CONSTANT_ATTENUATION、*l* は LINEAR_ATTENUATION、*q* は QUADRATIC_ATTENUATION です。

スポットライトを作成するには、光源の形を円錐形にします。次に示すように、SPOT_CUTOFF引数を使って円錐形の側面を定義します。

図13.16 スポットライト



遮断角のほかに、円錐形内の照射の輝度と方向も制御できます。SPOT_DIRECTIONは、スポットライトを向ける方向を指定し、SPOT_EXPONENTは、集光方法に影響します。

照明モデル

照明モデルは、Light Modelコマンドで指定します。

```
Light Model( argument, value,...,value )
```

照明モデルでは、照明の3つの属性を指定します。

- 広域環境光の輝度
- 視点が局所または無限の遠方のどちらにあるか
- 照明計算をオブジェクトの前面と背面で別々に実行するか

表13.3（527ページ）に、Light Modelコマンドの有効な引数を示します。

表13.4 Light Modelの引数とデフォルト値

引数	デフォルト値	意味
LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1)	シーン全体の環境光（周囲光）のRGBA 輝度
LIGHT_MODEL_LOCAL_VIEWER	0（偽）	正反射角の計算方法
LIGHT_MODEL_TWO_SIDE	0（偽）	ゼロ以外の値は2側面の照明を意味する

法線ベクトル

法線ベクトルは、物体の表面に対して垂直な方向を向いています。平面の場合、法線はすべて同じです。より複雑な曲面の場合は、法線も複雑になります。JSLを使うと、各頂点の法線ベクトルを指定できます。これらの法線は空間での曲面の方向を指定するもので、照明計算に不可欠です。法線の精度が高ければ、それだけ照明も正確になります。

法線ベクトルは、頂点に対して直角な、長さが1のベクトルです。通常、頂点は複数の多角形（ポリゴン）で共有されており、かつ、滑らかなシェーディング効果が望まれるので、頂点での垂線は多角形の標準の（加重）平均で計算されます。2面のシェーディングが可能でない限り、多角形の外側の面だけに照明が当てられるので、多角形の「外方向」の法線を計算することが重要です。多角形をスケーリングすると、法線の長さが1にならず、照明が間違ったものになります。

法線ベクトルは、曲面が作成されるときに設定され、Normal コマンドで指定できます。シーンが描かれるたびに法線が1に再正規化されるようにするには、Enable(NORMALIZE) コマンドを使います。

シェーディングモデル

多角形（ポリゴン）のシェーディングモデルは、Shade Model コマンドを使って設定します。

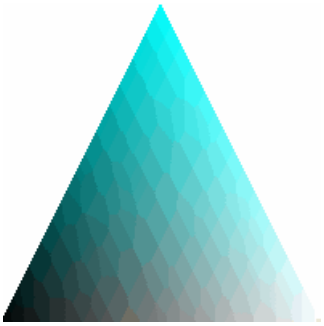
```
Shade Model (mode)
```

mode には、SMOOTH（デフォルト）または FLAT を指定できます。SMOOTH シェーディングは、頂点間で基本要素の色が滑らかになるように中間色を補間します。FLAT モードは、1つの頂点の色を基本要素全体に適用します。

次のスクリプトは、三角形の各頂点で色を変更します。SMOOTH シェーディングモデルは、内部の色を自動的に補間します。

```
// OpenGL シーンを保持するシーンボックスを作成する
scene = SceneBox( 200, 200 );

// シーンをウィンドウに配置する
NewWindow( "シェーディングモデル", scene );
scene << clear;
scene << Ortho2D (-1,1,-1,1);

scene << Shade Model(SMOOTH);
scene << Polygon Mode (FRONT_AND_BACK, FILL);
scene << Begin(TRIANGLES);
scene << color(0, 0, 0); // 黒色
scene << Vertex(-1, -1, 0);
scene << Color(0, 1, 1);// シアン
scene << Vertex(0, 1, 0);
scene << Color (1, 1, 1);// 白色
scene << Vertex(1, -1, 0);
scene << End();

scene << Update;
```

図13.17 シェーディング



材質プロパティ

曲面の材質プロパティを設定するには、Materialコマンドを使います。

```
Material( face, argument, value,...value )
```

*face*には、Front、Back、またはFront_and_backを指定できます （材質プロパティは、多角形の前面と背面に対して別々に設定できます）。

表13.5に、Materialの引数とデフォルト値を示します。

表13.5 Materialの引数とデフォルト値

引数	デフォルト値	意味
AMBIENT	(0.2, 0.2, 0.2, 1.0)	材質の環境反射色
DIFFUSE	(0.8, 0.8, 0.8, 1.0)	材質の拡散反射色
AMBIENT_AND_DIFFUSE		AMBIENTとDIFFUSEの両方
SPECULAR	(0.0, 0.0, 0.0, 1.0)	材質の鏡面反射色
SHININESS	0	0～128の鏡面反射に対する指数
EMISSION	(0, 0, 0, 1)	材質の放射色

アルファブレンド

BlendFuncコマンドでは、アルファブレンドを作成できます。そのためには、次の構文を使って、BlendFuncメッセージをシーンに送ります。

```
scene << BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
```

SRC_ALPHA と ONE_MINUS_SRC_ALPHA は OpenGL の定数で、アルファ値を使って既存の表示バッファに対するブレンドを行います。アプリケーションによっては、*z* バッファテストや、要素を後ろから前に向かって順番にレンダリングする手法を無効にする必要があります。デフォルトでは、*z* バッファテストが行われ、透明な多角形の後ろにオブジェクトを描画することはできません。

BlendFunc で使用できるすべての定数（大半は JSL プログラマには不要）の詳しい説明については、opengl.org にある OpenGL マニュアルを参照してください。

霧

霧を使うと、図形を遠方にフェードして、モデルをより現実に近付けることができます。すべてのタイプの幾何図形に霧をかけることができます。霧をオンにするには、FOG 引数を使います。

```
Enable (FOG)
```

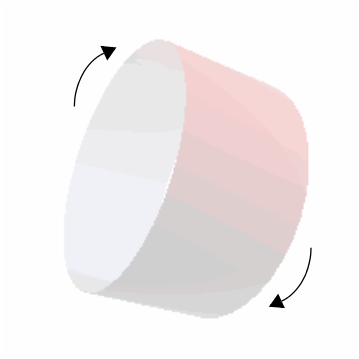
例

次の例では、照明、霧、正規化など、この節で説明した複数の概念を使っています。このスクリプトは、2 つの光源の影響を受ける、回転する円柱を描きます。

```
scene = SceneBox( 300, 300 ); // シーンボックスを作成する
New Window( "円柱", scene ); // シーンをウィンドウに配置する

for (i=1, i<360, i++,

scene << Clear;
// レンズは 45 度、近い距離（near）はカメラから 3 単位、遠い距離（far）は 7 単位
scene << Perspective( 50, 1, 10 );
// カメラで原点 (0,0,0) が見えるように移動する
scene << Translate( 0.0, 0.0, -2 );

scene<<Enable(Lighting);
scene<<Enable(Light0);
scene<<Enable(Light1);
scene<<Light(Light0,POSITION,1,1,1,1); // ビュアーに近い
scene<<Light(Light0,DIFFUSE,1,0,0,1); // 赤色の光源

scene<<Light(Light1,POSITION,-1,-1,-1,1);// オブジェクトの背後
scene<<Light(Light1,DIFFUSE,.5,.5,1,1); // 青灰色の光源

scene<<Enable(Fog);
scene<<Enable(NORMALIZE);

scene << Rotate(i,1,0,0);
scene << Rotate(i*3, 0, 1, 0);
scene << Rotate (i*3/2, 0, 0, 1);
```

```
scene << Cylinder(0.5, 0.5, 0.5, 40,10);
scene << Update;
Wait(0.01); )
```

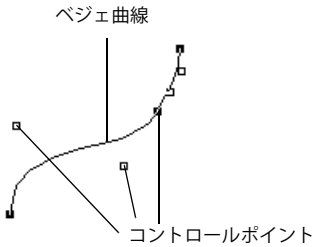
図13.18 霧



ベジェ曲線

このマニュアルでは、ベジェ曲線についての詳しい説明は省略します。JSLには、曲線や曲面とそれに対するメッシュを定義・描画するためのコマンドが用意されています。

図13.19 ベジェ曲線



1 次元評価機能

1次元のベジェ曲線を定義するには、Map1コマンドを使います。

```
Map1(target, u1, u2, stride, order, matrix)
```

*target*引数には、制御点で何を表すかを定義します。*target*引数の値は、表13.6に示しています。Enableコマンドを使って、引数を有効にする必要があります。

表 13.6 Map1 の *target* 引数とデフォルト値

target 引数	意味
MAP1_VERTEX_3	(*x*, *y*, *z*) 頂点座標
MAP1_VERTEX_4	(*x*, *y*, *z*, *w*) 頂点座標
MAP1_INDEX	色インデックス
MAP1_COLOR_4	R、G、B、A
MAP1_NORMAL	法線座標
MAP1_TEXTURE_COORD_1	*s* テクスチャ座標
MAP1_TEXTURE_COORD_2	(*s*,*t*) テクスチャ座標
MAP1_TEXTURE_COORD_3	(*s*,*t*,*r*) テクスチャ座標
MAP1_TEXTURE_COORD_4	(*s*,*t*,*r*,*q*) テクスチャ座標

2 番目の 2 つの引数（*u1* と *u2*）には、曲線の範囲を指定します。*stride* は、記憶されている制御点の、各ブロックごとの数値の個数（つまり、ある制御点から、次の制御点までのオフセット）です。*order* は、曲線の次数に 1 を足した値です。*matrix* には、制御点を指定します。

たとえば、Map1(MAP1_VERTEX_3, 0, 1, 3, 4, *<4x3 matrix>*) のように指定します。これは、2 つの端点に対し、制御点を 2 つ設定してベジェ曲線を定義する典型的な方法です。

MapGrid1 と EvalMesh1 コマンドを使って、等間隔のメッシュを定義し、適用します。

```
MapGrid1(un, u1, u2)
```

このコマンドは、*u1* から *u2* の範囲にわたって *un* 分割されるメッシュをセットアップします。0 から 1 までを範囲とすれば、コーディングが簡素化されます。

```
EvalMesh1(mode, i1, i2)
```

このコマンドは、*i1* から *i2* まで実際にメッシュを生成します。*mode* には、POINT または LINE を指定できます。EvalMesh1 コマンドは、固有の Begin 節および End 節を作成します。

次のスクリプト例では、ベジェ曲線を描きます。乱数によって決められた制御点をもとに、滑らかな曲線が描かれます。最初と最後の点だけが曲線上にあります。NPOINTS=4 を指定していますので、3 次のベジェ曲線が描かれています。

```
boxwide=500;
boxhigh=400;

gridsize=100; // 値を大きくすると、より見やすい間隔になる

NPOINTS = 4;
   /* 2 以上 8 以下の値を指定してください。この範囲外の数値は、実装方法によっては正しく動作しないかもしれません。この値は、曲線の次数に 1 を足したものです。*/
```

```
points = J(NPOINTS, 3, 0);
   // x、y、z の 3 次元配列を作成する
for( x = 1, x <= NPOINTS, x++,
   points[x, 1] = (x-1)/(NPOINTS-1) - .5;
      // x の範囲は -.5 ～ +.5
   points[x, 2] = randomuniform() - .5;
      // y は同じ範囲の乱数
   points[x, 3] = 0;
      /* 1 つの平面上に曲線を描くので、z は常にゼロ */
);

spline = SceneBox(boxwide,boxhigh);

spline << ortho( -.6, .6, -.6, .6, -2, 2 );
   /* x と y の範囲は -0.5 ～ 0.5 なので、それよりわずかに大きくする */

spline<<Enable(MAP1_VERTEX_3);
spline<<MapGrid1(gridsize, 0, 1);
spline<<Color(.2,.2,1); // 青色の曲線

spline<<Map1( MAP1_VERTEX_3, 0, 1, 3, NPOINTS, points );
spline<<Line Width(2); // 細すぎない曲線にする
spline<<EvalMesh1(LINE, 0, gridsize ); // LINE を POINT に変更して試してください

spline<<Color(.2, 1, .2);
spline<<Point Size(4); // 大きい緑色の点

// 点を表示してラベルを付ける
for( i=1, i <= NPOINTS, i++,
  spline<<Begin(POINTS);
  spline<<Vertex(points[i,1], points[i,2], points[i,3]);
  spline<<End;
  spline<<Push Matrix;
  spline<<Translate(points[i,1], points[i,2], points[i,3]);
  spline<<Text(center, bottom, .05,char(i));
  spline<<Pop Matrix;
);

New Window(" スプライン ", spline);
```

http://www.tinaja.com/glib/bezconn.pdf では、勾配と変化率が連結点で一致している、区分的な3次式を求める方法が説明されています。上記の例は区分的な曲線は1本だけで、複数の区分的な曲線を連結しているわけではありません。

2 次元評価機能

2 次元評価機能は、1 次元評価機能と対をなすもので、使い方も似ています。

```
Map2(target, u1, u2, ustride, uorder, v1, v2, vstride, vorder, matrix)
Eval Coord2(u, v)
```

*target*引数の値は、*Map1*を*Map2*に置き換えれば、表 13.6（533 ページ）に示した値と同じです。*u1*、*u2*、*v1*、および *v2*の値は、2 次元メッシュの範囲を指定します。

たとえば、Map2(MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, *<16x3 matrix>*) のように指定します。これは 16 点によってベジェ曲面を定義する典型的な方法です。

MapGrid2 と EvalMesh2 コマンドを使って、等間隔のメッシュを定義し、適用します。

```
MapGrid2(un, u1, u2, vn, v1, v2)
```

このコマンドは、*u1*から *u2*、*v1*から *v2*までの範囲にわたって、*un*分割および *vn*分割されるメッシュを作成します。0 から 1 までを範囲とすれば、コーディングが簡素化されます。

```
EvalMesh2(mode, i1, i2, j1, j2)
```

このコマンドは、*i1*から *i2*、*j1*から *j2*までのメッシュを実際に生成します。*mode*には、POINT、LINE、または FILL を指定できます。EvalMesh2 コマンドは、固有の Begin 節および End 節を作成します。

マウスの使用

マウスの動作に対する処理は、2 つのフィードバック関数によってサポートされています。Patch Editor.jsl サンプルスクリプトでは、これらの関数を使って、点をドラッグ & ドロップする例を示しています。そのスクリプトの一部、マウス動作に対するコールバック関数について、以下に説明します。スクリプトを実行するには、「Sample Scripts」フォルダの「Scene3D」フォルダにある「PatchEditor.jsl」を開きます。

```
topClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 1, 2 );
   1;
);
frontClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 1, 3 );
   1;
);
rightClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 2, 3 );
   1;
);

Click3d = Function( {x, y, m, k, hitlist},
   If( m == 1,
```

```
        If( N Items( hitlist ) > 0,
           CurrentPoint = hitlist[1][3], /* リスト内の最初の行列が最も近接している。行列の 3
      番目の要素は ID */
           CurrentPoint = 0
        );
        makePatch();
     );
     0; /* 0を戻すと、最初のマウスの押下時だけ処理。ドラッグ中も処理する場合は 1 を戻すようにする
     （その場合、マウスを放すと処理が終了。天体球は表示されない）。*/
  );

  /* 3 つの Click2d 関数から共通して呼び出される関数 */
  dragfunc = function( { x, y, m, ix, iy}, /* ix と iy は、座標を含む行列において、X、Y、お
     よび Z の列を示すためのインデックス */
     If( CurrentPoint > 0,
        points[CurrentPoint, ix] = (x / boxwide) * (orthoright - ortholeft) +
     ortholeft;
        points[CurrentPoint, iy] = (y / boxhigh) * (orthotop - orthobottom) +
     orthobottom;
        makepatch();
     );
  );
```

2-D 関数を呼び出すときの引数は、X、Y、M、K です。

- X と Y はマウスの座標です。
- M は、マウスとボタンの状態を示します。M=0 は、マウスが放されていることを表します。M=1 は、ボタンが押されたことを表します。M=2 は、ボタンが押されたままマウスが移動していることを表します。M=3 は、ボタンが放されたことを表します。
- K は、Shift、Alt、Ctrl の各キーに関連付けられます。K=1 は Shift キー、K=2 は Ctrl（command）キー、K=3 は Alt（Option）キーを表します。

3-D 関数も同様の方法で呼び出されます。引数は、X、Y、M、K、hitlist で、hitlist は行列のリストです。

```
[znear, zfar, id1, id2, id3, ...]
```

znear は、オブジェクトの近い頂点までの、カメラからの Z 方向の距離、zfar は、オブジェクトの遠い頂点までの、カメラからの Z 方向の距離です。行列は、znear と zfar の中間ポイントを基準にして近い方から遠い方へと並べ替えられます。リスト内の id は、表示リストに格納したプッシュ名（pushname）、ロード名（loadname）、およびポップ名（popname）の値です。

2-D 関数や 3-D 関数は、戻り値を使って、マウス処理を継続するかどうかを知らせます。継続する場合には、関数で「1」を戻してください。戻り値が 1 以外の場合、マウスのトラッキングが停止されます。2-D 関数と 3-D 関数は並行実行されないので、この機能は必要です。この例とは異なり、場合によっては、2-D ではなく 3-D でトラッキングされるように、2-D 関数で 0 を戻し、3-D 関数で 1 を戻すようにしてもかまいません。

Pick コマンド

SceneBox のコールバック関数では、2次元のマウス座標を取得した後、マウスの座標にある「名前が付けられた」オブジェクトをピックアップすることができます。以下の例では、hitlist には、(x,y) 点を中心にした 5x5 ピクセルの選択領域内に含まれる項目（最大 1000 個）の末端部の名前が戻されます。戻される形式は最後の引数で決まります。1 を指定すると 1 つの配列が、0 を指定すると深度で並べ替えられた配列のリストが戻されます。

```
Track2d=function({x, y, m, k},
hitlist = theSceneBox<<pick( x, y, 5, 5, 1000, 1 );
  if ( nrow(hitlist) > 0, // 選択領域が空ではない
     ... hitlist[1..n] // 表示リスト内で定義されたオブジェクトの名前が入れられる
  ) );
```

いっぽう、Track3d コールバック関数では、選択領域が常に 1 x 1 ピクセルであるのに加え、マウスを移動しない限り選択が行われません。通常はこの方法を使用しますが、1x1 ピクセルの選択領域は点と同じくらい小さいため、選択が困難なことがあります。また、Pick コマンドでは、マウスを移動しなくても選択が行われます。

Track3d 関数では、常に深度で並べ替えられた配列のリストが戻されます。各配列は、階層を構成する複数のオブジェクト名です（プッシュ名およびポップ名は、オブジェクトの階層を構成します）。大量のオブジェクトが選択されている場合は、並べ替えに長い時間がかかることがあります。なお、Pick 関数では、最後の引数（上の例では 1）で、並べ替えた配列のリストまたは 1 つの配列のどちらを戻すかを制御できます。1 つの配列を戻す場合は、末端部の名前だけが含まれ、それより上位の名前は含まれません。

引数

引数を使って、特定のモードと設定を指定できます。引数を有効にするには、Enable(*parameter*) コマンドを使います。引数を無効にするには、Disable(*parameter*) コマンドを使います。表 13.7 に、使用できる引数を示します。

表13.7 引数

ALPHA_TEST	LIGHT5	MAP2_TEXTURE_COORD_3
AUTO_NORMAL	LIGHT6	MAP2_TEXTURE_COORD_4
BLEND	LIGHT7	MAP2_VERTEX_3
CLIP_PLANE0	LIGHTING	MAP2_VERTEX_4
CLIP_PLANE1	LINE_SMOOTH	NORMALIZE
CLIP_PLANE2	LINE_STIPPLE	POINT_SMOOTH
CLIP_PLANE3	MAP1_COLOR_4	POLYGON_OFFSET_FILL
CLIP_PLANE4	MAP1_INDEX	POLYGON_OFFSET_LINE
CLIP_PLANE5	MAP1_NORMAL	POLYGON_OFFSET_POINT
COLOR_LOGIC_OP	MAP1_TEXTURE_COORD_1	POLYGON_SMOOTH
COLOR_MATERIAL	MAP1_TEXTURE_COORD_2	POLYGON_STIPPLE
CULL_FACE	MAP1_TEXTURE_COORD_3	SCISSOR_TEST
DEPTH_TEST	MAP1_TEXTURE_COORD_4	STENCIL_TEST
DITHER	MAP1_VERTEX_3	
FOG	MAP1_VERTEX_4	
LIGHT0	MAP2_COLOR_4	
LIGHT1	MAP2_INDEX	
LIGHT2	MAP2_NORMAL	
LIGHT3	MAP2_TEXTURE_COORD_1	
LIGHT4	MAP2_TEXTURE_COORD_2	

第 14 章

JMP の拡張

外部データソース、分析ツール、オートメーション

この章では、定期的に行う作業をプログラミングするときに役立つ、以下のスクリプト機能を紹介します。

- 測定機器などからのリアルタイムデータを取り込むデータフィード
- JMP スクリプト言語（JSL）を使った SAS、MATLAB、R の使用
- Excel との連携
- データベースへの接続
- OLE オートメーション（外部アプリケーションからの JMP の制御）

定期的な作業のプログラミングに関係した JSL 関数には、`Caption`、`Speak`、`Print`、`Write`、`Mail` といったものもあります。これらのコマンドについては、「プログラミング手法」の章の「メッセージを出力する関数」（252 ページ）に説明があります。

目次

リアルタイムのデータ取得 541
データフィードオブジェクトの作成 541
リアルタイムデータの読み込み 542
メッセージ付きデータフィードの制御 543
ダイナミックリンクライブラリ（DLL） 547
JSL でのソケットの使用 550
データベースアクセス 553
SAS の使用 556
SAS DATA ステップの作成 556
計算式列の SAS データステップコードの作成 556
SAS 変数名 557
SAS マクロ変数の値の取得 557
SAS Metadata Server への接続 558
環境設定 561
サンプルスクリプト 561
MATLAB の使用 562
MATLAB のインストール 563
R の操作 565
R のインストール 565
JMP から R へのインターフェース 567
R の JSL スクリプト可能なオブジェクトインターフェース 567
JMP データタイプと R データタイプの相互変換 567
トラブルシューティング 570
例 571
Excel の使用 572
OLE オートメーション 573
Visual Basic を使った JMP のオートメーション 573
Visual C++ を使った JMP のオートメーション 581

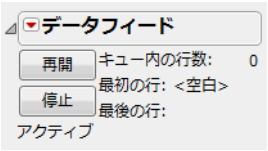
リアルタイムのデータ取得

データフィードは、シリアルポートに接続された測定機器などから、データをリアルタイムで読み込む機能です データフィードオブジェクトによって、入力用のキューを設定し、データの読み込みをバックグラウンドで処理できます。キューに届いたデータをデータテーブルに送るなどの処理も、スクリプトで実行できます。

たとえば、com1 ポートからレコードを取得し、それをログにリストするには、次のスクリプトを実行します。

```
feed = Open Datafeed(
   Connect( Port( "com1:" ), Baud( 9600 ), DataBits( 7 ) ),
   Set Script( Print( feed << getLine ) )
);
```

図14.1 データフィードウィンドウではステータスの確認や制御が可能



データフィードオブジェクトの作成

データフィードオブジェクトを作成するには、接続の詳細を指定した Open DataFeed コマンドを使います。

```
feed = Open DataFeed( options );
```

引数は必須ではありません。データフィードオブジェクトを引数を指定せずに作成して、後からメッセージを送ることもできます。ポートへ接続したり、入力されたデータを処理するためのスクリプトを設定したりすることは、メッセージでも行えます。しかし、通常は、Open DataFeed 関数の引数において、データフィードの基本設定は行っておきます。そして、引き続いて、データフィードを更に制御するのに必要なメッセージを送ります。後述のオプションはどれも、Open DataFeed 関数の引数としても、またデータフィードオブジェクトに送るメッセージとしても使用できます。

上記の例のように、データフィードオブジェクトへの参照を、グローバル変数に格納しておくと良いでしょう（上記の例では、*feed* という変数に格納している）。変数に格納しておけば、後から、簡単にオブジェクトにメッセージが送れます。なお、既存のオブジェクトへの参照を、添え字を使って格納することも可能です。たとえば、次のスクリプトは、2 番目に作成したデータフィードオブジェクトへの参照を変数に格納しています。

```
feed2 = Data Feed[2];
```

オプション

（Windowsのみ）生のデータソースに接続するには、Connect( )を使ってポートの詳細を指定します。設定項目は、それぞれ1つだけ引数をとります。ここの構文の説明では、引数間の記号「|」は「または」を意味します。接続する場合はPortを必ず指定してください。ポートを指定しなくてもオブジェクトは機能しますが、データフィードには接続されません。最後の3つの項目、DTR_DSR、RTS_CTS、XON_XOFFはブール値の引数をとり、データフィードがデータ取得可能な状態になったことを通知する際に、どの制御文字を送受信するかを指定します。通常、3つのうちの1つをオンにします。

```
feed = Open Datafeed(
   Connect(
      Port( "com1:" | "com2:" | "lpt1:" | ... ),
      Baud( 9600 | 4800 | ... ),
      Data Bits( 8 | 7 ),
      Parity( None | Odd | Even ),
      Stop Bits( 1 | 0 | 2 ),
      DTR_DSR( 0 | 1 ), // データターミナル準備
      RTS_CTS( 0 | 1 ), // 送信をリクエスト | 送信をクリア
      XON_XOFF( 1 | 0 ) // 送信器オン | 送信器オフ
   );
);
```

このコマンドにより、スクリプト可能なデータフィードオブジェクトが作成され、そのオブジェクトへの参照をグローバル変数 *feed*に格納します。引数Connectは、通信ポートを監視してデータのライン（行）をとりまとめるスレッドを立ち上げます。このスレッドは、ラインが確保されるまで文字を受け取ります。そして、それをキュー（待ち行列）に加え、スクリプトをコールするイベントのスケジュールを立てます。

注：データフィードの場合、データのライン（行）は単一の値です。データフィードのラインと、データテーブルの行を混同しないでください。後者は、1 つの行に対して複数の値を持つことができます。

Set Scriptはデータフィードオブジェクトにスクリプトを割り当てます。このスクリプトは、データが届くたびにOn DataFeedハンドラによって実行されます。Set Scriptの引数には、実行するスクリプトをそのまま記述するか、またはスクリプトを含むグローバル変数を指定します。

```
feed = Open Datafeed( set script( myScript ) );
feed = Open Datafeed( set script( Print( feed << getLine ) ) );
```

データフィードのスクリプトでは、通常Get Lineを使って1ラインのデータを取得し、そのラインに対して処理を行います。このスクリプトはラインのデータを解析し、結果をデータテーブルに追加します。

リアルタイムデータの読み込み

外部データソースから、物理または論理通信リンクを介して別の機器に情報を送信することを、**ライブデータフィード**といいます。Windowsコンピュータのシリアルポートを介してJMPをライブデータフィード元に接続し、データストリームをリアルタイムで読み取ることができます。次の点を確認してください。

- データフィードは、標準の9ピンシリアルポートを介して行う必要があります。シリアルポートをシミュレートするドライバがなければ、USBポートを介してデータを読み取ることはできません。
- 接続機器の正確なボーレート、パリティ、ストップビット、データビットが必要です。

接続機器に関する数値を、Open Datafeed()コマンドの引数に指定します （以下のスクリプト例の4800、even、2、および7を置き換えてください）。次に、データフィード元の機器をコンピュータに接続し、スクリプトを実行します。

```
streamScript = expr( line = feed <<Get Line;show(line);
              len = length(line); show(len);
   if (length(line)>=1, show("Hi"); show(line);
                    field = substr(line,5,8); show(field);
        x = Num(field); show(x);
        if (!IsMissing(x), current data table()<<add row({:Column1=x});
      show(x);
   )));
feed = open DataFeed(Baud Rate(4800),parity(even),Stop bits(2), Data bits(7));
feed<<Set Script(streamScript);
feed<<Connect;
```

JMPと外部のデータフィード機器の通信設定を揃えるには、[**ファイル**] > [**環境設定**] > [**通信**] を選びます。お使いの外部機器の適切な設定については、機器のマニュアルを参照してください。

メッセージ付きデータフィードの制御

データフィードオブジェクトは、ConnectやSet Scriptなどのいくつかのメッセージに応答します。これらのメッセージは、Open Datafeedの引数としてすでに説明しましたが、既存のデータフィードオブジェクトに送るメッセージとしても使用できます。

```
feed << connect( port( "com1:" ), baud( 4800 ), databits( 7 ), parity( odd ),
  stopbits( 2 ) );
feed << set script( myScript );
```

以下のメッセージはOn Data Feedの引数としても使用できます ただし、既存のデータフィードオブジェクトへ送るメッセージとして使用する方が一般的です。

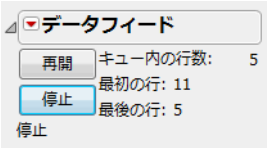
スクリプトからデータフィードへデータを送ることもできます。この方法により、データフィードを簡単にテストできます。引数にはテキストまたはテキストを格納するグローバル変数をとります。

```
feed << Queue Line( "14" );
feed << Queue Line( myValue );
```

以下は5行のデータを待機させるテストです。

```
feed << queue line( "11" );
feed << queue line( "22" );
feed << queue line( "33" );
feed << queue line( "44" );
feed << queue line( "55" );
```

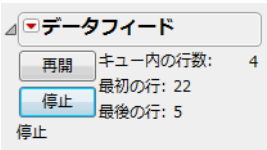
図 14.2 データフィード: キューに 5 つの行



キューにある最初の行を取得するには、Get Line（単数形であることに注意）メッセージを使います。行を取得すると、その行はキューから削除されます。上のテスト用スクリプトでは 5 つのデータがキューに入れられています。Get Line は最初のラインを取得した上で、それをキューから削除します。

```
feed << Get Line
   "11"
```

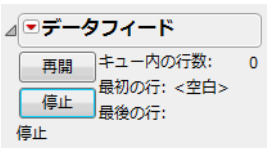
図 14.3 データフィード: キューに 4 つの行



データをすべてリストに移し、キューを空にするには Get Lines（複数形であることに注意）を使います。例では、テスト用スクリプトから次の 4 行を { } 形式のリストで戻します。

```
myList = feed << GetLines;
   { "22", "33", "44", "55" }
```

図 14.4 データフィード: キューに 0 個の行



キューにあるデータの処理を停止し、後で再開するには、データフィードウィンドウの［**停止**］ボタンおよび［**再開**］ボタンをクリックするか、または同じ機能を持つ次のメッセージを送ります。

```
feed << Stop;
feed << Restart;
```

データフィードとそのウィンドウを閉じるには、次のメッセージを送ります。

```
feed << Close;
```

ライブデータソースからの接続を切断するには:

```
feed << Disconnect
```

データフィードの例

データの読み取り

以下は、データフィードに対するスクリプトの例です。このスクリプトでは、文字列の長さを読み取り、もし14文字以上であれば、11文字目以降を数値として取り込み、データテーブル中の「太さ」列に、1行を追加します。

```
feed = Open Datafeed( );
myScript = Expr(
   line = feed << Get Line;
   If( Length( line ) >= 14,
      x = Num( Substr( line, 11, 3 ) );
      If( !Is Missing( x ),
         Current Data Table( ) << Add Row( { 太さ = x} )
      );
   );
);
```

上記のスクリプトを、既存のデータフィードオブジェクトに割り当てるには、次のように Set Script を使います。

```
feed << Set Script( myScript );
```

ライブ管理図のセットアップ

以下は、新しいデータテーブルを作成し、データフィードに基づいて管理図の作成を開始するスクリプトの例です。

```
// 1つの列を持つデータテーブルを作成する
dt = New Table( "ギャップの幅" );
dc = dt << New Column( "ギャップ", Numeric, Best );

// 管理図のプロパティを設定する
dt << Set Property( "管理限界", {XBar( Avg( 20 ), LCL( 19.8 ), UCL( 20.2 ) )} );
dt << Set Property( "Sigma", 0.1 );

// データフィードの作成
feed = Open Datafeed( );
feedScript = Expr(
   line = feed << get line;
   z = Num( line );
   Show( line, z ); // ログまたはデバッグ用
   If( !Is Missing( z ),
      dt << AddRow( {:ギャップ = z} )
   );
);
feed << SetScript( feedScript );
```

```
// 管理図を描く
Control Chart(
   Sample Size( 5 ),
   K Sigma( 3 ),
   Chart Col(
      ギャップ,
      XBar(
         Connect Points( 1 ),
         Show Points( 1 ),
         Show Center Line( 1 ),
         Show Control Limits( 1 )
      ),
      R(
         Connect Points( 1 ),
         Show Points( 1 ),
         Show Center Line( 1 ),
         Show Control Limits( 1 )
      );
   );
);

// 機器からのデータフィードを始めるか、またはテスト用データを送って
// 動作確認をする（いずれか１行をコメントアウトする）:
// feed<<connect(Port("com1:"), Baud(9600));
For( i = 1, i < 20, i++,
   feed << Queue Line( Char( 20 + Random Uniform( ) * .1 ) )
);
```

データテーブルにスクリプトを格納する

前述のようなデータフィードのスクリプトを、データテーブルのOn Openプロパティに記述しておくことにより、定型的な処理をさらに自動化できます。On Openプロパティに設定されたスクリプトは、データテーブルが開くたびに自動的に実行されます（環境設定で実行しないように設定することもできます）。データフィード処理のスクリプトをOn Openプロパティに設定したデータテーブルを保存しておけば、そのデータテーブルを開くたびに、データフィードのスクリプトが実行され、データデータが取得されます。

表14.1 データフィードへのメッセージ

メッセージ	構文	説明
Open Data Feed Data Feed	feed = Open Datafeed(commands)	データフィードオブジェクトを作成する。以下のメッセージはいずれも、Open DataFeed内のコマンド、または、既存のデータフィードオブジェクトに送るメッセージとして使用できます。Data Feedは同義語。

表14.1 データフィードへのメッセージ（続き）

メッセージ	構文	説明
Set Script	feed << Set Script(script)	データラインが届くたびに実行されるスクリプト（*script*）を設定する
Get Line	feed << Get Line	データフィードのキューの中から1ラインのデータを戻し、削除する。
Get Lines	feed << Get Lines	データフィードのキューのデータすべてをリストにして戻し、削除する。
Queue Line	feed << Queue Line(*string*)	データフィードのキューの最後へ1ラインのデータ送る。
Stop	feed << Stop	データラインへの処理を停止する。
Restart	feed << Restart	データラインへの処理を再開する。
Close	feed << Close	データフィードオブジェクトとそのウィンドウを閉じる。
Connect	feed << Connect(Port("com1:" | "lpt1:" | ...), Baud(9600 | 4800 | ...), Data Bits(8 | 7), Parity(None | Odd | Even), Stop Bits(1 | 0 | 2), DTR_DSR(0 | 1), RTS_CTS(0 | 1), XON_XOFF(1 | 0));	（Windowsのみ）デバイスに接続するために、ポートの設定を行う。引数の間にある記号「|」は「または」を意味し、設定ごとにどれか1つの引数を選びます。
Disconnect	feed << Disconnect	（Windowsのみ）データフィードオブジェクトをアクティブにしたまま、データフィードのキューとデバイスとの接続を切断する。

ダイナミックリンクライブラリ（DLL）

注：64ビット版のJMPは32ビットDLLがロードできず、32ビット版のJMPは64ビットDLLがロードできません。過去に32ビットDLLを使用していて、現在64ビット版のJMPを使用している場合は、DLLを再コンパイルして64ビット版でロードできるようにしなければなりません。

JMPスクリプト言語（JSL）によって、DLLをロードし、そのDLLに含まれている関数を呼び出すことができます。DLLの呼び出しを行うJSL関数が1つと、メッセージが6つあります。

```
dll_obj = Load DLL("path" <, AutoDeclare(Boolean | Quiet | Verbose) |Quiet |
  Verbose > )
```

Load DLL()は、パス（*path*）で指定されたDLLをロードします。ログウィンドウにメッセージが表示されないようにするには、AutoDeclare(Quiet)引数を使用します。

DLL内の関数を呼び出すには、Declare Functionメッセージを使って、宣言します。

```
dll_obj <<Declare Function("name", Convention(named_argument), Alias("string"),
  Arg(type, "string"), Returns(type), other_named_arguments)
```

Aliasは、JSLで使用できる代替の名前を定義します。たとえば、DLL内の"Message Box"という関数に対して、Alias("MsgBox")を宣言した場合、その関数を次のようにも呼び出せるようになります。

```
result = dll_obj <<MsgBox(...)
```

Conventionの引数には、次のいずれかを指定します。

- STDCALLまたはPASCAL
- CDECL

ArgとReturnsのtype引数には次のいずれかを指定します。

表14.2 ArgおよびReturnsの種類

Int8	UInt8	Int16	UInt16
Int32	UInt32	Int64	UInt64
Float	実数(double)	AnsiString	UnicodeString
Struct	IntPtr	UIntPtr	ObjPtr

Declare Functionメッセージの引数については、『スクリプト構文リファレンス』のDLLの節を参照してください。

最後に、UnLoadDLLメッセージでDLLをアンロードします。

```
dll_obj << UnLoadDLL
```

例

Windowsの32ビットDLLと64ビットDLLのうち、ユーザのコンピュータに応じて適切な方をロードするようなスクリプトを書きたいとします。この例は、Windowsオペレーティングシステムを確認し、次にコンピュータのプロセッサのビットレベルを確認します。

```
If( Host is( Windows ),
   If(
      Host is( Bits64 ),
      //  64 ビット DLL をロードする
         dll_obj = Load DLL( "c:\Windows\System32\user32.dll" ),
      // 32 ビット DLL をロードする
         dll_obj = Load DLL( "c:\Windows\SysWOW64\user32.dll"),
      // どちらの DLL も見つからない場合は実行を停止する
         Throw
      );
   dll_obj <<DeclareFunction(
      "MessageBoxW",
      Convention( STDCALL ),
      Alias( "MsgBox" ),
      Arg( IntPtr, "hWnd" ),
      Arg( UnicodeString, "message" ),
      Arg( UnicodeString, "caption" ),
      Arg( UInt32, "uType" ),
      Returns( Int32 )
   );
   result = dll_obj << MsgBox(
      0,
      "JMP からのメッセージです。",
      "DLL の呼び出し ",
      321
   );
   Show( result );
);
dll_obj << UnLoadDLL
```

その他の DLL メッセージ

Show Functions メッセージは、Declare Function で宣言された関数をすべてログに送ります。

```
dll_obj << Show Functions;
```

独自の DLL を作成するときには、DLL 内に JSL の関数宣言スクリプトを用意しておくことができます。Get Declaration JSL メッセージは、DLL 内の関数宣言スクリプトをログに送ります。

```
dll_obj << Get Declaration JSL;
```

単純な DLL 関数の場合

DLL の関数が非常に単純な場合には、Call DLL メッセージによって、DLL の関数を呼び出すこともできます。その際、関数に応じて、表 14.3 のようなシグニチャを指定します。

```
dll_obj <<Call DLL(function_name, signature, arguments)
```

表 14.3 シグニチャ値

シグニチャ	引数のタイプ
"v"	void
"c"	string
"n"	int

その後、引数（*argument*）で渡された関数が呼び出されます。

例

```
// パスを指定して、DLL をロードする
dll_obj = Load DLL( "D:¥Release¥MyDLL.DLL" );
// MyExportedFcn という名前の DLL 内部の関数を呼び出す
// この関数は 1 つの数値を入力する必要がある
// ここでは、入力引数の値を 654 とする
dll_obj << CallDLL( "MyExportedFcn", "n", 654 );
dll_obj << UnLoadDLL();
```

実際にこの 2 行のスクリプトで実行しているのは、MyExportedFcn を呼び出して、この関数に入力値として 654 を渡すという処理です。概念的には、JSL スクリプトが MyExportedFcn(654) を実行しているようなものです。

JSL でのソケットの使用

データをフィードするもう 1 つの方法は、ソケット通信です。JSL では、2 種類のソケットが作成できます。JSL では、2 種類のソケットが作成できます。

ストリーム　ストリームソケットは、JMP ともう 1 つのコンピュータとの間で信頼性の高い接続を確立します。JMP を実行しているコンピュータ、業務用機器、データ収集用コンピュータ、プリンタといったソケット通信が可能なデバイスと通信できます。ソケット通信を行える HTTP Web サーバーなどとも通信できます。

データグラム　データグラムソケットも、JMP ともう 1 つのコンピュータとの間に接続を確立しますが、信頼性はやや劣ります。データグラムはコネクションレスで、情報が何度も届いたり、まったく届かなかったり、順不同で届いたりします。データグラム接続には、信頼性が保証されているストリーム接続のようなオーバーヘッドが備わっていません。データグラムは、コネクションレスのため、（同じソケットであっても）接続先のアドレスを毎回指定する必要があります。

ソケットを作成すると、別のソケットに接続するか、別のソケットからの接続を待機することができます。以下に、別のコンピュータの Web サーバーに接続してデータを取得するという簡単なプログラム例を示します。

```
tCall = Socket( );
tCall << connect( "www.jmp.com", "80" );
tCall << Send( Char To Blob( "GET / HTTP/1.0~0d~0a~0d~0a", "ASCII~HEX" ) );
tMessage = tCall << Recv( 1000 );
text = Blob To Char( tMessage[3] );
Show( text );
tCall << Close( );
```

1行目は、ソケットを作成し、それに参照名（tCall）を与えます。デフォルトでは、ストリームソケットが作成されます。ソケットの種類を指定するには、オプションの引数socket(STREAM)またはsocket(DGRAM)を使います。

2行目は、tCallソケットをJMP Webサイトのポート80（通常はHTTPポート）に接続します。

3行目は、JMP WebサーバーにGET要求を送信します。このメッセージは、JMP Webサーバーに対し、JMPホームページを返信するように要求します。GETの後の / は、開くページのパスでなければなりません。/ はルートページを開きます。

4行目は、JMP Webサーバーから最大1000バイトを受信し、情報のリストをtMessageに格納します。ソケットの呼び出しは、それぞれリストを戻します。リストの最初の要素は、呼び出しの名前です。2番目の要素はテキストメッセージで、okだけのことも、長い診断メッセージのこともあります。場合によっては、各呼び出しに応じた追加の要素が存在します。この例では、リストの3番目の要素が受信したデータです。

5行目は、受信したバイナリ情報を文字列に変換します。tMessage[3]は、Recvによって戻されるリストの3番目の要素で、JMP Webサーバーからのデータです。

6行目はログにデータを表示します。

最後の行はソケットを閉じます。接続先のWebサーバーはすでに閉じているため、このソケットには再接続か適切な廃棄（close）が必要です。

JMPのSamples/Scriptsフォルダにソケットを使用したスクリプトの例がいくつかあります。

ソケット関連のコマンド

場合によっては、ソケットを作成して使用する前に接続先に関する情報を取得しておく必要があります。GetAddrInfo( )およびGetNameInfo( )は、アドレス引数とオプションのポート引数を取得し、情報のリストを戻します。たとえば、次のような設定が行えます。

```
Print( Get Addr Info( "www.sas.com" ) );
Print( Get Addr Info( "www.sas.com", "80" ) );
Print( Get Name Info( "149.173.5.120" ) );
   {"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120", "0"}}
   {"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120",
      "80"}}
   {"Get Name Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "www.sas.com", "0"}}
```

複数の回答がある場合があります。その場合は、サブリストが繰り返されます。これらの関数は、処理にかなり時間がかかる場合があるので、すべての Web サイト名を記載したデータテーブルを作成することはお勧めできません。IPV6 との互換性を維持するために、数値アドレスではなく、"www.sas.com" のような名前を使用してください。

ソケットへのメッセージ

socket( ) を使って作成したソケットには、さまざまなメッセージを送ることができます。

connect リスニングソケットに接続する。接続に成功した場合は {"connect", "ok"}、失敗した場合はエラーが戻されます（たとえば、{"connect", "CONNREFUSED: The attempt to connect was forcefully rejected."})。

close 処理の終了時に接続を閉じます。リストを戻します（たとえば {"Close", "ok"}）。

send ソケットの相手側に STREAM メッセージを送信します。

sendto ソケットの相手側に DGRAM メッセージを送信します。

recv STREAM メッセージを受信します。データとその他の情報がリスト形式で戻されます。Recv には、受信できるバイト数を指定する数値引数が必要です。

recvfrom DGRAM メッセージを受信します。

ioctl ソケットのブロック動作を制御します。デフォルトでは、ソケットはブロックに設定されています。データが使用できるようになるまで、ソケットは JSL プログラムに制御を戻しません。これによってスクリプトの作成は容易になりますが、接続先がデータの供給に失敗した場合に特に頑健ではありません。非ブロックに設定されているソケットは、即座に "ok" コードとデータを戻すか、または "WOULDBLOCK: ..." コードを戻しますが、ソケットがブロックのときは、データが使用可能になるまで待たなければなりません（次の JSL ステートメントへの進行をブロックする）。

重要: この問題を回避するため、JSL コールバックを使用するバックグランド処理があります。recv、recvfrom、または accept は、ソケットを非ブロックに設定し、Wait ステートメントの間や JMP がアイドルのときにポーリングして、バックグランドで動作させることができます。

ioctl はリストを戻します。たとえば、{"ioctl", "ok"}、またはソケットがバインド（下の bind を参照）または接続されていない場合は {"ioctl", "NOTCONN: The socket is not connected."} を戻します。

bind クライアントソケットが待機（リッスン）するアドレスを、サーバーソケットに通知します。bind は、ローカルコンピュータのポートをソケットに関連付けます。ソケットが listen するためには、この処理が必要です （下を参照）。bind で指定したポートは、接続するソケットに対していつも使用されるとは限りません。オペレーティングシステムが、使用されていないポートを選びます。サーバーには bind が必要です。サーバーに接続したいユーザは、どのポートが使用されるのかを知っている必要があるためです。通常使用されるポートは、HTTP ポートである 80 です。bind はリストを戻します。たとえば、{"bind", "ok"}、または使用しているコンピュータ上にない名前にバインドしようとしている場合は {"bind", "ADDRNOTAVAIL: The specified address is not available from the local

machine."}を戻します。別のソケットは、使用しているコンピュータの名前と番号がわかっていれば、このソケットに接続することができます。

listen 接続をリッスンするように、サーバーのソケットに伝えます。リスニングソケットは、他のソケットからの接続をリッスンします。一度、リッスンの状態にすれば、継続して、リッスンの状態になっています。accept（下を参照）は、他のソケットからの接続を受け入れるのに使用します。listenはリストを戻します。たとえば、{"listen", "ok"}、またはバインドコールに失敗した場合は、{"listen", "INVAL: The socket is（または状況によってはis not) already bound to an address.または、Listen was not invoked prior to accept.または、Invalid host address.または、The socket has not been bound with Bind."}を戻します。

accept サーバーソケットに、接続を受け入れて新しい接続ソケットを戻すよう伝えます。acceptは、何が起こったかを記したリストを戻します。処理が成功した場合は接続先のソケットに接続される新しいソケットもリストに記されます。たとえば、{"Accept", "ok", "localhost", socket( )}などです。ここで、localhostは接続したコンピュータの名前で、4番目の引数はメッセージをsendまたはrecvするのに使用するソケットです。

getpeername 接続先について問い合わせます。GetPeerNameは、接続先のソケットに関する情報のリストを戻します。たとえば、{"getpeername", "ok", "127.0.0.1", "4087"}といったリストを戻します。サーバーソケットの場合、接続したクライアントのアドレスとポートが判明します。クライアントソケットの場合、接続要求で使用したサーバーの名前とポートを再確認できます。

getsockname 接続元について問い合わせます。GetSockNameは、こちら側のソケットに関する情報のリストを戻します。たとえば、{"getsockname", "ok", "localhost", "httpd"}といったリストを戻します。クライアントソケットの場合、オペレーティングシステムが割り当てたポートが判明します。サーバーソケットの場合、bindによってすでにその情報はわかっています。

データベースアクセス

JMPではODBCがサポートされています。Open Databaseコマンドを使ってJSLでSQLデータベースへアクセスできます。

```
dt = Open Database(
   "Connect Dialog" | "DSN=...", // データソース
   "sqlStatement" | "dataTableName" | "SQLFILE=...", // SQL ステートメント
   Invisible,  // テーブルの読み込み時にそのテーブルを非表示にするオプションのキーワード
   "outputTableName" // 新しいテーブルの名前
);
```

第1引数には、読み込みたいデータソースを指定します。次のいずれかを指定してください。

- "Connect Dialog"と指定すると、ODBC接続を設定するためのウィンドウが呼び出されます。
- "DSN="の後にデータソースの名前、および、接続に必要な情報を指定すると、そのデータソースに接続します。

たとえば、次のような設定が行えます。

```
"DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/<バージョン番号>/Samples/Import
  Data;"
```

第2引数には、次の3つのうちいずれかを2重引用符で囲んで指定します。

1. 実行するSQLステートメント。たとえば次のように、2重引用符で囲んで、SELECTステートメントを指定してください。

   ```
   "SELECT AGE, SEX, WEIGHT FROM BIGCLASS"
   ```

2. データテーブルの名前。データテーブル名だけを指定した場合、"SELECT * FROM"というSQLステートメントを実行するのと同じ処理が行われます。たとえば、第2引数を次のように指定すれば、Open Databaseは"SELECT * FROM BIGCLASS"を実行することになります。

   ```
   "BIGCLASS"
   ```

3. "SQLFILE="に続いて、実行するSQLステートメントの書かれたテキストファイルへのパス。たとえば次の引数の場合は、ディレクトリC:¥にあるファイル「mySQLFile.txt」を開き、中に書かれたSQLステートメントを実行します。

   ```
   "SQLFILE=C:¥mySQLFile.txt"
   ```

オプションの引数Invisibleは、非表示のデータテーブルを作成します。なお、非表示のデータテーブルは、明示的に閉じるまでメモリ内にとどまるため、不要になったものは閉じるよう注意してください。非表示のテーブルを閉じるには、Close(dt)を実行します。ここで、dtは、データテーブル参照の変数です。

オプションの引数*outputTableName*は、作成される出力テーブルがあれば、その名前を指定します。Open Databaseが必ずデータテーブルを戻すわけではないことに注意してください。戻り値はヌルになることもあります。データテーブルが戻されるかどうかは、実行されるSQLステートメントの種類によります。たとえば、SELECTステートメントはデータテーブルを戻しますが、DROP TABLEステートメントはデータテーブルを戻しません。

JMPのデータテーブルを、JSLを使ってデータベースに保存するには、データテーブルの参照にSave Database( )メッセージを送ります。

```
dt << Save Database("connectInfo", "TableName");
```

第1引数は、Open Databaseの場合と同様に機能します。一部のデータベースでは、既存のテーブルに置き換えてテーブルを保存することができないので注意してください。そのような場合にデータベース上のテーブルを置換するには、Open Databaseコマンドで、「DROP TABLE」というSQLステートメントを実行して、そのテーブルをあらかじめ削除してください。

```
Open Database ("connectinfo", "DROP TABLE TableName");
```

次のスクリプトは、SQLクエリでデータベースを開き、それを新しい名前でデータベースに保存し、その後、新しいテーブルを削除します。

```
dt = Open Database("Connect Dialog", "SELECT age, sex, weight FROM
   \!"Bigclass$\!"", "My Big Class");
dt << Save Database("Connect Dialog", "MY_BIG_CLASS");
Open Database("Connect Dialog", "DROP TABLE BIGCLASS.MY_BIG_CLASS");
```

注：ODBCデータベースからデータを読み込む際、ユーザIDとパスワード情報を含んだテーブル変数が追加される可能性があります。これを回避するには、pref(ODBC Hide Connection String(1))のように環境設定を設定するか、または、［**ファイル**］>［**環境設定**］>［**テーブル**］で［**ODBC接続文字列を非表示にする**］を選択します。詳細については、『JMPの使用法』を参照してください。

その他のデータベースコマンド

以下の関数を使用すると、データベースに対して、より複雑な処理を行えます。

```
Create Database Connection(" パスワードを使用した接続文字列 ");
Execute SQL(db, "SQL statement", <invisible>, <"New Table Title">);
Close Database Connection(db);
```

これら3つの関数を使用すれば、接続を開き、Execute SQLを数回呼び出した後、接続を閉じることができます。Create Database Connectionは、Execute SQLとClose Database Connectionで使用できるハンドルを戻します。

送信されたSQL文によって、データベースのテーブルから、JMPのデータテーブルが作成される場合と、作成されない場合があります。「SELECT」は通常、JMPのデータテーブルを作成します。しかし、「INSERT INTO」は、データベース内のテーブルを変更するものなので、JMPのデータテーブルは作成されません。

例

次のプログラムは、データベースへの接続を開きます。

```
dbc = Create Database Connection(
   "DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/<バージョン番号>/Samples/Import
   Data/;"
);
```

次のプログラムは、上記の接続を使って、SQLステートメントを実行します。

```
dt = Execute SQL(dbc,
   "SELECT HEIGHT, WEIGHT FROM Bigclass", "NewTable"
);
```

次のプログラムは、処理が完了したら、ODBC接続を閉じます。

```
Close Database Connection(dbc);
```

SASの使用

JMPには、JMPからSASシステムを利用する機能がいくつか用意されています。

SAS DATAステップの作成

JMPのデータテーブルに、<<Make SAS Data Stepをに送ると、SASプログラムが作成されます。このSASプログラムは、「DATAステップ」と呼ばれるものであり、SASシステム上で実行すると、JMPデータテーブルと同じデータの、SASデータセットが作成されます。例:

```
Current Data Table( ) << Make SAS Data Step
```

実行すると、ログにSASプログラムが出力されます。作成されたSASプログラムを、SAS上で実行してください。

<<Make SAS Data Step Windowを送ると、SASプログラムが、別のウィンドウに出力されます。また、プログラムのファイル名には、「.SAS」という拡張子が付けられますので、SASシステムで簡単に利用できます。

計算式列のSASデータステップコードの作成

JMPのデータテーブルに、<<Get SAS Data Step for Formula Columnsというメッセージを送ると、列の計算式に対応したSASプログラムが作成されます。次の例は、まず、計算式を含む「rate」列を作成し、そのSASプログラムを生成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "rate", Formula( :Name("身長(インチ)") / :Name("体重(ポンド
  )")));
dt << Get SAS Data Step for Formula Columns;
```

このスクリプトは、次のようなコードをログに出力します。なお、列名における日本語などの文字は、標準的なSASの変数名には使えないため、アンダーバーに置き換えられます。

```
/*%PRODUCER: JMP - DataTable Formulas */
/*%TARGET: rate */
/*%INPUT: __ */
/*%INPUT: __2 */
/*%OUTPUT: rate */
/* Code to score rate */
rate =__/__2
drop ;
```

引数に列名を指定すると、その列の計算式だけが対象となりますが、次のように列名を省略すると、すべての計算式のコードが生成されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Get SAS Data Step for Formula Columns;
```

SAS Model Managerのスコアリングコードに列計算式を含めることもできます。データテーブルに<<Get MM SAS Data Step for Formula Columnsを送ります。

```
dt = Open( "$SAMPLE_DATA¥Tiretread.jmp" );
dt << Get MM SAS Data Step for Formula Columns;
```

このスクリプトの結果もログに表示されます。

<<Get SAS Data Step for Formula Columnsの場合と同様、列名の指定はオプションです。

SAS変数名

SAS Open For Var Names( )は、SASデータセットから変数名を取得し、それらを文字列のリストで戻します。

SASの変数名には、JMPのものよりも細かい制約があります。SAS Name関数は、JMP変数名をSAS変数名に変換します。その際、特殊文字・空白・日本語を下線に変更するなどのさまざまな変換を行って、有効なSAS変数名を生成します。

```
result = SAS Name(name);
result = SAS Name({list of names});
```

文字列を含んだリストを引数に指定した場合は、各名前をスペースで区切った文字列が戻されます。例:

```
SAS Name({"x 1", "x 2"})
```

は、次の文字列を戻します。

```
" x_1 x_2"
```

SASマクロ変数の値の取得

JMPには、SASマクロ変数を問い合わせるいくつかの方法が用意されています。

たとえば、次のプログラムは、SYSTIMEマクロ変数の値を取得します。

```
systime = sas << Get Macro Var("SYSTIME");
show(systime);
```

次のプログラムは、現在、定義されているすべてのSASマクロ変数を表示します。

```
macro_names = sas << Get Macro Var Names();
show(macro_names);
```

次のプログラムは、すべてのマクロ変数の名前と値を、反復処理によって表示します。

```
macro_names = sas << Get Macro Var Names();
for(i=1, i <= N Items(macro_names), i++,
   macro_value = sas << Get Macro Var (macro_names[i]);
   output = macro_names[i] || " " || char(macro_value);
   show(output);
);
```

次のプログラムは、「test」という名前でSASマクロ変数を定義するSASコードを実行した後、その値を取得します。

```
sas << Submit("%let test = 1;");
test = sas << Get Macro Var("test");
show(test);
```

どのマクロ変数も、可能であれば数値として評価され、可能でない場合は文字となります。

SAS Metadata Serverへの接続

JMPからSASサーバーに接続し、SASデータセットを処理できます。なお、これらの一連の操作は、JSLでもプログラミングできますが、メニューでも用意されています。詳細については、『JMPの使用法』を参照してください。

接続

まず、Metaconnectコマンドで SAS Metadata Serverに接続します。

```
connected = Meta Connect("MyMetadataServer", port)
```

メタデータサーバーのSASのバージョンを指定するには、SASVersion名前付き引数を使用します。

```
connected = Meta Connect("MyMetadataServer", port, SASVersion("9.3"))
```

コンピュータの名前（myserver.mycompany.comなど）とポートだけを指定した場合、認証ドメイン、ユーザ名、およびパスワードの指定を促すプロンプトが表示されます。これは、JSLで指定することもできます。

```
connected = Meta Connect("MyMetadataServer", port, "authdomain", "user name",
   "password")
```

SAS Metadata Serverの使用が終わったら、Meta Disconnect( )で接続を解除します。引数は必要ではなく、このコマンドで現在のメタサーバー接続が閉じられます。

メタデータサーバー上で使用可能なリポジトリを確認し、使用したいリポジトリを設定できます。

```
Meta Get Repositories( );
   {"Foundation"}
Meta Set Repository("Foundation");
```

使用できるリポジトリが1つしかない場合は、自動的にそれが選択されるため、特に設定する必要はありません。

リポジトリを設定すると、使用可能なサーバーを確認することができます。

```
mylist = Meta Get Servers( );
   {"SASMain", "Schroed1", "SASMain_ja", "SASMain_zh", "SASMain_ko", "SASMain_fr",
      "SASMain_de", "SASMain_Unicode"}
```

次に、SAS接続を設定します。このコマンドを使用すると、メタデータサーバーを使用せずに、ローカルまたはリモートサーバーに直接接続できます。

```
conn = SAS Connect("SASMain");
```

ここで、connオブジェクトにDisconnectおよびConnectメッセージを送信して、SAS接続を閉じる／開くことができます。

```
conn << Disconnect();
conn << Connect();
```

上記のプログラムは、メタデータのオブジェクトにメッセージを送って、SASに接続しています。JSLでは、関数によって、SAS接続をグローバルに確立することもできます。その場合、メタデータのオブジェクトにDisconnectやConnectメッセージを送っても、グローバルなSAS接続には影響しません。ただし、グローバルなSAS接続がない場合は、メタデータのオブジェクトから開かれたSAS接続が、グローバルなSAS接続に設定されます。

SASライブラリに自動的に接続する

SASサーバーに接続する際に、メタデータで定義されたSASライブラリに自動的に接続するには、Connect Librariesを使用します。

```
conn = SAS Connect("SASMain", Connect Libraries( 1 ));
```

メタデータで定義されたすべてのライブラリに接続されるため、接続に時間がかかる場合があります。

特定のライブラリに後で接続するには、SAS Connect Librefを使用するか、またはSASサーバーオブジェクトに対してConnect Librefメッセージを送ります。

SASライブラリの表示

SASサーバーに接続し、サーバー上のライブラリを見るにはSAS Get Lib Refコマンドを使用します。

```
librefs = SAS Get Lib Refs( );
   {"BOOKS", "EGSAMP", "GENOMICS", "GISMAPS", "JMPSAMP", "JMPTEST",
   "MAILLIB", "MAPS", "OR_GEN", "ORION_RE", "ORSTAR", "SASHELP",
   "SASUSER", "TEMPDATA", "TSERIES", "V6LIB", "WORK", "WRSTEMP"}
```

必要なデータセットを含んだライブラリが割り当てられていない場合は、そのライブラリを割り当てます。

```
librefs = SAS Assign Lib Refs(“MyLib”, “c:\public\data”);
```

SASデータセットを開く

まず、SASライブラリ参照名を割り当てます。

```
SAS Assign Lib Refs("MyLib", "c:\public");
```

第1引数は、任意のSASライブラリ参照名です。第2引数は、データセットがあるサーバ上のパスです。

次に、選択したライブラリの配下にあるデータセットの名前を、リストで取得します。

```
datasets=SAS Get Data Sets("MyLib");
   {"ANDORRA", "ANDORRA2", "ANYVARNAME", "BOOKS", "BOOKSCOPYNOT", "BOOKS_VIEW",
      "CATEGORIES", "DATETIMETESTS", "MOREUGLY", "NOTTOOUGLY", "PAYPERVIEW",
      "PUBLISHERS", "PURCHASES", "PURCHASES_FULL",
```

これで、データセットを開く準備ができました。

```
dt=SAS Import Data("MyLib", "PURCHASES");
```

または

```
dt=SAS Import Data(librefs[1], datasets[12]);
```

または

```
dt=SAS Import Data("MyLib.PURCHASES");
```

これで、ライブラリ内にあるデータセットについての情報を取得できます。次のプログラムは変数名をリストで取得できます。

```
bookvars=SAS Get Var Names("MyLib.PURCHASES");
   {"purchaseyear", "purchasemonth", "purchaseday", "bookid", "catid",
   "pubid", "price", "cost"}
```

この情報を元に、読み込む変数を指定し、データセットの一部だけを読み込むこともできます。

```
dt=SAS Import Data(librefs[1], datasets[12], columns(bookvars[1], bookvars[2],
  bookvars[4]));
```

SASデータセットの保存

JMPデータテーブルを、SASデータセットに保存するには、SAS Export Data( )コマンドを使用します。

```
SAS Export Data(dt, librefs[1], datasets[4], ReplaceExisting);
```

ストアドプロセスの実行

ストアドプロセスへの参照を取得するには、次の関数を使用します。

```
stp=Meta Get Stored Process("Samples/Stored Processes/Sample: Hello World");
```

JSLでストアドプロセスのリストを取得する方法はありません。実行したいストアドプロセスへのパスを知っている必要があります。

ストアドプロセスを実行するには、ストアドプロセスにメッセージを送ります。

```
stp<<run( );
```

JMPからのSASコードのサブミット

SASプログラムを直接サブミットして、その実行結果を取得することもできます。たとえば、次のような設定が行えます。

```
SAS Submit("proc print data=sashelp.class; run;");
```

オプションの2つの引数により、SASのアウトプットやログをJMP上に表示するかどうかを指定します。

```
SAS Submit("SAS Code" <,No Output Window(True|False)> <,Get SAS Log(True|False)>);
```

また、SASのログは、次のコマンドによりいつでも得ることができます。

```
SAS Get Log( );
```

SAS Get Log( )は、SASのログを文字列として戻します。この文字列は、JSLにおける通常の文字列と同じように、JSL変数に代入して使用することができます。

環境設定

現在のSASバージョンの環境設定を取得するには、次のコマンドを使用します。

```
Get SAS Version Preference();
```

現在のSASバージョンの環境設定を設定するには、次のコマンドを使用します。

```
Preference( SAS Integration Settings( SASVersion( "9.3" ) ) );
```

サンプルスクリプト

JMPのSample/Scripts/SAS Integrationフォルダに、サンプルスクリプトが含まれています。ストアドプロセススクリプトを正常に実行するには、ストアドプロセスがお使いのSAS Metadata Server上になければなりません。ストアドプロセスは、同じフォルダのsampleStoredProcesses.spkファイルに入っています。

sampleStoredProcesses.spkをお使いのSAS Metadata Serverに読み込むには

警告：これらのストアドプロセスは、実務用のシステムではなく、テスト用のSAS Metadata Serverに読み込んでください。

1. SAS Management Console（SAS管理コンソール）を実行します。
2. 管理者権限のあるアカウントを使用してSAS Metadata Serverに接続します。
3. SAS管理コンソールの左ペインにある［**BI Manager**］ノードを展開します。
4. ツリー内で、読み込んだサンプルストアドプロセスの保存先とするフォルダに移動します。
5. SAS 管理コンソールの左ペインまたは右ペインでそのフォルダを右クリックし、［**インポート**］を選択します。

   読み込みウィザードが表示されます。

6. sampleStoredProcesses.spkへのフルパスを入力するか、または［**参照**］ボタンを使って指定します。
7. ウィザードの読み込みオプションで、［**すべてのオブジェクト**］を選択します。
8. ［**Next**］をクリックします。

   読み込みプロセス中、パネルに、**Application Server**と**ソースコードリポジトリ**の値を指定する必要があることが表示されます。
9. ［**Next**］をクリックします。

   パネルで、読み込んだストアドプロセスを実行するのに使うアプリケーションサーバーを、SAS Metadata Serverで定義されたものの中から選択します。
10. 「ターゲット」の下のドロップダウンリストからアプリケーションサーバーを選択します。
11. ［**Next**］をクリックします。

    パネルで、読み込んだストアドプロセスのSASコードを格納するためのソースコードリポジトリ（ディレクトリ）を、SAS Metadata Serverで定義されたものの中から選択します。
12. 「ターゲットパス」の下のドロップダウンリストからソースコードリポジトリを選択します。
13. ［**Next**］をクリックします。

    パネルに、［**実行**］をクリックした場合に実行される処理の概要が表示されます。
14. 表示された情報を確認し、問題がなければ［**実行**］をクリックします。
15. 読み込みプロセスの途中で、メタデータサーバーへの接続に必要なログイン情報の入力を求められる場合があります。管理者権限のあるログイン情報を入力し、［**OK**］をクリックします。

読み込みが完了したら、ストアドプロセスを読み込んだフォルダの下に「BIP Tree」というフォルダが表示されます。「BIP Tree」の下には「JMP Samples」というフォルダが表示されます。「JMP Samples」フォルダには、「Shoe Chart」と「Diameter」の2つのサンプルストアドプロセスが含まれています。

サンプルスクリプトstoredProcessHTML.jslおよびstoredProcessJSL.jslでは、サンプルストアドプロセスを読み込んだフォルダと一致するように、サンプルストアドプロセスへのパスを調整する必要があります。そうしないと、これらのスクリプトは正常に機能しません。

MATLABの使用

MathWorks Inc.のMATLABは、インタラクティブな作業環境で計算モデルを分析、視覚化できる製品です。MATLABは、Windows（32ビット、64ビット）、Macintosh OS X、Linux（64ビット）で使用可能です。Windows版とMacintosh版のJMPは、いずれもMATLABへの接続をサポートしています。

JMPスクリプト言語（JSL）を使って、次のようにMATLABと連携することができます。

- JSLスクリプト内からMATLABにステートメントをサブミットする
- JMPとMATLABの間でデータをやり取りする
- MATLABで作成されたグラフを表示する

JMP で MATLAB 関数を使用する方法については、『スクリプト構文リファレンス』の「MATLAB インテグレーション関数」の節を参照してください。

MATLAB のテキスト出力やエラーメッセージは、ログウィンドウに表示されます。

MATLAB のインストール

MATLAB は JMP と同じコンピュータにインストールされている必要があります。

JMP を Windows の 32 ビット版と 64 ビット版のどちらで使用しているかに応じて、該当するバージョンの MATLAB をインストールしてください。サポートされている MATLAB のバージョンについては、JMP Web サイト（http://www.jmp.com/system/）を参照してください。

Windows に MATLAB をインストールする

メーカーの指示に従って標準インストールを行ってください。

Macintosh に MATLAB をインストールする

Macintosh に MATLAB をインストールするには:

1. Automator アプリケーションを呼び出します。
   - ［Finder］>［フォルダへ移動］>［アプリケーション］を選択します。

     または
   - Automator アプリケーションをダブルクリックします。
2. 書類の種類として［アプリケーション］を選択し、［選択］をクリックします。
3. ［アクション］ビューで［ライブラリ ］ツリーを展開し、［ユーティリティ］を選択します。
4. ［ユーティリティ］アクションの列で、［シェルスクリプトを実行］をダブルクリックします。
5. ［シェルスクリプトを実行］の編集領域で、現在表示されているテキストを次のテキストで置き換えます。

```
export MATLABROOT="<MATLAB のインストール先のパス >"
export DYLD_LIBRARY_PATH="$MATLABROOT/bin/maci64:$MATLABROOT/sys/os/
  maci64:$DYLD_LIBRARY_PATH"
export PATH="$MATLABROOT/bin:$MATLABROOT/bin/maci64:$PATH"

/usr/bin/env

/usr/bin/open -a <JMP インストール先のパス >
```

6. シェルのオプションを「/bin/bash」に設定します。
7. 入力の引渡し方法を「引数として」に設定します。
8. **<MATLAB インストール先のパス >** を、実際の MATLAB のインストール先のパスで置き換えます（たとえば、/Applications/MATLAB_R2013a.app）。

9. <**JMPインストール先のパス**>を、実際のJMPのインストール先のパスで置き換えます（たとえば、/Applications/JMP Pro 11.app）。
10. Automationアプリケーションをデスクトップに保存します。

JMPによるMATLABのサポートを使用するには、このアプリケーションを使ってJMPを呼び出し、MATLABの実行時環境が確実にセットアップされるようにしなければなりません。

JMPによるMATLABの検出方法

JSLスクリプトからMATLABへの接続命令が出されるまで、JMPはMATLABを起動しません。MATLABを呼び出すJSLスクリプトを実行するとき、JMPは、オペレーティングシステムのPATH環境変数に基づいてソフトウェアの場所を特定します（たとえば、C:\Program Files\MATLAB\R2012a\）。

インストールのテスト

お使いのコンピュータで、JSLベースのスクリプトを使ったMATLABの操作が可能かどうかは、次のJSLスクリプトで検証できます。

1. 次のJSLスクリプトを実行します。

```
MATLAB Init( );
MATLAB Submit( "m = magic(3)" );
magicMat = MATLAB Get(m);
Show( magicMat );
MATLAB Term();
```

MATLAB関数M = magic(3)は、1～3^2の整数を使った3x3行列を戻します。この行列は、行の和と列の和が等しく、「**魔方陣**」と呼ばれます。

2. ［**表示**］>［**ログ**］を選択します。

ログウィンドウには次のような応答が表示されます。

```
m =

     8     1     6
     3     5     7
     4     9     2


magicMat =
[  8 1 6,
   3 5 7,
   4 9 2];
0
```

もしログウィンドウに次のようなメッセージが表示された場合は、その下の操作を行ってください。

システム上にMATLABがインストールされていません。

1. MATLABROOT という名前の新しい環境変数を追加し、値を C:\Program Files\MATLAB\R2012a\ または C:\Program Files (x86)\MATLAB\R2012a\ に設定します。

注：入力するパスは、MATLAB のインストール先のパスによって異なります。

2. MATLAB のパスが PATH 変数に含まれていることを確認します。
3. スクリプトを再実行し、JMP から MATLAB にアクセスできるかどうかを検証します。

R の操作

JSL を使ってできる R の操作は次のとおりです。

- JSL スクリプト内から R コードを R にサブミットします。
- JMP と R の間でデータをやり取りします。
- R で作成したグラフを表示します。

R のテキスト出力やエラーメッセージは、ログウィンドウに表示されます。

R のインストール

R は JMP と同じコンピュータにインストールされている必要があります。R は Comprehensive R Archive Network の Web サイトからダウンロードできます。

http://cran.r-project.org

JMP を Windows の 32 ビット版と 64 ビット版のどちらで使用しているかに応じて、該当するバージョンの R をインストールしてください。サポートされている R のバージョンについては、次の JMP Web サイトのシステム要件を参照してください。http://www.jmp.com/support/system_requirements_jmp.shtml

デフォルトの R インストールディレクトリを変更する

R_HOME が Windows のシステムレジストリに定義されていない場合、通常、JMP は R_HOME を次のようにみなします。

64 ビット版 R のインストール場所

Computer¥HKEY_LOCAL_MACHINE¥SOFTWARE¥R-code¥R¥InstallPath

32 ビット版 R のインストール場所

Computer¥HKEY_LOCAL_MACHINE¥SOFTWARE¥Wow6432Node¥R-code¥R¥InstallPath

デフォルトの R インストールの場所を変更するには、次のいずれかの方法で R_HOME 環境変数を定義します。

1. コントロールパネルを使って、システム環境変数内に変数を作成します。それには、まず、［**スタート**］>［**コントロールパネル**］>［**システム**］>［**システムの詳細設定**］を選択します。

2. ［**環境変数**］をクリックします。
3. システム環境変数のペインで［**新規**］をクリックします。
4. ［**変数名**］にR_HOMEとタイプします。
5. R .exeファイルのパスを入力します（C:¥Program Files¥R¥R-2.15.3など）。
6. ［**OK**］をクリックした後、再度［**OK**］をクリックし、システムプロパティのウィンドウを閉じます。

または

- 次のように、JSLのSet Environment Variable()関数を使って環境変数を作成します。

```
Set Environment Variable("R_HOME", "C:¥Program Files¥R¥R-2.15.3");
```

JMPによるRの検出方法

JSLスクリプトからRへの接続命令が出されるまで、JMPはRを起動しません。JMPは、Rをロードする必要が生じたときに次のような順序でWindowsコンピュータ上のRを探します。

1. 環境変数R_HOMEを検索します。

   見つかった場合、指定のディレクトリからRをロードします。

2. 環境変数R_HOMEが存在しない場合は、Windowsレジストリ内で次のキーの下にあるInstallPath値を調べます。

   HKEY_LOCAL_MACHINE¥SOFTWARE¥R-core¥R

   64ビットのコンピュータで32ビット版のJMPを実行している場合、InstallPath値は次のキーにあります。

   HKEY_LOCAL_MACHINE¥SOFTWARE¥WOW6432NODE¥R-core¥R

   InstallPath値が存在する場合、指定のディレクトリからRをロードします。

3. InstallPath値が存在しない場合は、Rが見つからないことを示すエラーメッセージが表示されます。

セットアップのテスト

お使いのコンピュータで、JSLベースのスクリプトを使ったRの操作が可能かどうかは、次のJSLスクリプトでテストできます。

```
R Init( );
R Submit("
   x <- 1:5
   x
");
R Term( );
```

ログには次のような出力が表示されます。

```
[1] 1 2 3 4 5
```

JMPからRへのインターフェース

JMPには、Rへのインターフェースが関数として用意されています。基本的には、まず、Rへの接続を開始し、次に、何らかの処理をR上で実行し、最後にRの接続を解除します。これらの関数は、Rの処理が正常に実行された場合は0、そうでない場合は、エラーコードを戻すのが普通です。Rの処理が正常に実行されなかった場合は、ログにメッセージが表示されます。ただし、`R Get( )`関数だけは、エラーコード以外の値を戻します。

RのJSLスクリプト可能なオブジェクトインターフェース

R接続の機能は、JSL関数としてだけではなく、オブジェクトへのメッセージとしても用意されています。`R Connect()`というJSL関数によって、R接続オブジェクトへの参照を取得できます。R接続オブジェクトに対して、以下で述べるメッセージを送ることができます。

JMPデータタイプとRデータタイプの相互変換

表14.4に、`R Send( )`関数でJMPからRに変数を送った場合に、JMPのデータタイプ（データ型）が、Rにおいて、どのような型に変換されるかを示します。リストの場合は、リスト内の要素ごとにデータタイプをチェックして、変換します。なお、入れ子になっているリストも、サポートされています。入れ子になったリストもサポートされます。

表14.4 R Send( )でのJMPデータタイプとRデータタイプの対応

JMPデータタイプ	Rデータタイプ
数値	実数（double）
文字列	文字列
行列	実数の行列
リスト	リスト
データテーブル	データフレーム
行の属性	整数
日付時間	日付と時間
時間差	時間差

例

```
R Init( );
X = 1;
R Send( X );
S = "Report Title";
R Send( S );
M = [1 2 3, 4 5 6, 7 8 9];
```

```
R Send( M );
R Submit( "
X
S
M
" );
R Term( );
```

表14.5に、に、R Get( )関数でRからJMPに変数を取得した場合に、Rのデータタイプ（データ型）が、JMPにおいて、どのような型に変換されるかを示します。リストの場合は、リスト内の要素ごとにデータタイプをチェックして、変換します。なお、入れ子になっているリストも、サポートされています。入れ子になったリストもサポートされます。

表14.5 R Get( ) でのJMPデータタイプとRデータタイプの対応

Rデータタイプ	JMPデータタイプ
実数（double）	数値
論理（ブール値）	数値 (0 | 1)
文字列	文字列
整数	数値
日付と時間	日付時間
時間差	時間差
因子（factor）	文字列のリスト、または、数値の行列
データフレーム	データテーブル
リスト	リストや行列を含んだリスト
行列	行列
数値ベクトル	行列
文字列ベクトル	文字列のリスト
グラフ	ピクチャー
Time Series（時系列分析）	行列

JMPスコープ演算子とR

R Send() 関数によって、JMPの変数をRに送った場合、Rオブジェクトの名前には、JMPの変数と同じ名前が付けられます。たとえば、dtというJMP変数をRに送ると、dtという名前のRオブジェクトが作成されます。コロンおよび2重コロンのスコープ演算子（: および ::）は、Rオブジェクト名に使用できないため、次のように変換されます。

- 1重コロンのスコープ演算子はピリオド（.）に置き換えられます。

  たとえば、nsref:dtをRに送ると、nsref.dtという名前のRオブジェクトが作成されます。

- 2重コロンのスコープ演算子（グローバル変数を指定）は無視されます。

  たとえば、::dtをRに送ると、dtという名前のRオブジェクトが作成されます。

R Send() でのR Name() の使用

R Send()のR Name()オプションには、有効なRオブジェクト名を引用符付き文字列で指定します。Rに送られたJMPオブジェクトは、指定の名前のRオブジェクトになります。たとえば、次のような設定が行えます。

```
R Send( jmp_var_name, R Name( "r_var_name") );
R Submit( "print(r_var_name") )
```

例

次の例は、Here名前空間内に変数x、グローバル名前空間内に変数y、そしてどの名前空間にも明示的に参照されない変数zを作成します。変数zは、Names Default To Here(1)がオンでない限り、デフォルトでグローバルに設定されます。これらの変数は、その後、Rに渡されます。

```
Here:x = 1;
::y = 2;
z = 3;

//R 接続を開始する
R Init();

//Here 変数を R に送る
//Here:x は R オブジェクトの Here.x となる
R Send( Here:x );
R Submit( "print(Here.x)" );
/* JMP ログの出力では、元の JMP 変数の参照である Here:x が使用される */

//::y は R オブジェクトの y となる
R Send( ::y );
R Submit( "print(y)" );

//R オブジェクトに別の名前をつけるには、R Name() オプションを使用する
R Send( Here:x, R Name( "localx" ) );
R Submit( "print(localx)" );
/*R Send() コマンドを R Name オプションとともに使用すれば、JMP 変数 "Here:x" に相当する R オブジェクト "localx" が作成できる。この場合も、ログには元の JMP 変数名が表示される */

//z は R オブジェクト z となる
R Send( z );
R Submit( "print(z)" );
```

トラブルシューティング

グラフの記録

Windows版のRにおいて、作成したグラフをグラフウィンドウに記録していくには、R Submit( )関数にて、次のRコードを実行してください。

```
windows.options( record = TRUE );
```

文字ベクトル

JMPにおける文字列のリストは、Rの文字ベクトルと同じではありません。文字列のリストをJMPからRに送った場合、それは、文字ベクトルではなく、文字列のリストに変換されます。これを文字ベクトルにするには、R関数のUnlistを使用してください。

```
R Init();
X = {"Character", "JMP", "List"};
R Send( X );
R Submit( "class(X)" );
/* R の出力は :
[1] "list"
*/

R Submit( "Y<-unlist(X)
     class(Y)" );
/* これでオブジェクト Y は文字ベクトルとなった R の出力は :
[1] "character"
*/

R Term();
```

要素の名前

Rのリストは、属性をもつことができ、リストの各要素に対して、特定の名前を与えることができます。属性をもつリストは、リスト内での通し番号がわからなくても、名前によって、要素にアクセスできます。

次の例では、Rにおいて、List()関数でリストを作成しています。リストでは、xとyという属性が要素に与えられています。このようなリストを、JMPで取得し、再度Rに戻した場合、属性が失われます。そのため、戻されたリストに対しては、「pts$x」といった指定によっては、要素にアクセスできなくなります。代わりに、「pts[[1]]」というように通し番号でアクセスしてください。

```
R Init();
R Submit("
   pts <- list(x=cars[,1], y=cars[,2])
   summary(pts)
");
```

```
JMP_pts = R Get(pts);

R Send(JMP_pts);
R Submit("
   Summary(JMP_pts)
");
R Term();
```

例

Rにデータテーブルを送る

次のプログラム例は、R接続を開始し、データテーブルをRに送り、それをログに印刷し、最後にR接続を閉じます。

```
R Init();
dt = Open("$SAMPLE_DATA/Big Class.jmp",invisible);
R Send(dt); // dtという開いたデータテーブルをR に送る
R Submit("print(dt)");
R Term();
```

Rでのオブジェクトの作成

次のプログラム例は、R接続を開始し、Rオブジェクトを作成し、そのオブジェクトをJMPで取得し、最後にR接続を閉じます。

```
R Init();
R Submit(
   "
   L3 <- LETTERS[1:3]
   d <- data.frame(cbind(x=1, y=1:15), Group=sample(L3, 15, repl=TRUE))
"
);
R Get( d ) << NewDataView;
R Term();
```

Rの関数とグラフの使用

次のプログラム例は、R接続を開始し、Rにおいて正規密度関数のグラフを描きます。次に、そのグラフのイメージをRから取得して、JMPで表示します。最後に、R接続を閉じます。

```
R Init();
R Submit( "\[plot(function(x) dnorm(x), -5, 5, main = "Normal(0,1) Density") ]\" );
picture = R Get Graphics( "png" );
New Window( "Picture", picture );
Wait( 10 );
R Term();
```

R内で簡単な行列を追加する

次のプログラム例は、R接続を開始し、まず、1つの行列をJMPからRに送ります。次に、Rで、もう1つの行列を作成します。そして、この2つの行列を足し合わせた行列をJMPに送り、最後にR接続を閉じます。

```
R Init();
X = J( 2, 2, 1 );
R Send( X );
R Submit(
   "
   X                                        #X をログに印刷する
   Y <- matrix(1:4, nrow=2, byrow=TRUE)  #2x2 の行列オブジェクト Y を作成する
   Y                                        #Y をログに印刷する
   Z <- X + Y                               # 行列オブジェクト Z は X と Y を 1 つにしたもの
"
);
Z = R Get( Z );
R Term();
Show(Z);
```

ブートストラップの例

Rを用いたブートストラップの例として、サンプルスクリプトフォルダにあるJMPtoR_bootstrap.jslをご覧ください。

このスクリプトは、Rのライブラリを利用して、JMPにおいてブートストラップを実行します。

このスクリプトは、まずユーザに対し、分析対象の変数を指定するためのウィンドウを、JMPにおいて表示します。このウィンドウで、信頼区間を求めたい統計量も選択します。その後、JSLのRインターフェースを使って、データがRに送られます。

そして、Rのbootパッケージにおける`boot()`関数および`boot.ci()`関数によって、各ブートストラップ標本の統計量とブートストラップ信頼区間が計算されます。

結果はJMPに戻され、JMPの「一変量の分布」プラットフォームを使って表示されます。

Excelの使用

JMPが提供しているExcelアドインのうち、「JMPでのプロファイル」の機能は、スクリプトでも実行できます。しかし、データを転送する「JMPへの転送」は、スクリプトでは実行できません。「JMPでのプロファイル」の基本的な構文は次のとおりです。

```
excel_obj = Excel Profiler(
   Workbook( "excel_workbook_path" ),
   Model( "name_of_model" )
);
```

Model引数はオプションです。ワークブックのみを指定し、モデルの指定を省略した場合は、モデルを選択するよう促されます。モデルはワークブックのどこにあっても検出されるので、ワークシートを選択する必要はありません。

上記のような方法で作成したオブジェクトに、次のメッセージを送ると、予測プロファイルが描かれます。たとえば、次のような設定が行えます。

```
excel_obj <<Prediction Profiler( 1 );
```

OLEオートメーション

JMPの大部分はOLEオートメーションを使って操作することができます。JMPのオートメーションについては、「JMP¥11¥Documentation\ja」フォルダにある「Automation Reference.pdf」を参照してください。11このドキュメントでは、Visual BasicおよびMFCを使ったVisual C++によってJMPを自動化する方法を説明しています。また、Visual BasicやVisual C++のようなオートメーションクライアントで利用できるJMPのメソッドやプロパティについて詳しく説明しています。

「Samples¥Automation」フォルダには、JMPのオートメーション機能をVisual Basic .Net、Visual C# .Net、およびVisual C++ .Netでプログラミングする例があります。

Visual Basicを使ったJMPのオートメーション

JMPアプリケーションの起動

JMPのオートメーションの第一歩は、JMPの起動です。ここで、JMPのメソッドやプロパティを利用する上でどのようなリソースが役立つかに注目することが重要です。JMPに用意されているタイプライブラリを通じて、Visual Basic（以下VB）のようなオートメーションクライアントは、JMPが公開しているメソッドやプロパティのリスト、および各メソッドで必要となる引数を表示できます。このライブラリはJMP.TLBというファイルで提供されています。

JMPのタイプライブラリをVBから利用できるようにするには、次の手順を行います。

1. VB において、[**プロジェクト**] > [**参照の追加**] を選択します。すると、VB から参照可能なアプリケーションのリストが表示されます。リストにJMPがないときは、[**参照**] を選択し、ファイルウィンドウで.tlb（タイプライブラリ）の場所を指定します。JMPディレクトリ内で、JMPのタイプライブラリのアイコンを探します。このライブラリを選択し、[**OK**] ボタンをクリックします。
2. 次に、VB において、[**表示**] メニューなどから、[**オブジェクトブラウザー**] を選択して、オブジェクトブラウザーを起動します。そして、リストからJMPを選択します。

これでJMPのオートメーションクラスと定数を参照できます。クラスを選ぶと、そのクラスで利用可能なメソッドがオブジェクトブラウザの右側のリストボックスに表示されます。メソッドを選択すると、ウィンドウの下の方に、短い説明が表示されます。説明には、そのメソッドで使われる引数がリスト表示されます。メソッドが特定の引数群を必要とするときには、通常、特定のアクションを表記する定数が使われます。

これでタイプライブラリの情報にアクセスできるようになり、JMPのクラスのインスタンスを生成するのに必要なコードが書けます。これはCreateObjectを使って行います。VBのプロジェクトのグローバル宣言で、*JMPJ.Application*タイプの変数を作成します。次のように指定します。

```
Dim MyJMP As JMP.Application
```

この際に、他の変数の作成も行います。DataTable、Distrib、Oneway、およびJMPDocがその例です。それぞれ、JMP.DataTable、JMP.Distribution、JMP.Oneway、およびJMP.Documentで指定できます。

JMPのセッションを作成して、セッションを表示し、データテーブルを読み込むには、以下のコードをVBのスクリプトに加えます。

```
Dim JMPDoc As JMP.Document
MyJMP = CreateObject("JMP.Application")
MyJMP.Visible = True
Set JMPDoc = MyJMP.OpenDocument("C:\Program Files\SAS\JMP\<バージョン番号>\
   Samples\Data\Big Class.jmp")
```

Dimでは変数の型を宣言しています。ただし、この宣言は、VBのプロジェクトの一般的な宣言の中に入れる必要があります。そうしないと、処理の終わりでこの変数が有効範囲外となり、JMPオブジェクトは破棄されてしまいます。

オートメーションのガイドラインで定められているように、デフォルトではJMPは表示されません。このため、上のコードにあるように、処理の最初のステップでJMPを表示させる必要があります。

分析の開始

これでデータテーブルが開き、分析を開始し、操作できるようになりました。まず、各分析のオブジェクトを生成します。次に、分析に必要な引数を指定します。オプションを設定することもできます。指定が終わったら、分析のプラットフォームを開始します。その後、分析オブジェクトで追加オプションの処理が行われます。

```
Dim Oneway As JMP.Oneway
Oneway = JMPDoc.CreateOneway
Oneway.LaunchAddY(" 身長 ( インチ )")
Oneway.LaunchAddX(" 年齢 ")
' 起動前にオプションを指定
Oneway.Quantiles(True)
' 最初の分析出力を作成
Oneway.Launch()
Oneway.MeansAnovaT(True)
Oneway.MeansStdDev(True)
Oneway.UnequalVariances(True)
Oneway.NormalQuantilePlot(True)
Oneway.SetAlpha(0.05)
Oneway.Save(JMP.OnewaySaveConstants.oscCentered)
Oneway.Save(JMP.OnewaySaveConstants.oscStandardized)
Oneway.CompareMeans(JMP.OnewayCompareConstants.occAllPairs, True)
Oneway.CompareMeans(JMP.OnewayCompareConstants.occEachPair, True)
```

最初にDocumentクラスのCreateOnewayをコールして、分析オブジェクトを生成します。次にXの列とYの列を指定し、Launchをコールして実際の一元配置分析を生成します。各分析プラットフォームのオブジェクトはそれぞれ別のメソッドで生成されますが、これはオブジェクトブラウザのDocumentで確認できます。多くの場合、オブジェクトのLaunchの前にオプションを指定できるため、分析の出力はすでに設定されたオプションに従ったものになります。この例では、オプション処理の多くは分析の起動後に行われています。これはディスプレイの中のオプションのポップアップメニューに表示されます。ご覧のとおり、メソッドの多くは、メニューから設定するかのような簡単なオプション設定です。SetAlphaには1つの引数が指定されています。これは、オートメーションの実行中に入力を要求するウィンドウが表示されないようにするためです。CompareMeansは引数を2つとり、1つは比較の種類、もう1つはオン／オフの切り替えを指定します。Saveは予め定義された定数（オブジェクトブラウザで確認できます）をとり、一元配置分析で何を保存するかを指定します。

分析手法の多くは上のように処理されますが、Bivariate（二変量）のような手法には、コールされた時点で追加のオブジェクトを生成するものもあります。たとえば、次のように指定します。

```
Set Fit = Bivar.FitLine
Fit.ConfidenceFit (True)
Fit.ConfidenceIndividual (True)
```

ここで、FitLineはFit型のオブジェクトを生成します。このオブジェクトは固有のメソッドとプロパティを持ち、それを操作することができます。FitLineで生成された新しいオブジェクトは、その変数が参照可能な範囲内にある間だけ操作できることに注意してください。

あるメソッドから、自動制御のできるオブジェクトが生成された場合、オブジェクトブラウザーでこれを確認できます。オブジェクトブラウザーには、FitLineの戻り値はFit型であると示されます。

これは、shortやBSTRのように予め定義された型ではないので、オブジェクトであると推測できます。オブジェクトブラウザをさらに見ていくと、Fitがオブジェクト型であることがわかります。このように、FitLineによりオブジェクトが生成されることが確認でき、またFitがサポートするメソッドもわかります。

データテーブルを作成し、値を埋め込む

新しいデータテーブルは、（適当な名前をつけて）ApplicationオブジェクトのNewDataTableというメソッドで作成できます。ファイル名は作成時に割り当てられます。NewColumnメソッドでは列オブジェクトが戻され、これは行を追加していく限りは保持しておく必要があります。デフォルトでは20行が作成されます。各セルに値を埋め込むにはSetCellValメソッドを使います。行を加えるにはAddRowsを使います。以下はその例です。

```
Dim Col As Object
DT = MyJMP.NewDataTable("c:/test.jmp")
Col = DT.NewColumn("列1", JMP.colDataTypeConstants.dtTypeNumeric, 0, 8)
DT.Visible = True

' テーブルに値を埋め込む前に行を加える
DT.AddRows(20, 0)

'1.5刻みで増える値をセルに埋め込む
```

```
For i = 1 To 10
   Col.SetCellVal(i, i * 1.5)
Next i
DT.Visible = False
For i = 11 To 20
   Col.SetCellVal(i, i * 1.5)
Next i
DT.Visible = True

' テーブルの末尾に 5 行加える
DT.AddRows(5, 0)
'2 行目の後に 5 行加える
DT.AddRows(5, 2)

' すでに指定してあるファイル名でデータテーブルを保存する
DT.Document.Save()

' テーブルの 1 行目から 3 行目までだけを取り出したテーブルを作るには
' 次のようにする
' 注： Datatable のメンバー関数 AddToSubList を使って列をリストに加えれば、
' 特定の列だけを取り出したテーブルを作ることもできる
Dim NewDT As JMP.DataTable
Dim DTDoc As JMP.Document
DT.SelectRows(1, 3)
NewDT = DT.Subset

' 新しいテーブルを保存
DTDoc = NewDT.Document
DTDoc.SaveAs("C:/MySubset.jmp")
```

プログラム例

「Samples¥Automation」フォルダには、JMPのオートメーション機能をVisual Basic .Net、Visual C# .Net、およびVisual C++ .Netでプログラミングする例があります。Visual Basicのプログラムには、Visual Studio 2005以降が必要です。

「Analysis」では、ほとんどすべてのJMPプラットフォームの単純なオートメーションの例を示しています。例の中のコードはプラットフォームの機能をテストするもので、意味のある統計解析を行うものではありません。このプログラム例の目的は、オートメーション用コードの作成方法を紹介することです。JMPのタイプライブラリがVBのプロジェクトから参照できるようにしておくと便利です。この方法については、このマニュアルの最初の節を参照してください。オートメーションを行うプラットフォームにあるメソッドやプロパティを確認できます。

「Data Table」では、データテーブルのオートメーションに利用できるメソッドの例を示しています。意味のある出力を得ようとしているわけではありません。

「Time Import」では、JMPでテキストファイルを読み込みデータテーブルにする際に必要な手順を示しています。この処理を行うと、「Data Table」での例のようにデータテーブルを操作できるようになり、また「Analysis」での例のように、データについての分析ができるようになります。

「ODBC Demo」では、ODBCアクセスを使ってdBaseのファイルをJMPに読み込む簡単な例を示しています。

「Word Demo」では、JMPのレポートからグラフの部分をクリップボードにコピーし、Microsoft Wordの文書に挿入するためのコマンドを示しています。

「FitModel」と「DOE」の例では、JMPのそれぞれ該当する領域に固有の演算子を示しています。これらのプラットフォームの演算子は、他のプラットフォームとは若干異なっています。

サンプルのプログラム例では、データファイルが「Samples¥Data」ディレクトリにあることを前提としています。データファイルの場所を移動した際は、VBのプログラム例の中にあるパスの部分を変更してください。

このマニュアルにあるVisual Basicのコード例と、サンプルプログラムの例に違いがある場合は、サンプルプログラムのコードのほうを参考にしてください。

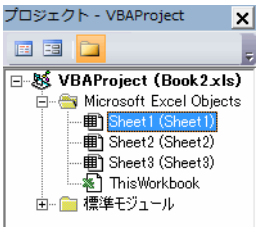
例: Excel 2007を使ったJMPのオートメーション

この例では、Excel 2007ワークシート内のマクロを使ってJMPを自動化します。マクロコードは、Visual Basicで記述されます。このマクロは、Excelワークシートが開かれたときに、JMPを表示して開始します。次に、ODBCオートメーションインターフェースを使って、Excelワークシートを読み込みます。ワークシートのデータがJMPに読み込まれると、個々のワークシートのセルに加えた変更がJMPに送られ、JMPのデータテーブルが変更されます。

初めてExcelで行の値が変更されたときに、JMPにより管理図が生成されます。それ以降Excelのワークシートに変更が加えられると、管理図も変更されます。この例では、Excelのデータが変更されると、JMPデータテーブルが更新されるようにします。そして、このJMPデータテーブルと管理図を連動されて、変更が反映されるようにします。また、Excelワークシートが5回変更されるごとに、管理図の.PNGファイルを生成するJMPのメソッドを呼び出します。PNGファイルは、WebブラウザなどのJMP以外のアプリケーションでも見ることができます。最後に、Excelワークシートを閉じるときに、オートメーション機能によって、JMPもシャットダウンします。

Microsoft Excelを開くところから始めてみましょう。Excelのブックで使うVisual Basicスクリプトを作成するには、［**開発**］リボンから［**Visual Basic**］を選びます。Visual Basic Editorが別のウィンドウに表示されます。Visual Basic Editorの左側には、「**VBAProject**」というペインがあります。ブックと、Visual Basicコードが関連付けられているシートが表示されています。

図 14.5 Excel 用 VBA プロジェクト



ブック用に記述されたコードは、通常ブック内のすべてのシートで動作します。

以下の例では、3 つのセクションに分けてコーディングしていきます。まず最初に、module1.bas ファイルでいくつかの変数をグローバル変数として宣言します。これで、宣言した変数が他のコードのモジュールで参照されるようになります。VBA のプロジェクトアイコンでコンテキストクリックし、[**挿入**] > [**標準モジュール**] を選ぶと、モジュールを Visual Basic プロジェクトに挿入できます。以下のコードをモジュールに入力します。このコードでは、JMP アプリケーションのインスタンス、JMP データテーブル、およびドキュメントが開いているかどうか確認するためのフラグを宣言しています。

```
Public MyJMP as JMP.Application 'JMP アプリケーションオブジェクト
Public DT As JMP.DataTable 'JMP アプリケーションオブジェクト
Public DocOpen as Boolean '「JMP テーブルが開いている」ことを示すフラグ
```

次のセグメントでは、Excel ワークシートのセルが変更されると、JMP が更新されるようにします。Excel では、セルが変更、削除、追加されると、必ず Worksheet_change イベントが発生します。そこで、Worksheet_change イベントが起きたときに、このセグメントを呼び出すようにします。

Excel の VBA プロジェクトブラウザには、現在ブックに含まれているワークシートが表示されます。以下のコードを、JMP にデータを送るワークシートに入力します。VBA プロジェクトウィンドウでワークシートアイコンをダブルクリックし、そのワークシートのコードのウィンドウを表示します。

```
Private Sub Worksheet_change(ByVal Target as Range)
   Dim Col as JMP.Column

   If(DocOpen) Then
      If(Target.Row = 1) Then
         Return
      EndIf
      If(DT.NumberRows < Target.Row - 1) Then
         DT.AddRows Target.Row - DT.NumberRows - 1, Target.Row
      EndIf
      If(Not IsArray(Target.Value) And Not IsEmpty(Target.Value)) Then
         Set Col = DT.GetColumnByIndex(Target.Column)
         Col.SetCellVal Target.Row - 1, Target.Value
      EndIf
   EndIf
End Sub
```

このコードでは、まずJMPでデータテーブルが開いていることを確認します。最初の行が変更されたときには、この行はJMPの列名なので無視されます。そして、Excelで列名が変更されても、その変更はJMPには**反映されません**。見出しを変更するコードをここに挿入することができますが、この例では省略しています。

次に、変更された行が、データテーブルでJMPが現在確認している行番号を超えている場合には、AddRowsメソッドが呼び出されて行が追加されます。

最後に、処理が1つの値について行われていて、削除を示していない場合には、JMPデータテーブルのセルの値はWorksheet_Changeに渡された値に変更されます。

メインのモジュールは、ブックに関するものです。VBAプロジェクトブラウザでは、ブックコード領域には通常ThisWorkbookという名前が付いていますが、この名前は簡単に変えられます。次のコードをこの領域に入力します。

```
' ブックの全サブルーチンにアクセスできる Public（グローバル変数）宣言
Public Counter As Integer '5 回変更されるたびに管理図を更新するためのカウンタ
Public JMPDoc As JMP.Document 'JMP ドキュメントのインスタンス
Public CChart As JMP.ControlChart ' 管理図のインスタンス
Public ChartOpen as Boolean ' 管理図が開いているかどうかを設定するフラグ
Public DB As AUTODB

' ブックを閉じる前に JMP をシャットダウンする
Private Sub Workbook_BeforeClose(Cancel as Boolean)
   DocOpen = False
   MyJMP.Quit
End Sub

' ［ファイル］>［開く］でブックを開くと、オートメーションの JMP をロードする
Private Sub Workbook_Open()
   Set MyJMP = CreateObject("JMP.Application") 'JMP のインスタンスを作成する
   MyJMP.Visible=True ' この JMP のインスタンスを表示させる
   Counter = 0 ' 変更をカウントするカウンタを初期化する
   DocOpen = False ' まだドキュメントを開いていない
   ChartOpen = False ' チャートも開いていない

   'Excel ワークシートを指すようにこのパスを変更する
   Set DB = MyJMP.NewDatabaseObject
   DB.Connect ("DSN=Excel Files;DBQ=C:\Book2.xls;")
   Set DT = DB.ExecuteSQLSelect("SELECT * FROM ""Sheet1$""")
   DB.Disconnect
   Set JMPDoc = DT.Document
   DocOpen = True ' ドキュメントが開いていることを示すフラグを設定する
End Sub

' ここが最も重要な部分である。
' 最初のデータが変更されたら、管理図を作成する。
'Excel ワークシートのセルに 5 回変更が加えられるたびに、管理図の PNG ファイルを作成する。
```

```
Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Source As Range)
   Counter = Counter + 1
   ' 要素が 5 つ更新されるたびに、管理図を PNG ファイルに保存する
   If (Counter Mod 5 = 0 Or Counter = 1) Then

      ' 管理図が作成されていなかったときには作成する
      If Not (ChartOpen) Then
         Set CChart = JMPDoc.CreateControlChart ' 管理図を作成する
         CChart.LaunchAddProcess "列 1" ' 列を追加する
         CChart.LaunchAddSampleUnitSize(5)
         CChart.LaunchSetChartType jmpControlChartVar
         CChart.Launch ' 管理図を起動する
         ChartOpen = True ' 管理図が開いていることを示すフラグを設定する
      EndIf
      CChart.SaveGraphicOutputAs "C:¥ControlChart.png", jmpPNG
   EndIf
End Sub
```

Excelのテーブルが最初にロードされるときに、Workbook_Openサブルーチンが呼び出されます。このサブルーチンは、変数を初期化し、JMPを起動して、現在Excel 2007にロードされているものと同じExcelファイルを（ODBCを使って）開くようにJMPに指示します。JMPでは、Excelファイルがファイルとしてではなく、データベースオブジェクトとして開かれます。これは、JMPでは別のアプリケーションですでに開いているファイルを開けないために必要な対処です。

ユーザがブック内のワークシートでセルのデータを変更するたびに、Workbook_Changeイベントが生成されます。このコード例では、ブックにあるアクティブなワークシートは1つだけと仮定しています。初めてユーザがワークシートのセル値を変更したときに、Workbook_Changeサブルーチンは、現在のデータテーブルを使ってJMPに管理図を作成します。

この例では、Workbook_changeサブルーチンは、管理図出力用のPNGグラフィックファイルも作成し、ブックに5回目の変更が加えられるたびにディスク上のそのファイルを更新します。これは、ExcelイベントとJMPオートメーションを一緒に使って結果を出力する方法をわかりやすくするためです。

最後に、Excelブックのウィンドウが閉じる直前に、Workbook_BeforeCloseサブルーチンが呼び出されます。このサブルーチンは、JMPにウィンドウを閉じるよう指示するコードです。

これらのメソッドにはいくつかの制限があることに注意してください。データに関して実行されるアクティビティが追加または変更の場合にだけ、このコード例はうまく機能します。ExcelのWorksheet_Changeイベントには、Excelが備えているレポート機能についてかなり制限があります。特に、削除、ドラッグ&ドロップ、ブロック反復を使用する必要がある場合、JMPのデータテーブルをセル単位で更新することが難しくなります。

このような場合には、必要な処理を行うコードを直接記述する方がおそらく賢明です。その1つの方法としては、ある一定の回数変更されるたびに、JMPにデータを再読み込みする方法があります。次にその例を示します。

```
Private Sub Workbook_SheetChange(ByVal Sh as Object, ByVal Source as Range)
   Counter = Counter + 1
   If (Counter Mod 10 = 0) Then
      ' 以前の管理図のテーブルが開いている場合は、まずそれを閉じる
      If(DocOpen) Then
         JMPDoc.Close False, ""
         CChart.CloseWindow
      EndIf

      Set JMPDoc = MyJMP.OpenDocument(InstallDir + "C:¥BOOK1.XLS")
      Set DT = JMPDoc.GetDataTable
      DocOpen = True

      ' 管理図を作成する。
      ' これは、「列 1」のデータに入る。
      '5 つ以上の値が変更されたとき、
      'JMP は新しい管理図を生成し、それを
      'PNG ファイルとしてディスクに保存する。
      'PNG ファイルは Internet Explorer で表示可能。

      Set CChart = JMPDoc.CreateControlChart
      CChart.LaunchAddProcess "列 1"
      CChart.LaunchAddSampleUnitSize(5)
      CChart.LaunchSetChartType jmpControlChartVar
      CChart.Launch
      CChart.SaveGraphicOutputAs "C:¥ControlChart.png", jmpPNG
   EndIf
End Sub
```

この例では、Excelのブックが10回変更されるたびに、データを再読み込みしています。まず、このコードでは、既存のJMP管理図とデータテーブルを削除しています。次に、新しいデータをロードして管理図を作成しています。

このサンプルコードは、少ないデータ量の場合に最もうまく動作します。この方法ではJMPにテーブルを再読み込みしているので、膨大なExcelファイルを読み込むときには効率的ではありません。

Visual C++を使ったJMPのオートメーション

CやC++を使ってオートメーションクライアントを作成するのは、時間のかかる退屈な作業です。しかし、Microsoft Visual C++のMFCの機能を使えば、作業は格段に楽になります。自動化サーバーアプリケーション（この場合はJMP）を起動できる状態になるまでには、いくつかの手順を踏む必要があります。「Samples/Automation/Visual C++ Sample」ディレクトリにあるAutoClientアプリケーションにはいくつかのコードがあり、それらを見ると、作業の開始方法がわかります。Microsoftのサンプルアプリケーションである「CALCDRIV」にも、MFCベースのオートメーションクライアントが示されています。「CALCDRIV」は、通常、Visual C++やMSDNのCDに含まれています。

AutoClientでは、JMPの起動方法と、二変量の分析やデータテーブルを扱う方法を示しています。このサンプルは、Visual Basicの他のサンプルに比べて小さいですが、すべてのオートメーションを呼び出す仕組みは、ここにある二変量やデータテーブルを使用した例と同じです。以下の手順はVisual C++ Version 5.0のユーザインターフェースに準拠しています。

JMPのオートメーションの手順

1. App Wizardを使うか、または手動でアプリケーションを生成します。まずOLEオートメーションのサポートを指定します。自分で作ったアプリケーションを自動制御するわけではなくても、OLEのヘッダと初期化コードをインクルードする必要があります。既存のアプリケーションを作り変える場合には、必ずOLEサポートを含める必要があります。このためには、通常、アプリケーションにafxole.hをインクルードして、アプリケーションのInitInstanceルーチンで`AfxOleInit()`をコールします。詳しい方法については、MFC OLEのマニュアルを参照してください。
2. Class Wizardを立ち上げ、オートメーションのタブを選択します。[**クラスの追加**]のドロップダウンメニューから[**タイプライブラリから**]を選びます。JMPインストールディレクトリにある「**JMP.TLB**」を選択します。「JMP.TLB」を選択します。
3. プロジェクトで使うクラスを確認するためのダイアログボックスが表示されます。どのオブジェクト（およびインターフェース）を選べばいいのかがわからない場合は、Shiftキーを押しながらクリックしてすべてを選択します。ClassWizardがインターフェースのスタブとヘッダ情報を生成するファイルの名前を選びます。ClassWizardは、MFC ColeDispatchDriveクラスに基づいたラッパークラスを生成します。これで、技術的な詳細を知らなくても、OLEオートメーションの関数**Invoke**に簡単にアクセスできるようになります。[**OK**]を選択します。クラスウィザードは2つのファイル（.hと.cpp）を作成します。たとえばViewクラスを実装するファイルのように、.cppファイルでJMPのオートメーションオブジェクトが使われる場合は、必ず.hファイルをインクルードしてください。
4. ワークスペースのClass Viewには、読み込まれたインターフェースクラスが表示されているはずです。このClass Viewを使うと、各クラスのメソッドとプロパティを調べることができます。
5. JMPを起動するには、オートメーションのセッションの間有効となる*`IJMPAutoApp`*型の変数を定義します。次いで、この変数について`CreateDispatch`をコールし、単独の引数としてJMP ProgID（「JMP.Application」）を渡します。この時点でコードが実行され、JMPが起動します。
6. ステップ5で生成されたJMPオブジェクトについて`SetVisible(TRUE)`をコールします。JMPの実行状況を見る必要がなければこれは不要です。ただし、デバッグの際には必要になります。
7. これでJMPのアプリケーションオブジェクトを使ってさらにオブジェクトを生成できます。できたオブジェクトからさらにオブジェクトを生成することもできます。まず、データテーブルをロードします。既存のJMPデータテーブルをロードするには、ステップ5で作成されたJMPオブジェクトについて`OpenDocument`メソッドをコールします。成功すれば、メソッドはディスパッチポインタを戻します。このポインタは`AttachDispatch`メソッドを使ってIJMPDoc型のオブジェクトに付加できます。
8. `IJMPDoc`オブジェクトには、「分析」プラットフォームと「グラフ」プラットフォームを起動するメソッドがあります。分析のオブジェクトを生成し、ディスパッチポインタを貼り付けると、分析で使われるデータテーブル列が指定でき、分析を起動できます。いったん分析を起動すると、その分析の種類に合わせたプロパティやメソッドを使った操作ができるようになります。以下は、ステップ5～8を説明するアプリケーション例のコードです。

プログラム例

```
// 注 : この例ではエラー処理はなし
IJMPAutoApp m_DispDriver;
IJMPDoc m_Doc;
IAutoBivar m_Bivar;
IAutoFit m_FitLine;

// IJMPAutoApp のインターフェース設定（jmpauto.h から取得）を使った
// 初期ディスパッチドライバを生成する
m_DispDriver.CreateDispatch("JMP.Application");

if (m_DispDriver)
{
   // JMP がうまく起動されたら、それを表示させる
   m_DispDriver.SetVisible(TRUE);

   // データテーブルをドキュメントとして開く。ドキュメントのインターフェース
   // ポインタが戻され、これが IJMPDoc インターフェース設定を使った
   // Doc ディスパッチドライバクラスに付加される
   m_Doc.AttachDispatch(m_DispDriver.OpenDocument(
      "C:¥¥JMPDATA¥¥BIGCLASS.JMP"));
}
   // 最初に Doc インターフェースについて CreateBivariate をコールし
   // 二変量の分析へのディスパッチオブジェクトを作成する // すでに以前のディスパッチインター
   フェースが m_Bivar にある場合、
   // 以前のディスパッチインターフェースが m_Bivar にすでにある場合、
   // MFC はそれを AttachDispatch で解放する。
   m_Bivar.AttachDispatch(m_Doc.CreateBivariate( ));

   // 分析対象の列として「身長 ( インチ )」と「体重 ( ポンド )」を加える
   m_Bivar.LaunchAddX(" 身長 ( インチ )");
   m_Bivar.LaunchAddY(" 体重 ( ポンド )");

   // 分析を起動する
   m_Bivar.Launch( );

   // FitLine を作成する。Fit はオートメーション可能なので、
   // FitLine( ) から戻るディスパッチポインタを DispatchDriver オブジェクトに付加する
   m_FitLine.AttachDispatch(m_Bivar.FitLine( ));

   // 別のあてはめを実行する。この例では、あてはめオブジェクトの自動化を
   // 行っていないが、自動化はサポートされている
   m_Bivar.FitPolynomial(3.0);
   m_Bivar.FitSpline(1000.0);
```

```
// 最初の FitLine オブジェクトを操作する
m_FitLine.ConfidenceFit(TRUE);
m_FitLine.ConfidenceIndividual(TRUE);
```

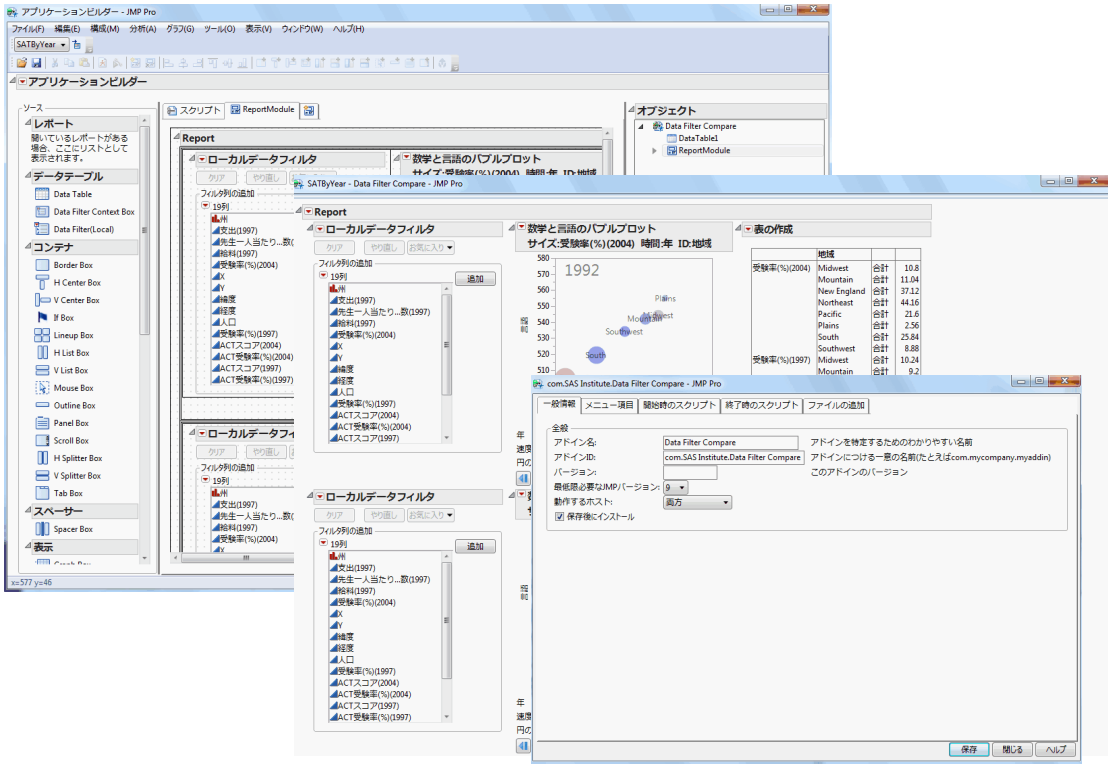
第 15 章

アプリケーションの作成と共有
アプリケーションビルダーとアドインビルダー

JMP では、プラットフォームによる統計分析だけでなく、自分の目的に沿ったアプリケーションも作成できます。アプリケーションを作成して、ルーチン作業（たとえば、決められたデータに対し、「一変量の分布」と「モデルのあてはめ」を毎日実行するなど）を自動化できます。アプリケーションビルダーを使えば、複数の分析結果を、同一のウィンドウに配置できます。アプリケーションビルダーでは、ドラッグ＆ドロップ操作が行え、大量のスクリプトを書くことなしに、アプリケーションを作成できます。

アドインビルダーでは、JMP にインストールできるアドインを簡単に作成できます。アプリケーションビルダーでアプリケーションを作成し、アドインビルダーでアドインとして保存すれば、アプリケーションを簡単に作成および配布できます。

図15.1 アプリケーションビルダーとアドインビルダー



目次

アプリケーションビルダー 587
例 587
アプリケーションビルダーの用語 589
アプリケーションの設計 591
「アプリケーションビルダー」ウィンドウ 591
赤い三角ボタンのオプション 592
アプリケーションの作成 594
アプリケーションの編集または実行 605
アプリケーションの保存オプション 606
JMPアドイン 609
アドインビルダーを使ったアドインの作成 609
アドインの編集 613
［アドイン］メニューからのアドインの削除 614
アドインのアンインストール 614
アドインの共有 614
JSLを使ったアドインの登録 615
手動でのアドインの作成 615

アプリケーションビルダー

アプリケーションビルダーは、ボタンやグラフなどのオブジェクトを、ドラッグ＆ドロップ操作によって、ウィンドウに配置できます。つまり、オブジェクトを配置するためのスクリプトを記述する必要はなく、各オブジェクトの機能を制御するスクリプトを記述するだけです。

また、JMP スクリプト言語（JSL）でのプログラミング経験が豊富なユーザは、アプリケーションビルダーによって、開発の生産性を高めることができます。アプリケーションビルダーでオブジェクトを配置し、自動作成されたスクリプトを編集します。独自の機能をスクリプトで記述することにより、目的に合ったアプリケーションを開発できます。

例

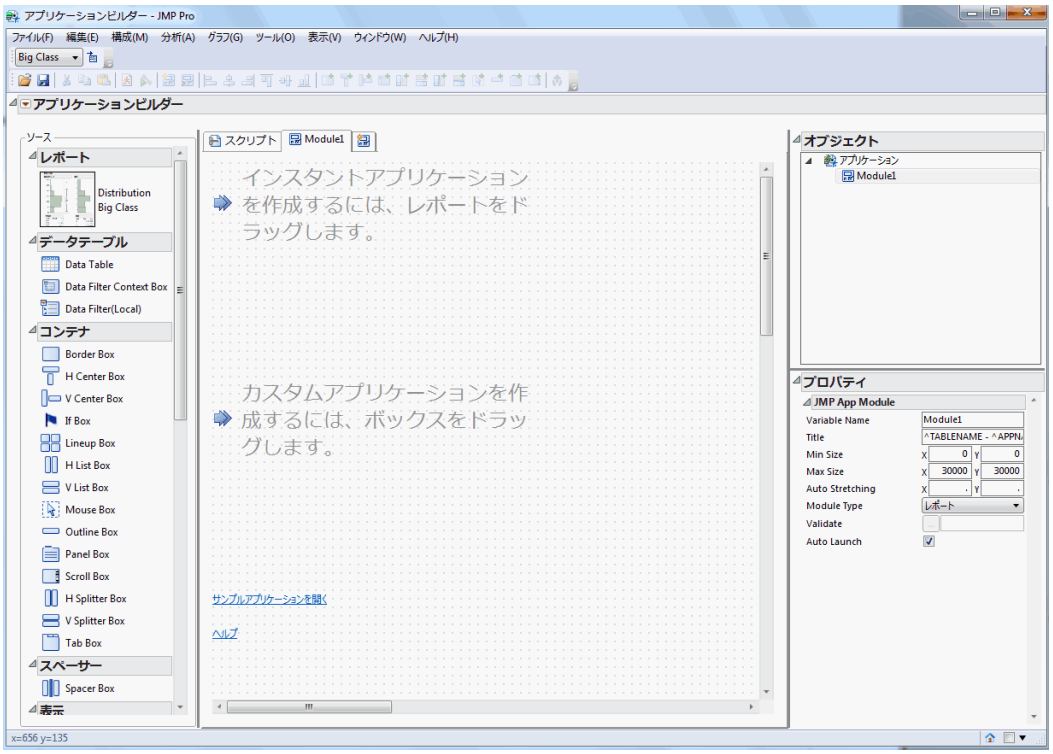
単純なものであれば、アプリケーションビルダーによって、アプリケーションを簡単に作成できます。単純なアプリケーションの1つには、レポートやグラフなどの複数のオブジェクトを、同一のウィンドウに配置したものがあります。

ここでは、「一変量の分布」レポートを表示するアプリケーションを作成します。この例では、起動ウィンドウで変数を選択しなくても、事前に指定された変数によって処理が実行されます。

1. 「Big Class.jmp」データテーブルを開きます。
2. **「一変量の分布」**テーブルスクリプトを実行してレポートを生成します。
3. **［ファイル］＞［新規作成］＞［アプリケーション］**（Macintosh の場合は**［ファイル］＞［新規］＞［アプリケーションの新規作成］**）を選択します。

   「アプリケーションビルダー」ウィンドウが開きます。

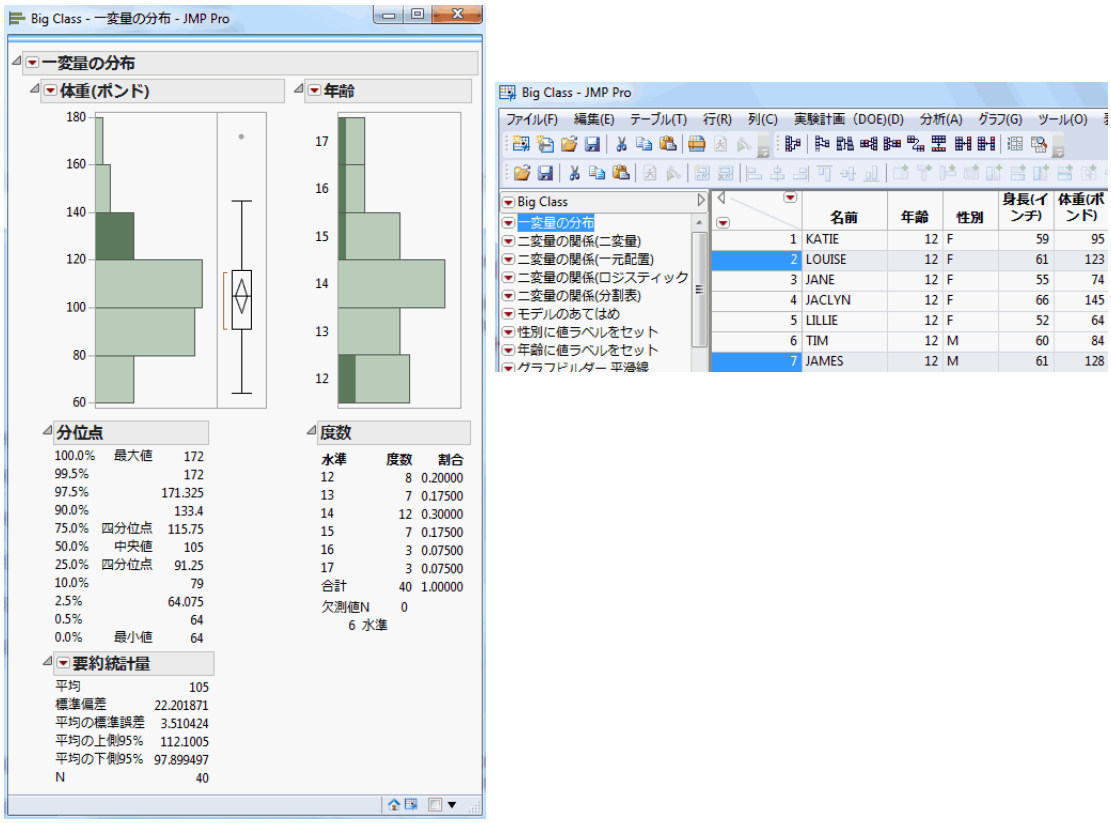
図 15.2「ソース」に「一変量の分布」レポートが表示



4. 「ソース」ペインにある「一変量の分布」レポートを、アプリケーションワークスペースにドラッグします。
5. ワークスペース上のレポートを右クリックし、[**隅に移動**] を選択します。
6. 「アプリケーションビルダー」の赤い三角ボタンメニューから、[**アプリケーションの実行**] を選択します。

「一変量の分布」レポートが、新しいウィンドウに表示されます（図 15.3）。ヒストグラムは、対話的で、データテーブルと関連付けられています。分析対象のデータが更新された場合は、アプリケーションを再実行すると、「一変量の分布」レポートも更新されます。

図15.3 単純なアプリケーションの例



アプリケーションビルダーの用語

アプリケーションは、1つ、もしくは、複数のモジュールで構成されます。**モジュール**には、1つのウィンドウで使われるオブジェクトとスクリプトが含まれています。アプリケーションを実行すると、モジュールごとに、1つのウィンドウが表示されます。あるウィンドウから、別のウィンドウを呼び出すこともできます。たとえば、あるウィンドウに、**「グラフの作成」**といったボタンを配置し、そのボタンをクリックすると、新しいウィンドウにグラフが表示されるようにできます。

表15.1に、アプリケーションビルダーを使用する際に知っておく必要のある用語を説明します。自動作成されたスクリプトを編集する必要はほとんどありませんが、これらの用語を知っておくと、モジュールの動作を理解するのに役立ちます。

表15.1 アプリケーションビルダーの用語

アプリケーション	1つまたは複数のモジュールで構成された最上位のファイル。
モジュール	オブジェクト、メッセージ、インスタンスなどのJSLステートメントを集めた、アプリケーションの構成要素。

表 15.1 アプリケーションビルダーの用語（続き）

単純なアプリケーション	レポートだけで構成されていて、プログラミングを全く必要としないアプリケーションのこと。
カスタムアプリケーション	自分の目的に沿った処理を、JSL スクリプトによって記述したアプリケーションのこと。
オブジェクトとメッセージ	**オブジェクト**とは、JMP の動的なエンティティで、たとえばデータテーブル、データ列、プラットフォーム結果ウィンドウ、グラフなどのこと。ほとんどのオブジェクトは、何らかのアクションの実行するメッセージを受け取ることができます。 **メッセージ**とは、オブジェクトに送られる式のこと。オブジェクトは、送られたメッセージを評価し、何らかのアクションを実行します。
モジュールインスタンス	モジュールが出現したもの。複雑なアプリケーションでは、実行時に実行されるスクリプトによって、複数のモジュールインスタンスを生成できます。
名前空間	変数名が衝突せず、一意に決められるように定義された変数のまとまり。 アプリケーションビルダーで作成されたアプリケーションでは、「`Application`」と「`ModuleInstance`」という名前空間が自動的に作成されます。`Application` 名前空間内のシンボルは、アプリケーション内のスクリプトだけを範囲とし、アプリケーション外のスクリプトでは使用できません。名前空間の詳細については、「プログラミング手法」の章の「高度な適用範囲指定と名前空間」（218 ページ）を参照してください。
変数	変数は、固有の名前をもち、何らかの値を含んでいます。アプリケーションビルダーで作成されたアプリケーションを実行すると、「`thisApplication`」と「`thisModuleInstance`」という変数が自動的に作成されます。 • `thisApplication` 変数は、アプリケーションのオブジェクトへの参照を含みます。アプリケーションは、1 つ、もしくは複数のモジュールで構成されています。 • `thisModuleInstance` 変数（`ModuleInstance` 名前空間で使用される）は、モジュールインスタンスのオブジェクトへの参照を含みます。モジュールは、ボックスやボタンなどで構成されています。
コンテナ	タブやアウトラインなどの、他のオブジェクトを含むことができるディスプレイボックスのこと。

アプリケーションの設計

アプリケーションを作成するにあたっては、まず目的を明確にし、どのようなグラフ要素を配置するのか、また、どの処理をJSLスクリプトでプログラミングする必要があるかを考えてください。

目的　アプリケーションで何を行うのかを明確にしてください。また、起動ウィンドウ、レポート、それともグラフなどから、カスタマイズの必要があるJMP機能は何かを考えてください。

グラフ要素　まず、モジュールがいくつ必要かを決めてください。そして、モジュールごとに配置するオブジェクトを決定します。アプリケーションビルダーの「ソース」ペインには、モジュールにドラッグ&ドロップできるボックスやアイコンが用意されています。また、用意されているサンプルを試してみて、自分の目的に近いモジュールがあるかどうかを確認してください。

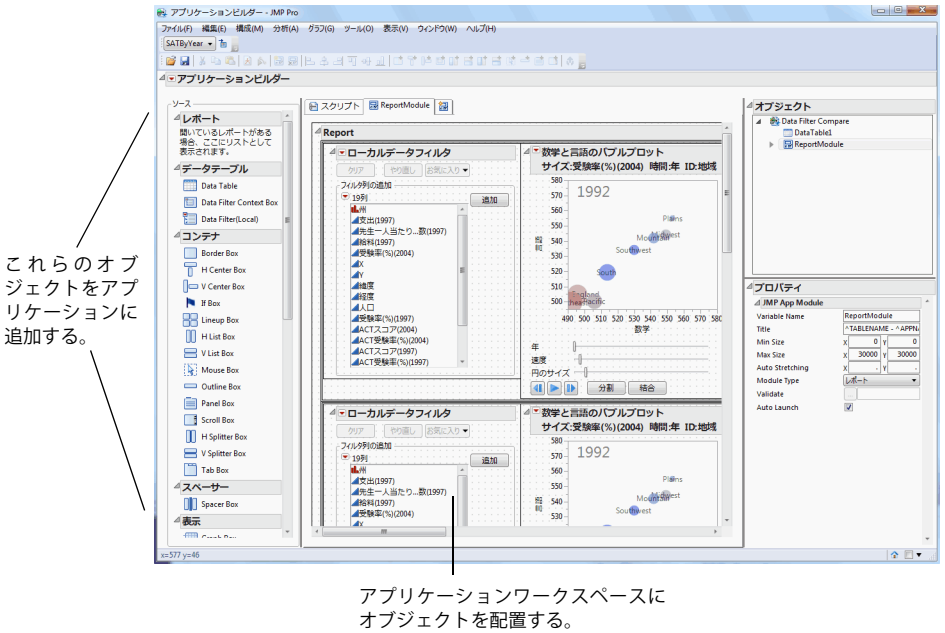
スクリプト　複雑なアプリケーションを作成するには、JSLスクリプトを記述します。アプリケーションのスクリプトの記述方法の詳細については、「スクリプトの記述」（602ページ）を参照してください。

非対話型で、オブジェクトを表示するだけの単純なアプリケーションであれば、スクリプトを記述する必要はありません。たとえば、JMPが標準的に生成するレポートを表示するだけならば、スクリプトを記述する必要はありません。

「アプリケーションビルダー」ウィンドウ

図15.4は、アプリケーションビルダーで作成中のアプリケーションです。

図15.4「アプリケーションビルダー」ウィンドウ



「アプリケーションビルダー」ウィンドウには、次の機能があります。

- ツールバーを使うと、オブジェクトの整列や一般的なディスプレイボックスの挿入など、数多くの機能をすばやく実行できます （ツールバーを表示するには［**表示**］>［**ツールバー**］>［**アプリケーションビルダー**］を選択します）。
- 「ソース」ペインには、アプリケーションに含めることができるオブジェクトが表示されています（ここには、開いたレポートやグラフなども表示されます）。オブジェクトとそのプロパティに関する情報を見るには、そのオブジェクトを右クリックして、［**スクリプトのヘルプ**］を選択してください。
- 中央は、ワークスペースです。点線のグリッド線が表示されており、オブジェクトを整列するのに便利です。ワークスペース内の［モジュール］タブにて、オブジェクトをクリックし、［スクリプト］タブにスクリプトを記述することにより、特定の機能をプログラミングできます。
- 「オブジェクト」ペインには、アプリケーションの階層が表示されます。モジュールと、各モジュールに含まれているオブジェクトが階層的に表示されます。オブジェクトをクリックすると、そのプロパティが表示され、また、ワークスペース上でオブジェクトが選択された状態になります。
- 「プロパティ」ペインでは、オブジェクトの位置や幅、名前などのプロパティを設定します。プロパティは、オブジェクトの種類によって異なります。

ヒント：作成するすべてのアプリケーションにおいて、グリッドを隠したり、グリッドにスナップするオプションを無効にするには、［**ファイル**］>［**環境設定**］>［**プラットフォーム**］>［**JMP App**］を選択し、該当するオプションの選択を解除します。これらの機能は、アプリケーションビルダーの赤い三角ボタンのメニューで解除することもできます。

赤い三角ボタンのオプション

アプリケーションビルダーの赤い三角ボタンのメニューには、アプリケーションの実行やデバッグ、サンプルアプリケーションを開く、グリッドを表示するなどのオプションがあります。

アプリケーションの実行　アプリケーションを開始します。各モジュールのウィンドウが開き、実際にユーザが操作するのと同様に、アプリケーションを対話的に操作できます。

アプリケーションのデバッグ　エラーを解決するために、JSL デバッガでスクリプトを開きます。詳細については、「スクリプト作成のツール」の章の「スクリプトのデバッグ／プロファイル」（63 ページ）を参照してください。

サンプルを開く　JMP の「Samples¥Apps」フォルダにあるサンプルアプリケーションの 1 つを開きます。用意されているサンプルは、一般的なアプリケーションを設定するための見本であり、必要に応じて変更することができます。表 15.2 は、サンプルについて説明したものです。

グリッドにスナップ　ワークスペースにオブジェクトをドラッグすると、オブジェクトが一番近いグリッド点線に沿って配置されます。デフォルトでオンになっています。

グリッドの表示　ワークスペースにグリッド点線を表示します。デフォルトでオンになっています。

ソースの表示　「ソース」パネルの表示／非表示を切り替えます。

オブジェクトとプロパティの表示 「オブジェクト」パネルと「プロパティ」パネルの表示／非表示を切り替えます。

自動スクロール オブジェクトをワークスペースの端近くまでドラッグすると、縦または横に自動的にスクロールします。デフォルトでオンになっています。

スクリプト アプリケーションをデータテーブル、ジャーナル、スクリプトウィンドウ、またはアドインに保存します。詳細は、「アプリケーションの保存オプション」（606ページ）の節を参照してください。

表15.2 提供されているサンプルアプリケーション

Six Quality Graphs.jmpappsource	3つの管理図、「一変量の分布」レポート、「工程能力分析」レポートを作成する。
Data Filter Compare.jmpappsource	バブルプロットと、それに対応したローカルデータフィルタと「表の作成」レポートを、2つ作成する。このアプリケーションでは、`Data Filter Context Box`関数を用いています。
Data Table Application.jmpappsource	層化抽出を行う。層別変数の列と、標本抽出率を選択できます。抽出結果は、データテーブルの行が選択された状態になります。
Graph Launcher.jmpappsource	等式の入力、軸の設定、グラフの作成を可能にする。ウィンドウが開いたままなので、等式に変更を加えて新しいグラフを作成することができます。
Instant App.jmpappsource	主成分分析と多変量管理図のレポートを組み合わせて表示する。
Instant App Customized.jmpappsource	Instant App.jmpappsourceに変更を加えたもの。主成分分析レポートを選択するオプション、マーカーのサイズを変更するオプション、平均を表示するオプションが含まれています。
Launcher With Report.jmpappsource	起動ウィンドウでユーザにデータテーブル列を選択させ、グラフを作成する。
Parameterized Instant App.jmpappsource	ユーザにデータテーブル列を選択させ、2つの多変量のレポートを作成する。Yの役割に引数が割り当てられます。つまり、レポートは、アプリケーションで指定されているテーブルだけでなく、開いているどのデータテーブルからでも作成できます。
Parameterized Measurement Systems Analysis (MSA) Combo Chart.jmpappsource	測定システム分析（MSA）の一連のレポートを作成する。
Presentation.jmpappsource	ナビゲーションボタンと組み込みのスクリプトを持つ、スライドショーのようなプレゼンテーションを作成する。
R Application.jmpappsource	分析対象の列を JMP で選択させた後、R の機能を用いて「Chernoff の顔グラフ」を描く。このアプリケーションを実行するには、Rにおいて、TeachingDemosパッケージが必要です。

表 15.2 提供されているサンプルアプリケーション（続き）

SAS Application.jmpappsource	SAS スクリプトを実行して、結果をレポートに追加する。SAS サーバーに接続していない場合は、SAS サーバーにログオンするためのダイアログが表示されます。

アプリケーションの作成

アプリケーションの仕様を決めたら、空白のアプリケーションを作成し、オブジェクトとスクリプトを追加していきます。

ここでは、アプリケーションを作成するための基本的な手順を説明します。

- 「新しいアプリケーションの作成」（594 ページ）
- 「オブジェクトの整列と削除」（597 ページ）
- 「オブジェクトプロパティの変更」（600 ページ）
- 「スクリプトの記述」（602 ページ）

新しいアプリケーションの作成

空白のアプリケーションを作成する

1. ［**ファイル**］>［**新規作成**］>［**アプリケーション**］（Macintosh の場合は［**ファイル**］>［**新規**］>［**アプリケーションの新規作成**］）を選択します。

   「アプリケーションビルダー」ウィンドウが開きます。
2. ［**ファイル**］>［**名前を付けて保存**］を選択し、ファイルを JMP ソースファイルとして保存します。拡張子は .jmpappsource です。

開いたウィンドウを結合して 1 つのアプリケーションを作成する

空白のアプリケーションから開始するのではなく、開いたデータテーブルやレポートからアプリケーションを作成することができます。

1. 結合したいウィンドウを選択します。
   - Windows では、開いたデータテーブルまたはレポートウィンドウの右下隅にあるチェックボックスにチェックを入れます。チェックボックスの横のポップアップメニューから［**選択したウィンドウを結合**］を選択します。
   - Macintosh では、［**ウィンドウ**］>［**ウィンドウの結合**］を選択し、開いたデータテーブルまたはレポートウィンドウを選択してから［**OK**］をクリックします。

   対象のオブジェクトが、1 つの新しいレポートに表示されます。
2. そのレポートの赤い三角ボタンメニューから、［**アプリケーションの編集**］を選択します。

JMP でのウィンドウの結合の詳細については、『JMP の使用法』を参照してください。

モジュールの管理

新しいアプリケーションを作成すると、［Module1］というモジュールのタブがデフォルトで表示されます。このモジュールにオブジェクトを追加していきます。アプリケーションに別のウィンドウを追加するには、新しいモジュールを追加します。

モジュールへのオブジェクトの追加

モジュールにオブジェクトを追加するには、次の手順に従います。

1. 「ソース」ペインで、オブジェクトの種類を選択します。
2. オブジェクトを［**Module1**］タブにドラッグします（または、オブジェクトをダブルクリックします）。
3. オブジェクトを選択し、「プロパティ」ペインでプロパティを更新します。詳細は、「オブジェクトプロパティの変更」（600 ページ）の節を参照してください。
4. オブジェクトにスクリプトを追加します。詳細は、「スクリプトの記述」（602 ページ）の節を参照してください。
5. （オプション）アプリケーションの実行時にモジュールが起動しないようにするには、「オブジェクト」ペインでモジュールを選択し、［**Auto Launch**］の選択を解除します （モジュールのうち 1 つだけをテストする場合などに便利）。
6. アプリケーションビルダーの赤い三角ボタンメニューから［**アプリケーションの実行**］を選択し、アプリケーションをテストします。

   アプリケーションが表示されます。

モジュールの追加

1. ［**構成**］>［**モジュールの追加**］を選択します。
2. 「プロパティ」ペインで、「**Module Type**」を指定します。
   - ダイアログ
   - メニュー付きダイアログ
   - モーダルダイアログ
   - 起動ウィンドウ
   - レポート

ヒント：アプリケーションの実行時に表示されるウィンドウのタイトルは、データテーブルの名前にハイフンとアプリケーション名を加えたものとなります。アプリケーション名を変更するには、JMP App オブジェクトの「**Name**」プロパティのテキストを変更します。

モジュール名の変更

「**Variable Name**」プロパティを変更します。

モジュールの削除

［**構成**］>［**モジュールの削除**］を選択します。

データテーブルの削除

データテーブルオブジェクトを選択し、右クリックして［**削除**］を選択します。

注：オブジェクトが関連付けられているデータテーブルは、削除できません。その場合、まずオブジェクトをすべて削除してから、データテーブルを削除します。

モーダルダイアログのモジュール

モーダルダイアログのモジュールには特殊な動作が見られるため、使用時に注意が必要です。

- `ret = Module1 << Create Instance()` は、ダイアログが完了するまで値を戻しません。その時点では、ダイアログはなくなっています。そのため、戻り値は、モジュールインスタンスへのハンドルにはなりません。`New Window()` と同様、値は、`{Button(1 | -1), User Data}` という形式で戻されます。(1 | -1) は、［OK］（1）または［キャンセル］（-1）ボタンが押されたことを示します。
- `User Data()` は、モジュールインスタンスの新しいプロパティです。ダイアログの実行中、モジュールのスクリプトは、ユーザデータを設定することができます。

  ```
  thisModuleInstance << Set User Data(...);
  ```

  これは、呼び出し側によって解釈される何かを保存するために行われます。ユーザデータは、設定時に評価されます。対応する `Get User Data()` もあります。
- `New Window()` と同様、［OK］ボタンや［キャンセル］ボタンが含まれていない場合は［OK］ボタンが追加されます。
- モジュールインスタンスには、オプションの「`Validate`」スクリプトプロパティがあります。このスクリプトは、モーダルダイアログにのみ使用され、`New Window()` の `On Validate()` のように動作します。［OK］ボタンが押されたときに呼び出され、入力を受け付けるときは 1、閉じる操作を許可しないときは 0 を戻します。

ヒント：「Validate」プロパティを使用すると、`thisModuleInstance << Set User Data()` を呼び出すことができます。

- モーダルダイアログは、他の種類のモジュールとは異なり、モジュールスクリプトが完了するまで表示されません。他の種類のモジュールの場合、ウィンドウは、`thisModuleInstance << Create Objects` の呼び出し中に作成されます。モーダルダイアログの場合は、この時点で表示すると、制御が停止し、表示されているボックスの内容を初期化する手段がなくなってしまうため、表示されません。ウィンドウがまだ作成されていないため、ウィンドウタイトルの設定、クローズ時に実行するスクリプトの設定といったアクションを行うことはできません。

オブジェクトの整列と削除

オブジェクトをモジュールに追加するには、オブジェクトを「ソース」ペインからワークスペースにドラッグするか、またはダブルクリックします。なお、現在、選択されているオブジェクトの周りには、青色の枠が表示されます。このとき、いくつかの方法でオブジェクトを整列させることができます。また、オブジェクトのコンテナを変更したり、オブジェクトを新しいコンテナに挿入することもできます。

ヒント：［オブジェクト」ペインでも、オブジェクトを選択できます。特に、外側のコンテナに完全に覆われているオブジェクトや、枠の中にテキストボックスがあるオブジェクトを選択するには、こちらを用いたほうが簡単に選択できます。

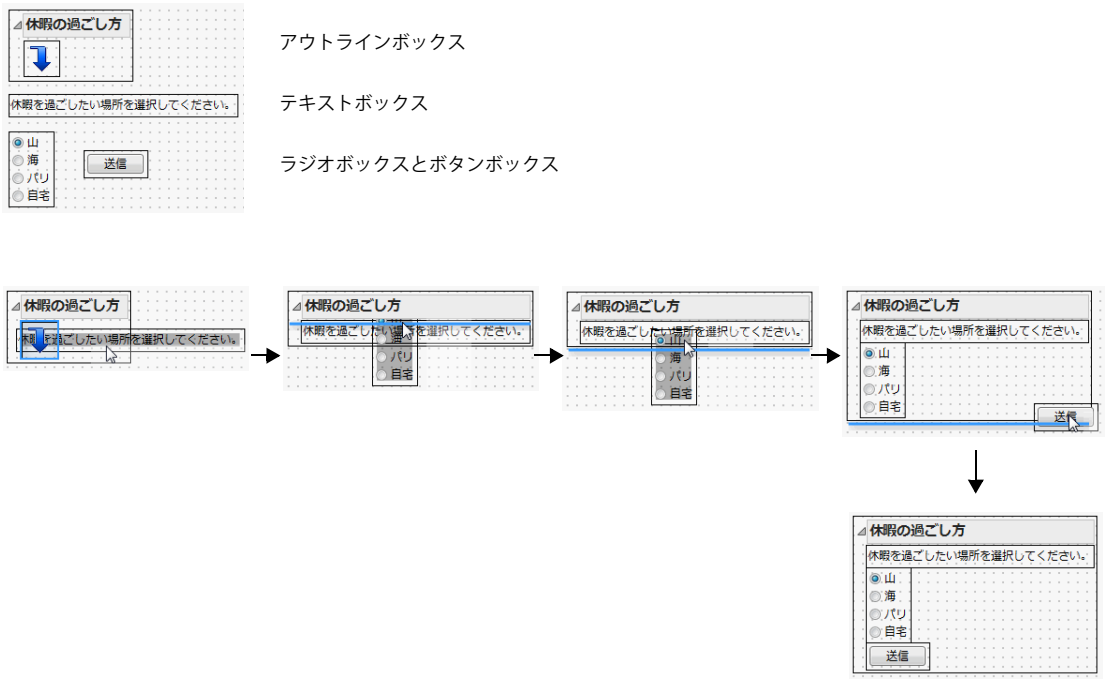
オブジェクトのドラッグ

オブジェクトを配置する方法の1つは、ワークスペースでドラッグする方法です。

- ワークスペースにオブジェクトをドラッグすると、オブジェクトの左上隅が最も近いグリッド点に配置されるように調整されます。より細かく配置したい場合は、アプリケーションビルダーの赤い三角ボタンのメニューで［**グリッドにスナップ**］の選択を解除します。また、JMPの環境設定（［ファイル］>［環境設定］>［プラットフォーム］>［JMP App］を選択）で設定すると、すべてのアプリケーションで［**グリッドにスナップ**］を無効にできます（［**ファイル**］>［**環境設定**］>［**プラットフォーム**］>［**JMP App**］を選択）。
- オブジェクトをドラッグする際、Shiftキーを押すと、移動を1つの軸方向に制限できます（たとえば、オブジェクトを縦方向に動かさず、横方向のみに動かすことができます）。
- コンテナの中にオブジェクトをドラッグする場合は、オブジェクトをドロップできる場所に矢印が表示されます。
- オブジェクトを別のオブジェクトの上にドラッグすると、ドロップできる場所に青い線が表示されます。

図15.5に、コンテナ内にオブジェクトをドラッグする方法を示します。

図15.5 コンテナ内にオブジェクトをドラッグする例



コンテナオブジェクトのX座標とY座標の変更

- コンテナオブジェクトを配置する座標を細かく決めたい場合は、オブジェクトを選択した後、「**X Position**」と「**Y Position**」のプロパティを変更します。新しいX座標を入力した後、Tabキーを押し、オブジェクトの移動を確認してから、新しいY座標を入力します。
- オブジェクトを左上隅に配置するには、右クリックして［**隅に移動**］を選択します。このオプションは、XとYの座標を0に設定します。Macintoshの場合は、Ctrlキーとcommandキーを押しながら［**隅に移動**］を選択します。

複数のオブジェクトの整列

オブジェクトの縦方向や横方向の座標を揃えるには、複数のオブジェクトを選択し、右クリックメニューの［**配置**］>［**ボックスの配置**］からオプションを選択します。

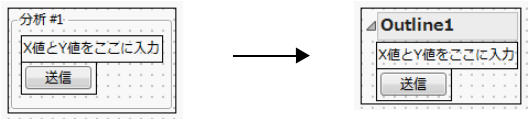
ヒント

- コンテナ内部に配置されている複数のオブジェクトではなく、コンテナ全体を選択した場合、［**配置**］オプションは使用できません。
- コンテナ内部で右クリックすると、内部にあるオブジェクトが、意図せずに選択されてしまう場合があります。コンテナ外部の、ワークスペースのどこかで右クリックするようにしてください。

コンテナの種類の変更

コンテナを挿入した後、種類を変更したくなった場合、オブジェクトを作成し直す必要はありません。たとえば、テキストとボタンを含むパネルを作成したとしましょう。そのパネルをアウトラインに変更する場合は、パネルを選択した後、ワークスペースで右クリックし、［**コンテナの変更**］>［**アウトライン**］を選択します（図 15.6）。

図 15.6 パネルをアウトラインに変更



この例では、アウトラインのタイトルは、「Outline1」のように初期化されますので、パネルのタイトルに合わせて変更してください。

ヒント：オブジェクトの［**Horizontal**］プロパティを選択したり、選択を解除したりすると、ボックスの配置方向を簡単に変更できます。

新しいコンテナへのオブジェクトの挿入

アプリケーションビルダー上部のツールバーには、境界ボックスやマウスボックスなどのコンテナのボタンがあります（図 15.7）。

選択したオブジェクトをコンテナに挿入するには、ツールバーで該当するボタンをクリックするか、［**構成**］>［**コンテナの追加**］を選択します。

図 15.7 コンテナツールバー

オブジェクトの複製

オブジェクトをコピーして貼り付けた場合、複製されたオブジェクトには、新しい名前が付けられます。また、オブジェクトに付随しているスクリプトも名前が変更されます。Ctrl キー（Macintosh の場合は command キー）を押しながらオブジェクトをドラッグしても、オブジェクトの複製を作成できます。

オブジェクトの削除

- 1つ、または複数のオブジェクトを選択して Delete キーを押すと、そのオブジェクトが削除されます。
- オブジェクトをモジュールの外にドラッグしても、そのオブジェクトが削除されます。
- Macintosh の場合、オブジェクトを選択し、Ctrl キーと command キーを押しながら［**削除**］を選択しても、削除されます。
- Windows の場合、［**編集**］>［**クリア**］を選択すると、モジュール内のすべてのオブジェクトが削除されます。

- オブジェクトを選択し、BackSpace キーを押しても、そのオブジェクトが削除できます。

オブジェクトのスクリプトが不要になった場合は、スクリプトも削除できます。

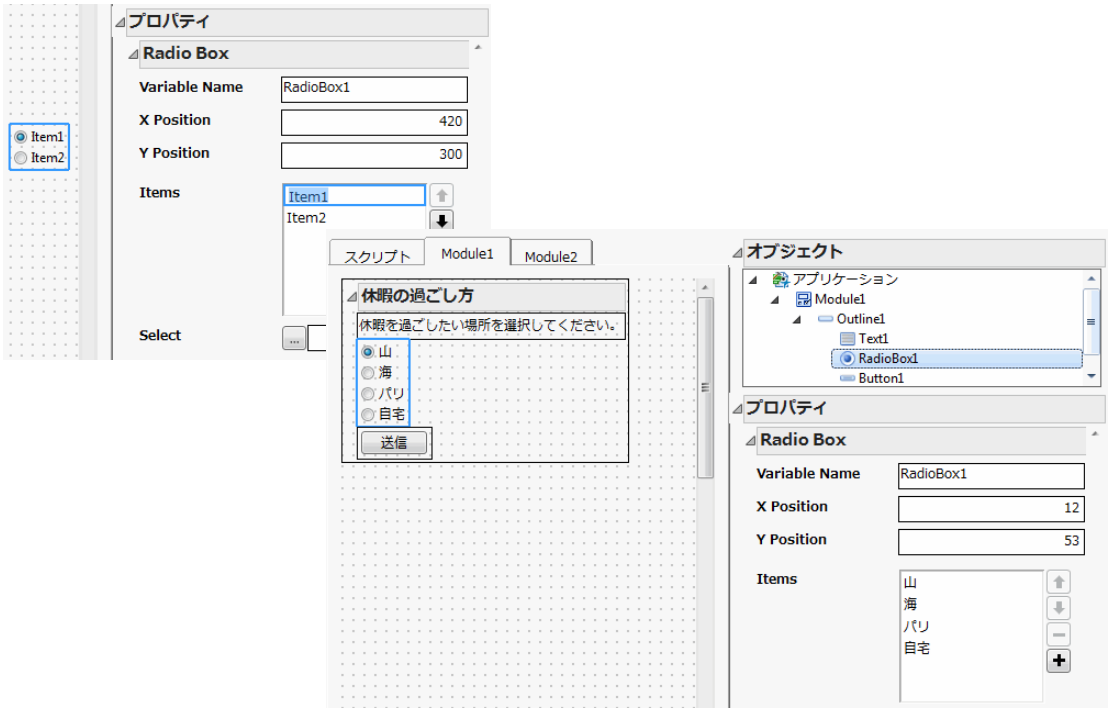
オブジェクトプロパティの変更

ワークスペースに置かれたオブジェクトのプロパティは、「プロパティ」ペインで変更できます。リスト項目やボタン名などの設定は、これらのプロパティを変更するだけで行えますので、JSL を記述する必要はありません。

変数名には大文字／小文字の区別があり、スペースも認識されます。スクリプトに「Button1」という名前のオブジェクトが含まれているときに、別のオブジェクトの名前を「Button 1」に変更しようとすると、警告が表示されます。

たとえば、図 15.8 のようなラジオボックスのリスト項目を設定するには、「Item1」と「Item2」をダブルクリックして、新しいリスト項目を入力していきます。

図 15.8 ラジオボックスのオブジェクトプロパティ



オブジェクトの詳細を参照するには、オブジェクトを右クリックし、[**スクリプトのヘルプ**] を選択します。選択したオブジェクトが強調表示され、JMP のスクリプトの索引が開きます。「スクリプトの索引」にスクリプトが用意されている場合は、それを実行するとオブジェクトの例を確認できます。

「プロパティ」ペイン内の項目については、項目の上にカーソルを置くと説明が表示されます。

アプリケーションのプロパティ

アプリケーションに対するプロパティには、実行パスワードやデータテーブル名などがあります。これらのプロパティを設定するには、「オブジェクト」のリストボックスで、「アプリケーション」を選択します。

Name　ここで設定した名前は、アプリケーションのタイトルバーにおいて、データテーブル名の後に表示されます。

Auto Launch（起動ウィンドウ）　ユーザに対し、アプリケーションで定義されている引数を選択するための起動ウィンドウが表示されます。この起動ウィンドウは、ユーザが、モジュールとして定義できるものではありません。この起動ウィンドウは、各モジュールの「Auto Launch」プロパティがオンになっている場合に自動的に呼び出されます。

Encrypt（暗号化）　テキストエディタを使ったアプリケーションの編集が不可能になります。ユーザが実行するアプリケーションだけが暗号化されます（.jmpapp ファイルと JMP アドイン内のアプリケーション）。スクリプト（データテーブル、ジャーナル、およびアドインに保存されたスクリプト）が、暗号化されます。暗号化の詳細については、「プログラミング手法」の章の「スクリプトの暗号化と暗号解読」（243 ページ）を参照してください。

Run Password（実行パスワード）　アプリケーションの実行に必要なパスワードを入力します。パスワードをテストするには、アプリケーションビルダーの外でアプリケーションを実行します。

テーブルモジュールのプロパティ

Data Table オブジェクトを挿入すると、データテーブルオブジェクトが作成されます。データテーブルパスなどのプロパティを定義するには、まず、「オブジェクト」ペインでそのデータテーブルモジュールを選択します。そして、「プロパティ」ペインで次のオプションを変更します。

Variable Name　データテーブルオブジェクトの名前を指定します。この名前は「プロパティ」ペインと「オブジェクト」ペイン、そして、アプリケーションの JSL スクリプト内に表示されます。

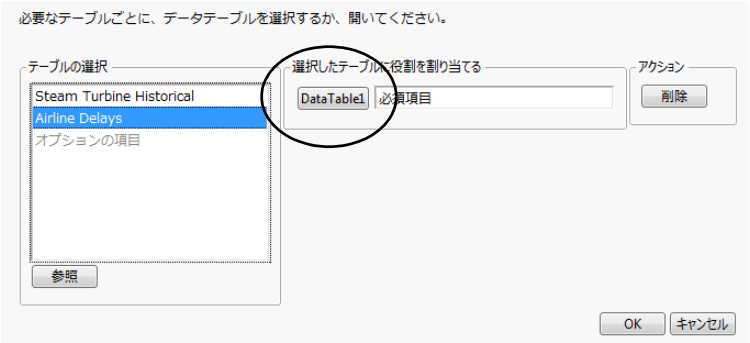
Path　アプリケーションで使用されるデータテーブルの絶対パスまたは相対パスを指定します。データテーブル名の前に、$HOME や $USER_APPDATA といったパス変数を使用することもできます。

このアプリケーションを編集する際、指定したデータテーブルがアプリケーションビルダーによって開かれます。「Path」プロパティが空である場合や、指定のデータテーブルが見つからない場合は、テーブルを開くよう指示するプロンプトが表示されます。

データテーブルを閉じたとき、アプリケーション内にそのデータテーブルに依存するオブジェクトがある場合は、それらのオブジェクトがアプリケーションから削除され、警告メッセージが表示されます。オブジェクトを復元するには、アプリケーションを開き直します。

Label　ユーザにデータテーブルを開くことを促す際に使用する文字列を指定します。図 15.9 では、デフォルトの値が表示されてます。

図15.9 データテーブルを開くためのプロンプト



Location ユーザがアプリケーションを実行したときに、アプリケーション内で使用されるデータテーブルが選択される方法を指定します。

- Current Data Table: 現在のデータテーブルを使用します。開いているデータテーブルがない場合、ユーザにデータテーブルを開くよう促します。
- Full Path:「Path」プロパティで指定されたデータテーブルを使用します。
- Name: 指定の名前を持つ、最初に開かれたデータテーブルを使用します。そうでない場合、「Path」プロパティで指定されたデータテーブルを使用します。
- Prompt: ユーザに、開かれているデータテーブルを選択するよう、または、データテーブルを指定するよう促します。
- Script: アプリケーションスクリプトまたはモジュールスクリプト内で定義されているデータテーブルを使用します。

Invisible データテーブルを非表示にします。ただし、JMP ホームウィンドウにはリストしたままです。このオプションは、「Location」で［Full Path］および［Name］を選択した場合に使用できます。

スクリプトの記述

オブジェクトに特定の機能を持たせるためには、スクリプトを記述します。たとえば、ユーザがボタンをクリックすると何らかの処理を行う機能や、ディレクトリを選択してデータテーブルを選択する機能などは、スクリプトで記述する必要があります。スクリプトでプログラミングすれば、ラジオボタンの選択に応じて、異なったグラフを表示するといった機能も実現できます。

アプリケーションとモジュールの名前空間

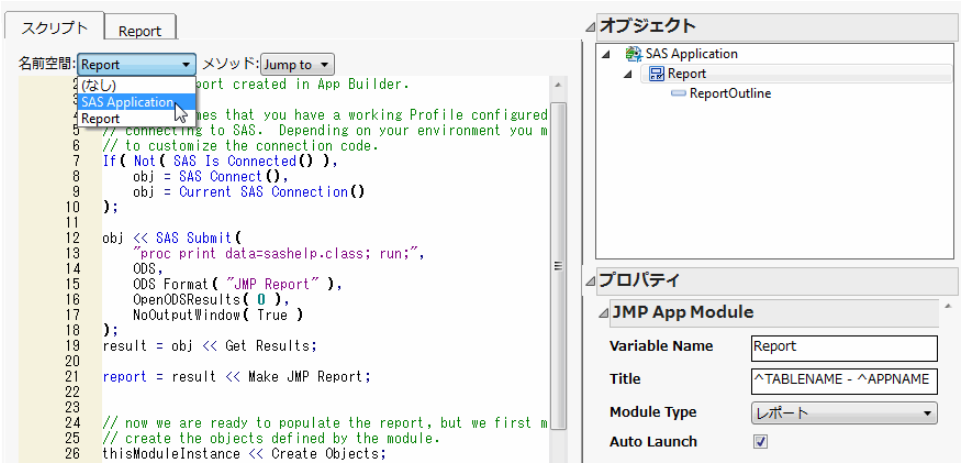
アプリケーションビルダーは、変数がスクリプト間で競合（衝突）しないようにするため、アプリケーションと各モジュールに対して、名前空間を自動的に設定します。

- アプリケーションの名前空間には、アプケーション全体で使用されるスクリプトを記述してください。この名前空間で定義された関数は、どのモジュール内でも使用できます。

- 各モジュールの名前空間には、そのモジュールのインスタンスが作成されたときに実行するスクリプトを記述してください。たとえば、あるアプリケーションにおいて、ボタンをクリックしたときに新たなウィンドウを開くとします。このウィンドウは、そのウィンドウのモジュールのインスタンスです。なお、同じモジュールから2つのインスタンスを作成した場合、各インスタンスに、それぞれ自身の変数や関数のコピーが含まれます。

これらの名前空間内のスクリプトを見るには、[**スクリプト**]タブをクリックし、「名前空間」リスト（または「オブジェクト」ペイン）で名前空間を選択します。詳細については、図15.10を参照してください。

図15.10 アプリケーションとモジュールの名前空間



名前付きスクリプトと**匿名スクリプト**の2種類のスクリプトがあります。

名前付きスクリプト

名前付きスクリプトは、関数の形式で定義されています。複数の異なるコントロールにおいて、共通のコードを使用できるという利点があります。名前付きスクリプトの関数において、`this`引数は、関数を呼び出しているコントロールを示します。次の例では、ボタンをクリックしたときに、`Get Button Name`が`this`引数に送られ、ログにボタン名が印刷されます。

```
Button1Press = Function({this}, Print(this <<Get Button Name))
```

別のボタンで`Button1Press`スクリプトを実行しても同じ結果となります。

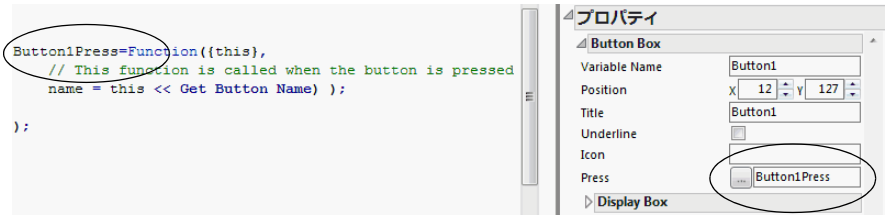
他の利点としては、名前付きスクリプトを追加した場合は、「スクリプト」タブにプログラムのテンプレートが挿入されるので、プログラミングの手間が省ける点が挙げられます。さらに、「スクリプト」タブでは「メソッド」リストからスクリプトを選択でき、該当の関数までジャンプできます。これは、プログラムが長い場合に便利です。

名前付きスクリプトは、次のように追加します。

1. オブジェクトを右クリックして［**スクリプト**］を選択し、追加したいスクリプトを選択します。この例のボタンでは、［**Press**］を選択します （Macintosh の場合は、Option を押しながら［**スクリプト**］>［**Press**］を選択）。

   ［**スクリプト**］タブに、スクリプトのテンプレートが表示されます（図 15.11）。また、名前付きスクリプトの名前である **Button1Press** が、スクリプト中では関数名として使われ、また、オブジェクトのプロパティでも表示されます。

図 15.11 新しいスクリプトとスクリプトプロパティ



2. 必要な機能を実現するため、スクリプトとプロパティを編集します。たとえば、図 15.11 のようにボタンボックスを変更すると、次のようになります。
   - 「Title」プロパティのテキストが、「**送信**」に変更されています。ボタンに、「送信」と表示されます。
   - `Close Window` メッセージを、オブジェクトに送るように、スクリプトが変更されています。これにより、ユーザが［**送信**］ボタンをクリックすると、ウィンドウが閉じます。

ヒント：オブジェクトを削除しても、該当するスクリプトは削除されません。スクリプトを追加した後、オブジェクトを削除し、そのスクリプトも不要であるなら、［**スクリプト**］タブから、削除してください。スクリプトを自動的に削除しないのは、将来必要になるかもしれないスクリプトが削除されるのを防ぐためです。

また、オブジェクトのプロパティで、スクリプトの名前を変更した場合は、［**スクリプト**］タブでも、その名前に応じて変更してください。該当するスクリプトがアプリケーションの別の部分でも使用されているときは、そこでも名前を変更してください。

匿名スクリプトの作成

匿名スクリプトは、それを定義したオブジェクトに対してしか実行できません。たとえば、ボタンに対して、そこでしか使用しないログ出力の機能（`Print`）が必要な場合、匿名スクリプトで記述すれば、スクリプト全体で定義する名前付きスクリプトの数を減らすことができます。匿名スクリプトは、オブジェクトのプロパティとして設定します。そのため、［スクリプト］タブにおいて、名前付きスクリプトが雑多になりすぎるのも回避できます。

次に、2 種類の匿名スクリプトの例を示します。

```
Print(Button1 <<GetButtonName) // 単純な匿名スクリプト
Function({this}, Print(this <<GetButtonName)) // パラメータ化された匿名スクリプト
```

1番目のスクリプトでは、「Button1」変数にメッセージを送っています。一方、2番目のスクリプトでは、コントロールを表す「this」に対して、メッセージを送っています。

チェックボックスなどのオブジェクトは、thisを用いて、必要な情報を取得する必要があります。たとえば、チェックボックスを用いた場合、どの項目がチェックされたかを取得する必要があるでしょう。

匿名スクリプトは、次の手順により設定できます。

1. オブジェクトを選択し、オブジェクトのプロパティで...をクリックします。

   匿名スクリプトエディタが表示されます。

2. スクリプトを入力し、[**OK**]をクリックします。

   匿名スクリプトのテキストが、オブジェクトのプロパティに表示されます（なお、名前付きスクリプトを設定した場合、ここには、その関数の名前が表示されます）。

ヒント：プログラムの保守を簡単にするには、同じコードを、匿名関数として何度もコピーして用いるのは避けてください。同じコードを複数の場所で用いる場合は、名前付きスクリプトを使用してください。

特定のスクリプトの表示

特定のオブジェクトのスクリプトを表示する方法はいくつかあります。

- [**モジュール**]タブでオブジェクトをダブルクリックします。
- オブジェクトに複数のスクリプトがある場合は、オブジェクトを右クリックして[**スクリプト**]を選択し、表示したいスクリプトを選択します。また、[**スクリプト**]タブを選択し、**「名前空間」**リストからモジュール名を選択して、**「メソッド」**リストからスクリプト名を選択する方法もあります。同様に、アプリケーションスクリプトを表示するには、**「名前空間」**リストから**「アプリケーション」**を選択します。

どちらの場合も、[**スクリプト**]タブが表示されたとき、そのオブジェクトに対するスクリプトの一行目に、カーソルは移動します。

ヒント：雑然としたスクリプトを読みやすくするには、右クリックして[**スクリプトを再フォーマット**]を選択します。

スクリプトとともにオブジェクトをコピーして貼り付け

スクリプトを持つオブジェクトをコピーして貼り付けた場合、新しいオブジェクトとスクリプトには、元のものとは別の名前が与えられます。

アプリケーションの編集または実行

アプリケーションを開いて編集するには、[**ファイル**] > [**開く**]を選択し、.jmpappsourceファイルを選択して[**開く**]を選択します。

注：「Table」プロパティが空である場合や、指定のデータテーブルが見つからない場合でも、アプリケーションは実行されます。ただし、データテーブルを要求するオブジェクトは作成できないため、警告が表示されます。

開いているアプリケーションを実行するには、アプリケーションビルダーの赤い三角ボタンのメニューから、［**アプリケーションの実行**］を選択します。

閉じているアプリケーションを実行するには、［**ファイル**］>［**開く**］を選択し、.jmpapp ファイルを選択して［**開く**］を選択します。

Windows の場合、次のいずれかを行うことによっても、JMP ホームウィンドウからアプリケーションを開いたり、実行したりできます。

- Windows エクスプローラに表示されたファイルを、JMP ホームウィンドウまたは空のアプリケーションウィンドウにドラッグします。
- 「最近使ったファイル」内のファイルをダブルクリックします。
- .jmpappsource ファイルまたは .jmpapp ファイルを右クリックし、［**アプリケーションの編集**］または［**アプリケーションの実行**］を選択します。

アプリケーションの保存オプション

JMP には、アプリケーションファイルを保存するためのオプションがあります。［**ファイル**］>［**名前を付けて保存**］（Macintosh の場合は［**ファイル**］>［**書き出し**］）を選択した場合、アプリケーションソースファイル（.jmpappsource）、アプリケーション（.jmpapp）、またはスクリプト（.jsl）のどれで保存するかを指定できます。

「**スクリプト**」の赤い三角ボタンのメニューには、スクリプトを保存するためのその他のオプションがあります。暗号化されたスクリプトを保存した場合、そのスクリプトは `JSL Encrypted()` 関数で閉じられ、空白とコメントが保持されます。

スクリプトをデータテーブルに保存　これにより、データテーブルスクリプトからアプリケーションを実行できます。このオプションはデータテーブルが開いている場合にのみ有効です。

スクリプトをジャーナルに保存　アプリケーションスクリプトを、開いているジャーナルまたは新しいジャーナルに組み込みます。ジャーナルに保存されたアプリケーションを実行するには、リンクをクリックします。

スクリプトをスクリプトウィンドウに保存　開いているスクリプトウィンドウ、または、新しいスクリプトウィンドウに、スクリプトを表示します。これらのスクリプトは編集できますが、ここで大幅に変更したアプリケーションは、アプリケーションビルダーで編集できなくなる場合があるので注意してください。

スクリプトをアドインに保存　アドインを作成します。アドインをインストールすると、開発したアプリケーションが JMP メニューに設定されます。アドインを作成する方法については、「JMP アドイン」（609 ページ）を参照してください。

その他の例

以下の例で、JMPにおけるアプリケーションのさまざまな用途を紹介します。

分析対象の列

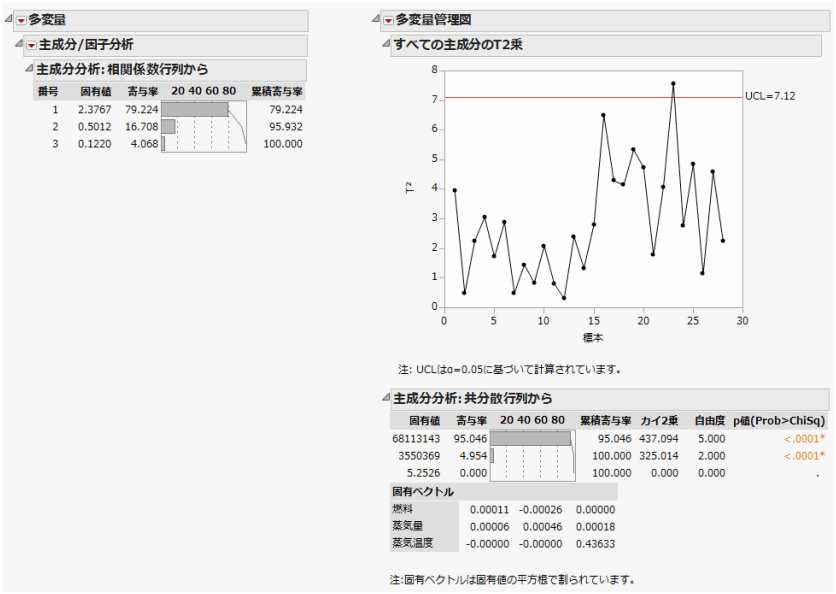
次の例は、分析対象の列を指定できるアプリケーションです。このアプリケーションでは、ユーザは、起動ウィンドウで列を選択できます。選択された列に対して、特定のレポートが作成されます。

1. JMPのSamples¥Data¥Quality Controlフォルダにある「Steam Turbine Historical.jmp」サンプルデータテーブルを開きます。
2. 「主成分分析」と「主成分負荷量プロット」のテーブルスクリプトを実行してレポートを作成します。
3. ［**ファイル**］>［**新規作成**］>［**アプリケーション**］を選択します。

   「アプリケーションビルダー」ウィンドウが開きます。
4. ウィンドウのサイズを大きくします。
5. 「ソース」ペインの各多変量レポートをアプリケーションワークスペースにドラッグし、横に並べます。
6. 両方のレポートを選択して右クリックし、［**配置**］>［**上端揃え**］を選択します。
7. 2つのレポートそれぞれにおいて、レポートをクリックした後、「オブジェクト」の「**Y Variable**」プロパティに、「yvar」と入力します。
8. 「アプリケーションビルダー」の赤い三角ボタンメニューから、［**アプリケーションの実行**］を選択します。
9. 「**燃料**」、「**蒸気量**」、「**蒸気温度**」の各変数を選択し、［**Y**］ボタンをクリックします。
10. ［**OK**］をクリックします。

    選択された列に対する2つのレポートが、1つのウィンドウ内に表示されます（図15.12）。

図15.12 多変量アプリケーション



ヒント：データテーブルのプロパティには、デフォルトでは、データテーブルへの絶対パスが設定されます。代わりに、$SAMPLE_DATA などのパスや、任意の絶対パスまたは相対パスを設定することもできます。ただし、設定されたパスは、ユーザがアクセスできるものでなければいけません。パスは、ユーザにとってアクセス可能でなければなりません。

日付セレクタ

アプリケーションに日付セレクタウィンドウを挿入するには、次の手順に従います。

1. 「ソース」ペインから、「Number Edit Box」をワークスペースにドラッグします。
2. 「Number Edit Box」を選択し、「プロパティ」ペイン内の「Format」の隣のボタンをクリックします。
3. リストから、［**日付**］>［**m/d/y**］を選択します。
4. 「総桁数」ボックスに「25」とタイプします。
5. ［**OK**］をクリックします。
6. アプリケーションビルダーの赤い三角ボタンのメニューから［**アプリケーションの実行**］を選択します。

   これで、新しいウィンドウ内に数値編集ボックスが作成され、日付が入力されました。
7. 日付セレクタウィンドウを開くには、マウスをボックスの上に置き、青い三角形が表示されるのを待ちます。
8. 青い三角形をクリックすると、日付セレクタウィンドウが表示されます。

図15.13 日付セレクタの例

日付セレクタを使うと、ボックスに表示する年月日と時間を選択することができます。

JMP アドイン

JMP アドインは、JMP の［**アドイン**］メニューからいつでも実行できる JSL スクリプトです。JMP アドインをサブメニューにまとめたり、必要に応じて複数レベルのメニューを作成したりできます。

JMP アドインは、基本的に、一連のファイルを 1 つのファイルに圧縮して保存したものです。アドインは、他の JMP ユーザに配布できます。

アプリケーションビルダーと、アドインビルダーを組み合わせて用いると良いでしょう。まず、アプリケーションビルダーで、特定の処理を行うスクリプトを作成します。続いて、アドインビルダーで、そのスクリプトをアドインにして配布すれば、ユーザは、ファイルを開いて実行するのではなく、メニューからスクリプトを実行できるようになります。

注：アプリケーションビルダーの詳細については、「アプリケーションビルダー」（587 ページ）を参照してください。

アドインビルダーを使ったアドインの作成

JMP アドインを作成するには

- Windows では［**ファイル**］>［**新規作成**］>［**アドイン**］を選択します。
- Macintosh では［**ファイル**］>［**新規**］>［**アドインの新規作成**］を選択します。

アドインを作成するには、次のような手順が必要です。

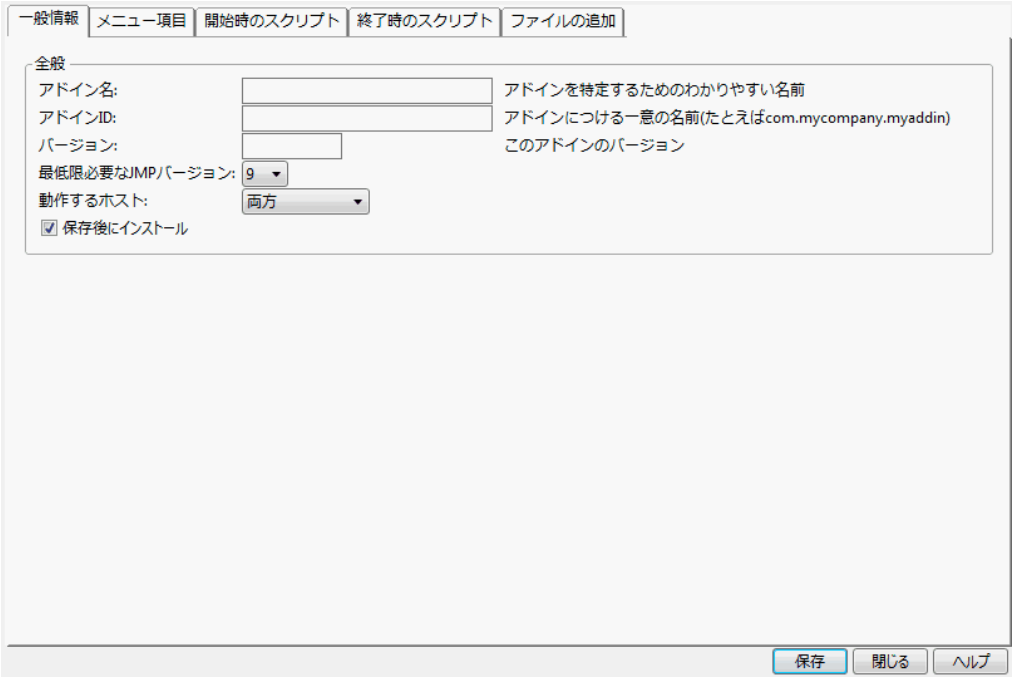
- 「一般情報の追加」（610 ページ）
- 「メニュー項目の作成」（611 ページ）

- 「開始時や終了時のスクリプトの指定（オプション）」（612 ページ）
- 「その他のファイルの追加」（613 ページ）
- 「アドインの保存」（613 ページ）
- 「アドインのテスト」（613 ページ）

一般情報の追加

まず、［**一般情報**］タブで、アドインの識別と設定に必要な一般情報を入力します。

図 15.14 アドインビルダーの［一般情報］タブ



1. アドインの名前を入力します。

   これがアドインの登録名となり、［**表示**］>［**アドイン**］のウィンドウに表示されます。

2. 一意の ID 文字列を入力します。

   一意の ID 文字列では、大文字と小文字が区別されます。一意であることを確実にするため、DNS 名の逆（たとえば `com.mycompany.myaddin`）を使用することを推奨します。ID 文字列は、次の要件を満たす必要があります。

   – 64 文字以内である。
   – 文字で始まる。
   – 文字、数字、ピリオド、および下線のみを使用する。

– スペースを含まない。

JSL では、アドインの参照にこの文字列を使用します。

3. アドインのバージョンを入力します。

後でアドインに変更を加える予定がある場合は、バージョン番号を更新していけば、ユーザに正しいバージョンが配布されているかどうかを確認することができます。

4. アドインが使用できる最低の JMP バージョンを選択します。

注：アドインは JMP 9 で導入されたため、それ以前のバージョンではサポートされません。また、アプリケーションをアドインとして保存する場合は、JMP の最低バージョンとして 10 または 11 を選択してください。

5. アドインを Windows と Macintosh のどちらでサポートするか、または両方でサポートするかを選択します。

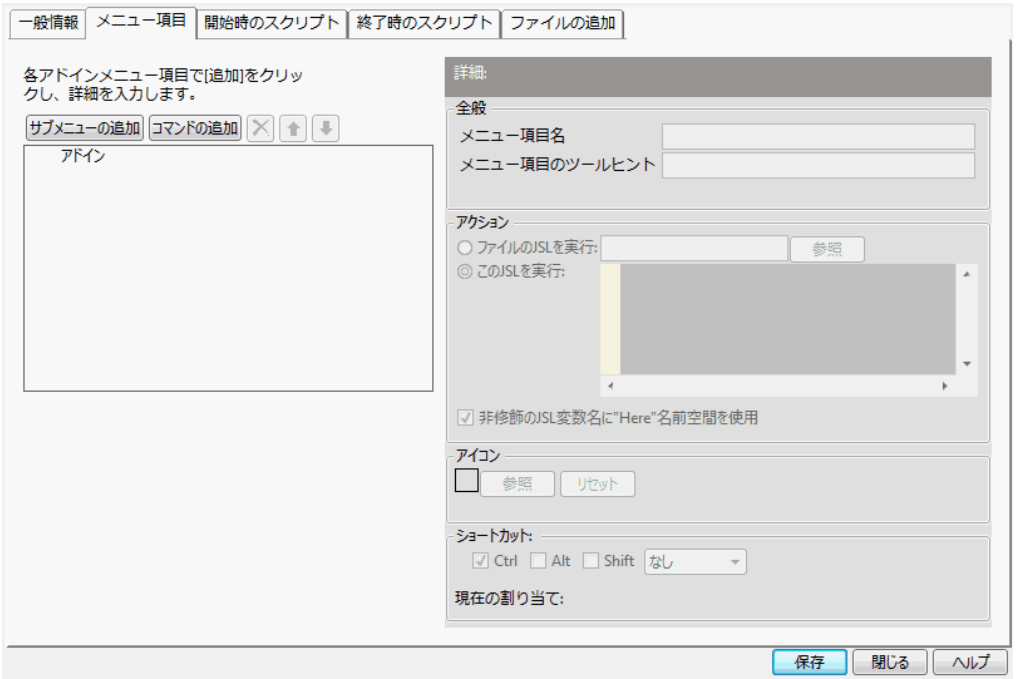
6. （オプション）アドインを保存後にインストールする場合は、［**保存後にインストール**］のチェックボックスにチェックを入れます。

このオプションにチェックが入っていない場合、アドインを保存してもインストールはされません。なお、アドインをインストールすると、［**アドイン**］メニューの項目に表示されます。

メニュー項目の作成

1. ［**メニュー項目**］タブをクリックします。

図 15.15 アドインビルダーの［メニュー項目］タブ



2. （オプション）［**サブメニューの追加**］をクリックします。

   複数のアドインがある場合、それを1つのサブメニューにまとめることができます。

3. サブメニューを追加するときは、［**メニュー項目名**］の隣にサブメニューの名前をタイプします。

   この名前が［**アドイン**］メニューに表示されます。

4. ［**コマンドの追加**］をクリックします。

5. ［**メニュー項目名**］の隣にアドインコマンドの名前をタイプします。

6. （オプション）ツールヒントを表示させたい場合には、**「メニュー項目のツールヒント」**の横に、ツールヒントのテキストを入力します。このテキストは、ユーザがメニュー項目の上にカーソルを置いたときに表示されます。

7. スクリプトを追加します。［**このJSLを実行**］を選択し、スクリプトをコピーして貼り付けるか、［**ファイルのJSLを実行**］を選択し、［**参照**］をクリックしてスクリプトを含むファイルを選択します。

8. （オプション）すべての非修飾のJSL変数が`Here`名前空間に含まれ、該当のスクリプトに対してのみローカルとなるようにするには、［**非修飾のJSL変数名に“Here”名前空間を使用**］を選択します。

注：カスタムメニューまたはツールバーを作成するスクリプトの場合、変数はデフォルトで`Here`名前空間に含まれます。

注：`Here`名前空間の詳細については、「プログラミング手法」の章の「高度な適用範囲指定と名前空間」（218ページ）を参照してください。

9. （オプション）必要に応じて、アイコンを追加します。ここで設定されたアイコンは、［**アドイン**］メニューのメニュー項目の横に表示されます。

10. （オプション、Windowsのみ）必要に応じて、アドインに対するキーボードショートカットを設定してください。

11. 複数のメニュー項目を追加する場合は、この手順を繰り返します。

    サブメニューとアドインコマンドを追加し、複数のレベルを構成することができます。

12. ［**保存**］をクリックし、アドインを任意のディレクトリに保存します。

13. ［**閉じる**］をクリックします。

注：JMPのメニューのカスタマイズ方法については、『JMPの使用法』を参照してください。

開始時や終了時のスクリプトの指定（オプション）

［**開始時のスクリプト**］タブでは、JMP（とアドイン）の起動時に実行されるスクリプトを設定できます。既存のスクリプトを選択するか（［**ファイルのJSLを実行**］）、スクリプトをコピーして貼り付けます（［**このJSLを実行**］）。たとえば、起動の直後にアドインがインストールされていることを知らせるメッセージを表示できます。

［**終了時のスクリプト**］タブでは、JMPの終了時、または、アドインを無効にしたときに実行されるスクリプトを設定できます。既存のスクリプトを選択するか（［**ファイルのJSLを実行**］）、スクリプトをコピーして貼り付けます（［**このJSLを実行**］）。たとえば、アドインを終了するとき、または無効にするときに、開いているJMPデータテーブルを保存するかどうかを確認するプロンプトを表示できます。

その他のファイルの追加

スクリプトが別のスクリプトを呼び出す場合や、グラフィックまたはデータテーブルを含む場合、それらのファイルをここに追加します。

アドインの保存

いずれかのタブの［**保存**］ボタンをクリックすることで、アドインを保存します。この操作により、アドインのファイルが作成されます。

- ［一般情報］タブで［**保存後にインストール**］オプションが選択されている場合は、すぐに［**アドイン**］メニューにアドインメニュー項目が表示されます。
- ［**保存後にインストール**］オプションが選択されていない場合は、保存したアドインファイルを開いたときに、インストールするかどうかを確認するプロンプトが表示されます。

アドインのテスト

アドインをインストールした後、次のようにしてアドインをテストします。

1. ［**表示**］>［**アドイン**］を選びます。
2. アドインを選択して、［**登録解除**］をクリックします。
3. ［**ファイル**］>［**開く**］を選択するか、.jmpaddinファイルをダブルクリックして、アドインを再インストールします。
4. メニューまたはツールバーボタンからスクリプトが正常に実行されること、そして、スクリプト自体が正常に動作することを確認します。

アドインの編集

保存したアドインを編集するには

1. ［**ファイル**］>［**開く**］を選びます。
2. アドインファイルを指定します。
3. 次のいずれかのオプションを選択します。
   - Windowsの場合、［**開く**］ボタンの右の矢印をクリックし、［**アドインビルダーを使って開く**］を選択します。
   - Macintoshの場合、［**編集モードで開く**］オプションを選択します。
4. ［**開く**］をクリックします。

   アドインビルダーでファイルが開きます。引数を更新し、変更を保存します。

［アドイン］メニューからのアドインの削除

［**アドイン**］メニューからアドインを削除するには

1. ［**表示**］>［**アドイン**］を選びます。
2. 「登録済みアドイン」リストからアドインを選択します。
3. ［**有効**］チェックボックスをオフにします。

アドインのアンインストール

アドインをアンインストールするには

1. ［**表示**］>［**アドイン**］を選びます。
2. 「登録済みアドイン」リストからアドインを選択します。
3. ［**登録解除**］をクリックします。

アドインの共有

.jmpaddin ファイルとして保存したアドインは、他のユーザに配布できます。.jmpaddin ファイルを電子メールで送付するか、またはネットワークフォルダや、(インターネット上の JMP User Community にある) JMP File Exchange などにアップロードしてください。

JMP ユーザが .jmpaddin ファイルを開くと、ファイルが適切なフォルダ内に展開され、アドインの登録とインストールが行われます。そして、JMP の［**アドイン**］メニューから、そのスクリプトを起動できるようになります。

複数のアドインのインストール

複数のアドインをインストールしたい場合は、アドインを次の場所にコピーします。

- Windows のアドインファイルの場所
  - %ALLUSERSPROFILE%¥SAS¥JMP¥AddIns（このコンピュータを使うすべてのユーザがアドインにアクセスできる）
  - %LOCALAPPDATA%¥SAS¥JMP¥AddIns（このコンピュータの現在のユーザのみがアドインにアクセスできる）
- Macintosh のアドインファイルの場所
  - /Library/Application Support/JMP/AddIns（このコンピュータを使うすべてのユーザがアドインにアクセスできる）
  - ~/Library/Application Support/JMP/AddIns（このコンピュータの現在のユーザのみがアドインにアクセスできる）

JMP は起動の際に、「addinRegistry.xml」ファイルを読み込みます。このファイルには、これまでに登録された JMP アドインの情報が含まれています。続いて、JMP はアドインフォルダ内のその他のアドインを検出し、自動的にインストールします。

次の点を念頭に置いてください。

- アドインのホームフォルダは、「addin.def」ファイルを含んでいるフォルダである必要はありません。つまり、「addin.def」ファイルを含むフォルダ以外の場所に、スクリプトなどのファイルを保存してもかまいません。実際のアドインファイルがある場所は、home オプションで指定できます。
- 新たに自動検出されたアドインの ID が、以前に明示的に登録されたアドインの ID と同じである場合、自動的に検出されたアドインの方が使用されます。

JSL を使ったアドインの登録

アドインが .jmpaddin ファイル形式になっていない場合も、JSL 関数の `Register Addin()` を使って、addin.def ファイルに基づいて、アドインを手動で登録できます。この関数は、アドインのインストールと登録を行います。

- これらの JSL 関数について詳しくは、『スクリプト構文リファレンス』の Register Addin および Unregister Addin の節を参照してください。
- 「addin.def」ファイルの作成方法については、「手動でのアドインの作成」（615 ページ）を参照してください。

次の点を念頭に置いてください。

- JMP が、指定のホームフォルダ内で「addin.def」ファイルを検出した場合、`Register Addin()` 関数で指定されていないオプションの引数については、そのファイルの値が使用されます。
- `Register Addin()` 関数で指定されていない値のみ、「addin.def」ファイルの値が使用されます。関数による引数の指定は開発中には便利な機能かもしれませんが、通常、アドインを登録するには「addin.def」ファイルだけで十分です。

手動でのアドインの作成

「addin.def」ファイルは、JMP　アドインの登録情報を示す名前と値を含んだテキストファイルです。「addin.def」ファイルに含まれる名前と値は、次のとおりです。

`id`　必須。アドインの一意の ID。最大 64 文字まで使用できます。文字列の最初は文字でなければならず、文字、数字、ピリオド、下線を使用できます。一意である確率を上げるために、DNS の逆の名前を使用することをお勧めします。

`name`　（オプション）表示名。アドインが GUI に表示されるときの名前です。制限のある ID 名の代わりになる、ユーザにわかりやすい名前をつけてください。この表示名は、翻訳名が指定されていない場合、また、翻訳が指定された言語以外で JMP が実行されている場合に使用されます。

name_xx （オプション）表示名をさまざまな言語に翻訳できます。*xx*は、各言語の2桁のISO 639-1コードです。翻訳名を指定しても、翻訳名を指定しなかった地域設定の下でJMPが実行される場合に備えて、言語に左右されない名前も指定しておきましょう。

home （オプション）アドインファイルのパス。指定されていない場合、「addin.def」ファイルがあるフォルダが、アドインのホームフォルダとなります。ホームフォルダが別の場所（たとえば、ネットワーク上の共有フォルダなど）にある場合は、このhomeオプションで指定する必要があります。

home_win （オプション） WindowsでJMPを実行する場合に使用されるアドインファイルのパス。Windows上のホームに値が指定されている場合、home_winの値で上書きされます。

home_mac （オプション）Macintosh版JMPでのアドインファイルのパス。Macintosh上でhomeに値が指定されている場合、home_macの値で上書きされます。

autoLoad （オプション）ブール値。デフォルト値は、1（真）。JMPの起動時にアドインを自動的にロードするかどうかをあらかじめ設定しておくことができます。

host （オプション）有効な値はWinおよびMac。

minJMPVersion （オプション）指定できる値は、アドインが使用できる最低のJMPメジャーバージョンを表す整数。

maxJMPVersion （オプション）指定できる値は、アドインが使用できる最高のJMPメジャーバージョンを表す整数。この設定は、アドインとJMPの特定のバージョンとの間に互換性がないことがわかっている場合**のみ**使用してください。以降のJMPバージョンについては、アドインの新バージョンを提供する必要があります。

「addin.def」ファイルの例

```
id="com.mycompany.myaddin"
name="My Add-In's Friendly Name"
name_fr="My Add-In's French Name"
name_de="My Add-In's German Name"
home="¥¥server¥share¥myjmpaddin"
Autoload=1
MinJMPVersion=9
```

JMPアドインの例

次の場所にある「Samples¥Scripts」フォルダに「Simple Calculator.jmpaddin」という名前のサンプルアドインが用意されています。

- Windowsの場合: C:¥Program Files¥SAS¥JMP¥<*バージョン番号*>¥Samples¥Scripts
- Macintoshの場合: \Library\Application Support\JMP\<*バージョン番号*>\Sample\Scripts

アドインの内容を見るには、拡張子を.zipに変更し、それを新しいフォルダ内に展開します。アドインの動作を確認するには、拡張子を.jmpaddinに変更し、インストールします。

アドインには次のファイルが含まれています。

addin.def

アドインをJMPに登録するための設定です。次の2行が含まれています。

```
id="com.jmp.sample.calculator"
name="Simple Calculator"
```

addin.jmpcust

対話式にカスタムメニューを構築する際に作成されるメニューカスタマイズファイルです。この例では、アドインメニュー項目をデフォルトの［**アドイン**］メニューに配置します。

calculator.jsl

単純な計算機のJSLスクリプト

calc_icon.gif

イメージファイル。計算機のアイコンとして使用されています。

第 16 章

プログラム例の紹介
サンプルによるプログラミングの学習

プログラミングを勉強するには、実際のプログラムコードを見るのが一番良い方法でしょう。この章では、日付時間値の変換や、レポートからの数値の取得など、JSLによるプログラミングで良く行われる処理を、実例にもとづき紹介します。JMPのインストール時に、これらのサンプルスクリプトもインストールされます。自分が行いたい処理に近いならば、これらのサンプルを利用してください。

目次

起動時のスクリプトの実行 621
文字の日付を数値の日付に変換 621
日付によるデータ抽出 623
計算式を含んだ列の作成 624
分析結果の一部を抜き出す 626
対話型プログラムの作成 628

起動時のスクリプトの実行

JMP の起動時に必ず実行したいスクリプトには jmpStart.jsl という名前を付け、使用しているオペレーティングシステムに応じて、次のフォルダに格納します。起動時に、JMP はここにリストされた順番でフォルダ内の jmpStart.jsl スクリプトを検索します。最初に検出された jmpStart.jsl スクリプトが実行され、検索は直ちに停止します。

1. C:¥Users¥<*ユーザ名*>¥AppData\Local¥SAS¥JMP¥<*バージョン番号*>（Windows）

   /Users/<*ユーザ名*>/Library/Application Support/JMP/<*バージョン番号*>（Macintosh）
2. C:¥Users¥<*ユーザ名*>¥AppData¥Local¥SAS¥JMP（Windows）

   /Users/<*ユーザ名*>/Library/Application Support/JMP（Macintosh）

jmpStart.jsl スクリプトは、コンピュータ上の特定のユーザに対してのみ実行されます。jmpStartAdmin.jsl という名前をつけたスクリプトを、使用しているオペレーティングシステムに応じて次のいずれかの場所に格納できます。このスクリプトは、コンピュータ上のすべてのユーザに対して実行されます。JMP はまず管理者のスタートアップスクリプトを検索し、見つかればそれを実行します。次にユーザのスタートアップスクリプトを検索し、見つかればそれを実行します。

1. C:¥ProgramData¥SAS¥JMP¥<*バージョン番号*>（Windows）

   /Library/Application Support/JMP/<*バージョン番号*>（Macintosh）
2. C:¥ProgramData¥SAS¥JMP（Windows）

   /Library/Application Support/JMP（Macintosh）

文字の日付を数値の日付に変換

データテーブルの列において、自分は「数値」としてデータを扱いたいのに、列プロパティで見ると、データタイプが「文字」に設定されてしまっている場合があります。また、JMP において、日付時間形式を列に与えるには、列のデータタイプが「数値」である必要があります。

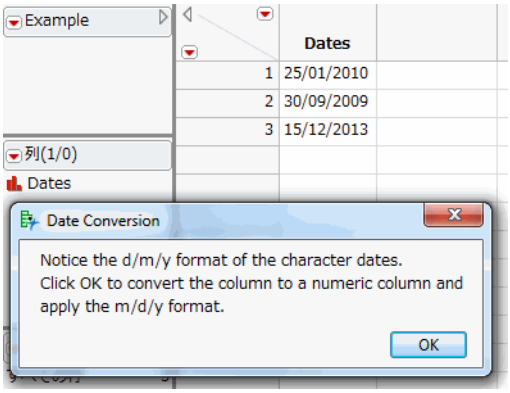
「Convert Dates.jsl」（図 16.1）は、まず、データテーブルを作成します。次に、データ入力形式を指定し、データタイプを「文字」から「数値」に変更します。最後に、その列に、m/d/y 形式を適用します（図 16.2）。

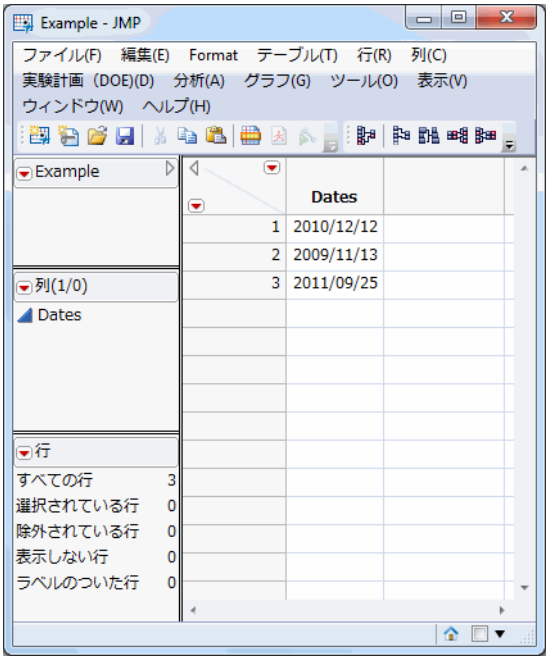
注：データの入力形式を指定せずに列のデータタイプを「文字」から「数値」に変更すると、予期しない結果を招くことがあります。JMP は、列で最初に検出した有効な形式を、残りのセルに適用します。この問題を回避するには、`Informat()` を使ってデータ入力形式を指定します。

図16.1 文字の日付を数値の日付に変換するスクリプト

```
// Create a data table with character dates.
dt = New Table( "Example",
    Add Rows( 3 ),
    New Column( "Dates",
        Character,
        Nominal,
        Set Values( {"12/12/2010", "11/13/2009", "9/25/2011"} )
    )
);

// Display a modal dialog for the user to confirm the format conversion.
nw = New Window( "Date Conversion",
    <<Modal,
    tb = Text Box(
        "Notice the m/d/y format of the character dates.
Click OK to convert the column to a numeric column and apply the y/m/d format."
    )
);

// Convert the character date to numeric date with y/m/d format.
col = Column( dt, "Dates" );
col << data type( "numeric" );
col << modeling type( "continuous" );
col << Format( "y/m/d" ); // Apply this format to the data.
col << input format( "m/d/y" ); // The input format is m/d/y.

// Display the data table in front of the script.
dt << Data Table Window();
```

図16.2 文字の日付の変換（変換前と変換後）





日付によるデータ抽出

次のプログラムは、日付データを扱う例です。JMP では、日付や時刻に関して、多数の機能が用意されています。次の例は、元のデータから、特定の期間のデータだけを抽出する例です。

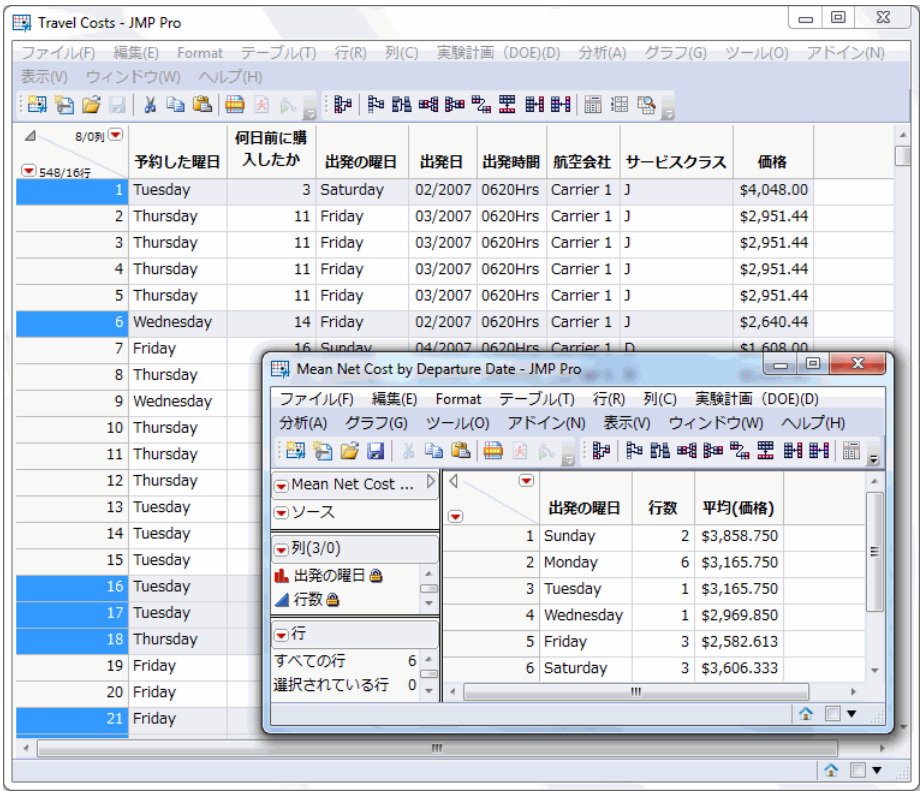
「Select Where Using Dates.jsl」（図16.3）は、出発日（Departure Date）の列をもとに、MDY関数を利用して、特定の期間だけを含んだデータを作成します。そして、そのデータから、正味費用（Net Costs）の曜日（Departure Day of Week）ごとの平均を求めています（図16.4）。

なお、この例では、データテーブルの列名に英語名が使われています。「Travel Costs.jmp」サンプルデータの列には、英語名と日本語名が与えられています。日本語JMP でこのデータテーブルを開いた場合、日本語名が列名として表示されます。しかし、プログラムでは英語名も用いることができます。

図16.3 日付を選択するスクリプト

```
hdt = Open( "$SAMPLE_DATA/Travel Costs.jmp" );

/* Apply the Date MDY format to Departure Date values and then select only February dates. */
hdt << Select Where(
        (Date MDY( 02, 01, 2007 ) <= :Departure Date < Date MDY( 03, 1, 2007 ))
        );

/* Subset the selected rows into two tables: one table contains February
departure dates, the other contains all data for those departure dates. */
nt1 = hdt << Subset( Columns( :Departure Date ),
    Output Table Name( "February Departure Date" ) );
nt2 = hdt << Subset( Output Table Name( "February Data" ) );

/* Create a summary table, grouping mean cost by day of week that departure
took place. */
sumDt = nt << Summary(
    Group( :Departure Day of Week ),
    Mean( :Net Cost ),
    Output Table Name( "Mean Net Cost by Departure Date" )
);
```

図16.4 元のテーブルと最終的な要約テーブル



計算式を含んだ列の作成

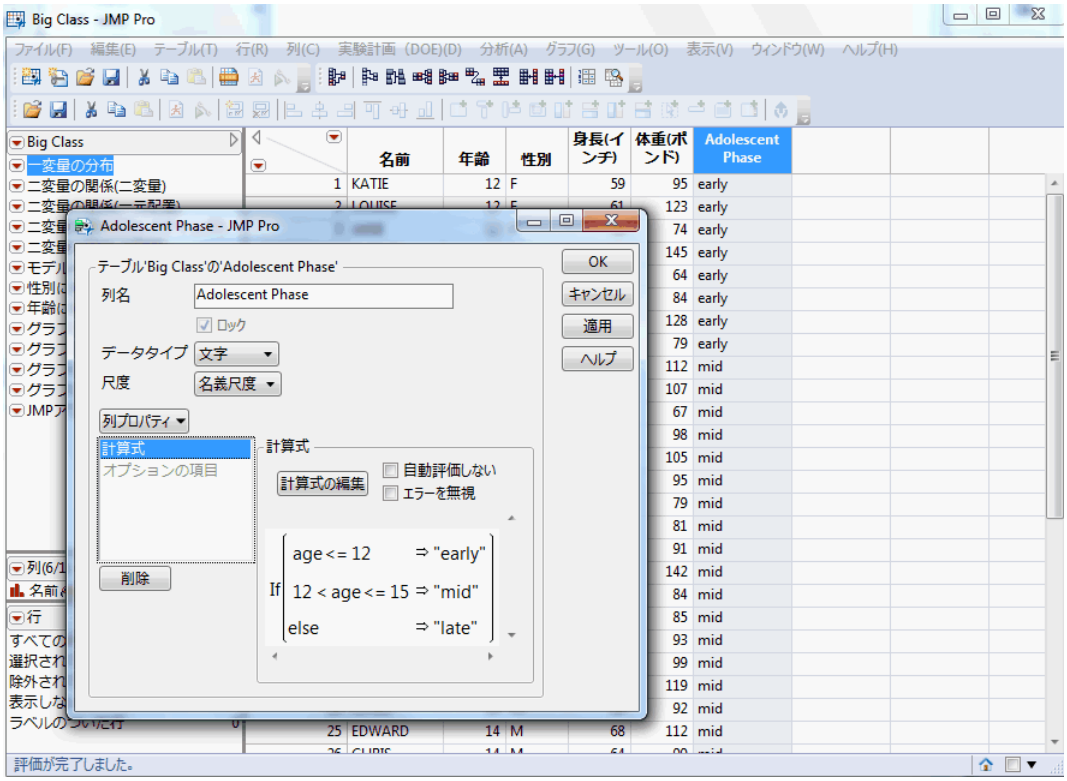
この例では、If 文の計算式を含んだ列を、データテーブルに追加します。「Create a Formula Column.jsl」（図16.5）は、「Big Class.jmp」サンプルデータにおいて、「年齢」(Age) 列に対する If 文の結果を含む列を、新規に作成します（図16.6）。

なお、この例では、データテーブルの列名に英語名が使われています。「Big Class.jmp」サンプルデータの列には、英語名と日本語名が与えられています。日本語 JMP でこのデータテーブルを開いた場合、日本語名が列名として表示されます。しかし、プログラムでは英語名も用いることができます。

図16.5 計算式列を作成するスクリプト

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

/*  Create a new character column for the formula.
    Insert "early" in the new column if the age is less than
    or equal to 12.
    Insert "mid" if the age is less than or equal to 15 but
    greater than 12.
    For ages greater than 15, insert "late".
*/

dt << New Column( "Adolescent Phase",
    Character,
    Formula(
        If( :age <= 12, "early",
            12 < :age <= 15, "mid",
            "late"
            )
        )
);
```

図16.6 計算式内の条件式



分析結果の一部を抜き出す

この例では、レポートの一部だけを抜き出し、別のウィンドウに表示します。

JMP の［分析］と［グラフ］の各メニューで実行されるプラットフォームには、分析層とレポート層という 2 つのオブジェクトがあります。分析層のオブジェクトに特定のメッセージが送られることにより、ユーザの求めている分析が実行されます。

「Extract Values from Reports.jsl」（図 16.7 と図 16.8）は、「二変量の関係」プラットフォームから、標本サイズ、R2 乗、相関などの一部の分析結果だけを取得し、それを別の新たなウィンドウに表示します。図 16.9 は、元の「二変量の関係」レポートと、カスタマイズしたレポートです。なお、スクリプトが終了した時点で、「二変量の関係」レポートのウィンドウは閉じられます。

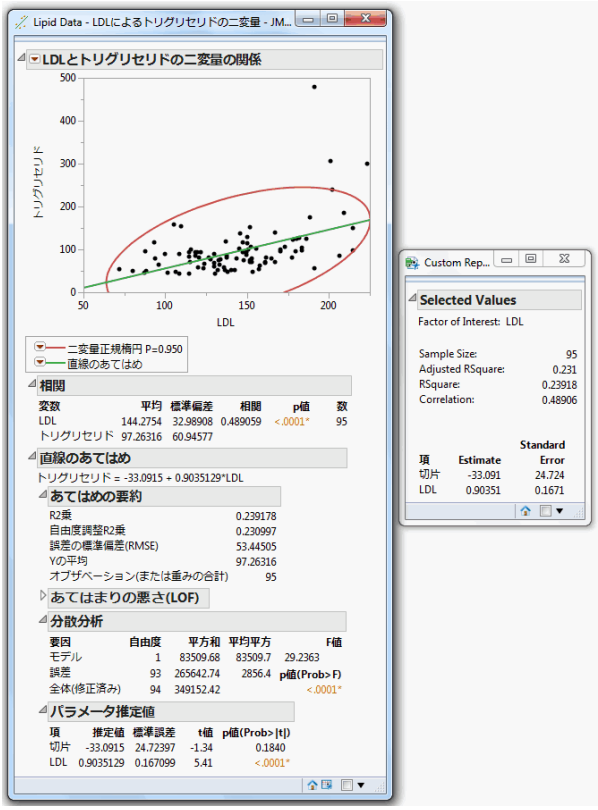
図 16.7 レポートの一部を抜き出すスクリプト（パート 1）

```
sd = Open( "$SAMPLE_DATA/Lipid Data.JMP" );

biv = Bivariate(        //biv is the analysis layer.
    Y( :Triglycerides ),
    X( :LDL ),
    Density Ellipse( 0.95, {Line Color( {213, 72, 87} )} ),
    Fit Line( {Line Color( {57, 177, 67} )} ),
    SendToReport(
        Dispatch(
            {},
            "Correlation ",
            OutlineBox,
            {Close( 0 )}  //Make sure the report is open.
        )
    )
);

reportbiv = biv << Report;  //reportbiv is the report layer.

// The density ellipse is generated first.
// Extract the correlation coefficient.
corrvalue = reportbiv["Correlation"][Number Col Box( 3 )] << Get( 1 );

// ...followed by Fit Line
// Extract the numeric values from the Summary of Fit report
// and place them in a matrix.
sumfit = reportbiv["Linear Fit"]["Summary of?"][Number Col Box( 1 )] << Get as Matrix;

// Extract the values of RSquare and AdjRSquare as one by one matrices.
rsquare = sumfit[1];
adjrsq = sumfit[2];
avg = sumfit[4];
samplesize = sumfit[5];

// Extract the first column of the Parameter.
// Estimates report as two objects.
term = reportbiv["Linear Fit"]["Parameter?"][String Col Box( 1 )] << Get();
```

図16.8 レポートの一部を抜き出すスクリプト（パート2）

```
// Clone the report layer as a String Col Box.
cloneterm = reportbiv["Linear Fit"]["Parameter?"][String Col Box( 1 )] << Clone Box;

// Extract the Parameter Estimates values as a matrix.
est = reportbiv["Linear Fit"]["Parameter?"][Number Col Box( 1 )] << Get as Matrix;

// Extract the Standard Error values as a matrix.
stde = reportbiv["Linear Fit"]["Parameter?"][Number Col Box( 2 )] << Get as Matrix;

dvalues = [];
dvalues = samplesize |/ adjrsq |/ rsquare |/ corrvalue;
sfactor = term[2];

dlg = New Window( "Custom Report",
    Outline Box( "Selected Values",
        /* The Lineup box defines a two-column layout, each of which contains
           a Text Box. */
        Lineup Box( N Col( 2 ),
        Text Box( "Factor of Interest: " ),
        Text Box( sfactor ), ),
        Table Box(
            /* Display an empty string in the first column
               and the text in the second column. */
            String Col Box( "  ",
                {"Sample Size: ", "Adjusted RSquare: ", "RSquare: ", "Correlation:"}
            ),
            // Insert a 30 pixel x 30 pixel spacer between the columns.
            Spacer Box( Size( 30, 30 ) ),
            ,
            /* Display an empty string in the first column
               and the dvalues in the second column. */
            Number Col Box( "  ", dvalues )
        ),
        // Insert a 1 x 30 spacer.
        Spacer Box( Size( 0, 30 ) ),
        ,
        Table Box(
            /* Display the cloned String Col Box followed by a spacer.
                Then insert the Parameter Estimates and Standard Error values. */
            CloneTerm,
            Spacer Box( Size( 10, 0 ) ),
            ,
            Number Col Box( "Estimate", est ),
            Spacer Box( Size( 10, 0 ) ),
            ,
            Number Col Box( "Standard Error", stde )
        )
    )
);

Close( sd ); // Close the data table.
```

図16.9 二変量分析の結果のカスタマイズレポート



対話型プログラムの作成

この例では、ユーザに数値を入力させ、それを元に計算を実行し、結果を新しいウィンドウに表示します。

「Prime Numbers.jsl」（図16.10と図16.11）は、ユーザに整数の入力を促し、素因数分解の結果、もしくは、その整数は素数であるというメッセージを、新しいウィンドウに表示します（図16.12）。このスクリプトでは、異なる種類のディスプレイボックスを整列したり、テキストを結合したり、条件関数を用いたりしています。

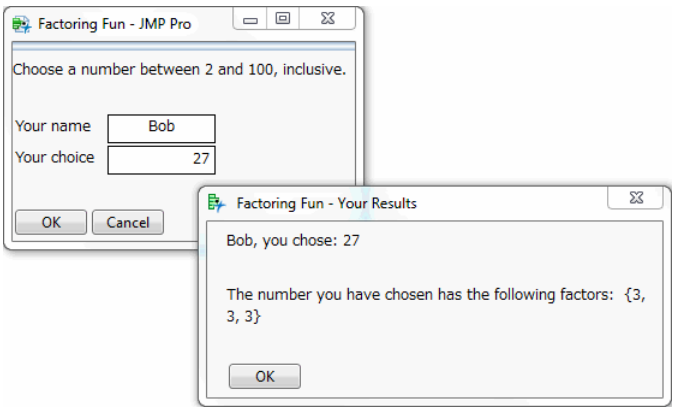
図16.10 対話型プログラムのスクリプト（パート1）

```
nw = New Window( "Factoring Fun",

    V List Box(
        Text Box( "Choose a number between 2 and 100, inclusive. " ),
        Spacer Box( Size( 25, 25 ) )
    ),
    V List Box(
        Lineup Box(
            2,
            Text Box( "Your name   " ),
            uname = Text Edit Box( "< name > ", << Justify Text( Center ) ),
            Text Box( "Your choice " ),
            uprime = Number Edit Box( 2 )
        ),
        Spacer Box( Size( 25, 25 ) ),
        H List Box(
            Button Box( "OK",
            // Unload reponses.
                username = uname << Get Text;
                fromUser0 = uprime << Get;

            // Test input for out of range condition.
                If( fromUser0 <= 1 | fromUser0 > 100,
                // Send message to user that input value is out of range.
                    nw2 = New Window( " Factoring Fun: Message for " || username,
                        <<Modal,
                        Text Box(
                            "The number you chose, " || Char( fromUser0 ) ||
                            " is not between 2 and 100, inclusive.  Please try again.  "
                        ),
                        Button Box( "OK" )

                    )
                ,   // Else the number is within range.
                    // Test for a prime number. If not prime, factor it.
                    // Create a vector which holds the prime numbers within specified range.
                    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
                    67, 71, 73, 79, 83, 89, 97];
                    // Count the number of primes in the vector.
                    p# = N Row( primes );

                    isprime = 0; //Set flag.
```

図16.11 対話型プログラムのスクリプト（パート2）

```
                fromuser1 = fromuser0;   // Make a copy of the value for processing.
                factors = {}; // Initialize list.

                // Process the value by checking for prime then factoring if needed.
                While( isPrime == 0,

                    // Compare value to vector of prime numbers.
                    If( Any( fromuser0 == primes ),
                        Insert Into( factors, fromUser0 ); // If found, place value in factor list.
                        isPrime = 1 // Set condition to exit While loop.
                        ;
                    ); // End For loop.

                    If( isprime == 0,
                        For( q = 1, q <= p#, q++,
                            If( Mod( fromuser0, primes[q] ) == 0,
                                fromUser0 = fromUser0 / primes[q];
                                Insert Into( factors, primes[q] );
                                q = p# + 1 // end if-then loop
                                ;
                            ); //End If loop.
                        ); // End For loop.
                    );  // End If/Then loop.

                ); //end while

                cfUser0 = Char( fromUser1 );
                nf = N Items( factors );
                If( nf >= 2,
                    fmsg = "The number you have chosen has the following factors: ",
                    fmsg = "The number you have chosen is a prime number: "
                );
            // Show message to user about results.
                nw3 = New Window( " Factoring Fun - Your Results",
                    <<Modal,
                    Text Box( username || ", you chose: " || cfUser0 ),
                    Spacer Box( Size( 25, 25 ) ),
                    Text Box( fmsg || " " || Char( factors ) ),
                    Spacer Box( Size( 25, 25 ) ),
                    Button Box( "OK" )
                );
            );
        ), // End the main OK button script.
        // Close the window and the program.
        Button Box( "Cancel", nw << Close Window )
    )
  )
);
```

図16.12 素因数分解の結果



付録 A

互換性に関するメモ
JMP 11 における JMP 10 からの変更点

「寿命の一変量」の Bayes 推定

Bayesian Estimates の引数（Scale Prior の値など）は、すべて指定してください。適切なデフォルト値の特定が難しいため、デフォルトの引数はサポートされなくなりました。

PLS 回帰の負荷量相関図

Correlation Loading Plot() の引数は、プロットで使用される因子の数でなければなりません。これまでは、プロットの表示／非表示を示すブール値を引数としていました。

JMP 11 では、Correlation Loading Plot(1) は 1 つのプロットを作成します。JMP 10.0.2 では、次のスクリプトを実行すると、モデルとして 7 つの因子が選択されました。Correlation Loading Plot(1) は、これらの 7 つの因子を含む散布図行列を作成していました。

```
Open("$SAMPLE_DATA\Baltic.jmp");
obj = Partial Least Squares(
   Y( :ls, :ha, :dt ),
   X(
      :v1, :v2, :v3, :v4, :v5,
      :v6, :v7, :v8, :v9, :v10,
      :v11, :v12, :v13, :v14, :v15,
      :v16, :v17, :v18, :v19, :v20,
      :v21, :v22, :v23, :v24, :v25,
      :v26, :v27
   ),
   Validation Method( Name("Leave-One-Out")),
   Fit( Method( NIPALS ), Initial Number of Factors( 15 ), Correlation Loading
Plot( 1 ) ) );
```

JMP 11 では、プロットの因子数はデフォルトで 2 に設定されています。

管理図ビルダーで点をつなぐ

管理図ビルダーで点をつなぐときに使用するコマンドの場所が変わり、［**点**］>［**点をつなぐ**］になりました。JMP 10.0.2 では、［**折れ線**］>［**点をつなぐ**］と選択していました。

JSL コマンドは Connecting Line(Show Connect Line(0|1)) に変わっています。JMP 10.0.2 での同コマンドは Connecting Line(0|1) でした。

次に、新しい構文の例を示します。

```
dt = Open( "$Sample_Data/Quality Control/Diameter.jmp" );
obj = Control Chart Builder(
   Variables( "Subgroup"(: 日付 ), "Y"(: 直径 ) ),
   Chart(
      Position( 1 ),
      Points( Statistic( "Average" ), Show Points( 1 ) ),
      Limits( Sigma( "Range" ), Show Limits( 1 ), Show Center Line( 1 ), ),
      Connecting Line( Show Connect Line( 1 ) )
   )
);
```

Combo Box() の空の文字列

Combo Box() で、空の文字列がリストの終わりを示すマーカーとみなされなくなりました。文字列でない項目があると、空の選択リスト項目が作成され、ログにエラーメッセージが出力されます。

これまでは、ログにメッセージが出力されることはなく、文字列でない項目は次のいずれかの方法で処理されました。

- 文字列でない項目を無視する
- 文字列でない最初の項目をリスト項目の終わりとする
- 文字列でない値を補間して文字列にする
- サブリストを「list」という項目に変換する

空の行列

Mx0 行列と 0x**N** 行列が、次のようにサポートされるようになりました。

- J(M,0,x)、J(0,N,y)、J(0,0,x) は、欠測値ではなく空の行列を戻します。
- As List(J(0,1,0)) は、エラーではなく、{} を戻します。
- 行数（*M*）または列数（*N*）が 1 以上の空の行列は、[](M,0) または [](0,N) として出力されます。これまでは、[] または [,,,] として出力されていたため、評価ができませんでした。*M* と *N* の両方が 0 である場合、[] が戻されます。
- J(5,0,*x*)*J(0,5,y) は J(5,5,.) となります。これまでのバージョンでは、空の和がチェックされず、結果が J(5,5,0) とされていました。
- []+[] は [] となります。これまでのバージョンでは、[]+[]=[.] でした。
- 一部の JSL 関数は、J(0,0,.) ではなく J(0,1,.) を戻す場合があります。たとえば、Rank() は、*N*x1 ベクトルを作成しますが、この *N* は 0 である可能性があります。

Microsoft Excel ワークシートの読み込み

Windows 上の .xls ファイルの場合は、環境設定で［Excel ファイルを開く方法］が［個々の Excel シートを選択］に設定されていても、Open() の Worksheets 引数が実行されます。これまでは、Worksheets 引数ではなく環境設定の方が優先されていました。

計画行列関数内の欠測値

`Design()`、`Design Nom()`、`Design Ord()` は、欠測値を因子の別の水準として扱うようになりました。たとえば、`Design Nom( ., [.0 1] )` は、`[0,0]` ではなく `[1,0]` を戻します。この変更は、`Design Nom()` の別名である `DesignF()` にも影響します。

プラットフォームのスクリプトにおける引用符なしのキーワードと変数名の名前解決

プラットフォームのスクリプトでは、同じ名前の変数が存在する場合、キーワードを引用符で囲みます。名前を引用符で囲むことで、名前解決のエラーを防止できます。

たとえば、`<<Preselect Role()` の1つの引数は「Y」です。スクリプト内でYを変数としても使用している場合は、この引数を引用符で囲みます。

新しいJSL関数

JMP 11では、次のような関数が新しく導入されています。ユーザ定義の関数を同名で作成した場合は、スコープ演算子を使用しないと呼び出すことができません。

関数について詳しくは、JMPの［**ヘルプ**］>［**スクリプトの索引**］を参照してください。

Alpha Shape
Busy Light
Col Span Box
Collapse Whitespace
Contour Seg
Cytometry Logicle
Cytometry Logicle Inverse
Data Grid Box
Debug Break
Faure Quasi Random Sequence
File Size
Get Color Theme Detail
Get Color Theme Names
H Scroll Box
H Splitter Box
Hist Seg

HP Time
Integrate
Is Directory
Is Directory Writable
Is File
Is File Writable
Logist Percent
Logit Percent
Mandelbrot
MATLAB Connect
MATLAB Control
MATLAB Execute
MATLAB Get
MATLAB Get Graphics
MATLAB Init
MATLAB Is Connected
MATLAB JMP Name To MATLAB Name
MATLAB Send
MATLAB Send File
MATLAB Submit
MATLAB Submit File
MATLAB Term
Open Help
Parallel Assign
Pat Look Ahead
Pat Look Behind
R Control
Random Category
Random ChiSquare
Random F

Random T
Range Slider Box
Return
RunProgram
Set Environment Variable
Set Toolbar Visibility
Shape Seg
Sobol Quasi Random Sequence
Spin Box
Summarize Y by X
Titlecase
Triangulation
V Scroll Box
V Splitter Box
V Standardize
XPath Query

プラットフォーム環境設定の設定

`Platform Preferences()` とその別名である `Set Platform Preferences()` を使用すると、プラットフォームのオプション値の設定や再設定、オプションの有効／無効の切り替えがより柔軟にできます。

- 次の式は、「一変量の分布」プラットフォームの［棒の幅の設定］を有効にし、値を2とします。

  ```
  Platform Preferences( Distribution( Set Bin Width( 2 ) ) );
  ```

- 次の式は、［棒の幅の設定］の値を変更し、さらに、このオプションを無効にします。

  ```
  Platform Preferences( Distribution( Set Bin Width( 2, <<Off ) ) );
  ```

  JMP 10.0.2では、値を設定してからオプションを無効にすることはできませんでした。

- 次の式は、［棒の幅の設定］の値をデフォルト値にリセットし、さらに、このオプションを無効にします。

  ```
  Platform Preferences( Distribution( Set Bin Width( <<Default, <<Off ) ) );
  ```

`Platform Preferences()` のすべてのオプションの詳細は、『スクリプト構文リファレンス』を参照してください。

プラットフォーム環境設定の取得

- Get Platform Preferences() を使って、すべてのプラットフォームまたは特定のプラットフォームで変更されたプラットフォームオプションの値が取得できるようになりました。

```
Get Platform Preferences( <<Changed );
Get Platform Preferences( Distribution( <<Changed ) );
```

- Get Platform Preferences() も、特定のプラットフォームオプションの値が変更されている場合に、その値を取得します。オプションが変更されていない場合は、何も戻されません。

```
Get Platform Preferences( Distribution( Set Bin Width ( <<Changed ) ) );
```

Get Platform Preferences() のすべてのオプションの詳細は、『スクリプト構文リファレンス』を参照してください。

ディスプレイボックスとウィンドウの更新

JMP 11 では、ディスプレイボックスやそれを含むウィンドウのサイズを変更する場合に、既存のスクリプトに及ぶ影響をより細かく制御できるようになりました。

- <<Reshow メッセージは、ウィンドウを直ちに更新します。そのため、ディスプレイボックスに <<Reshow メッセージを送った後、Wait(0) を含める必要がなくなりました。<<Reshow の変更点は、使用する上では気付かない程度のものです。
- 複数のディスプレイボックスをそれぞれ更新し 1 つに組み合わせる場合は、<<Reshow を <<Inval に置き換えてください。<<Inval は、ウィンドウ内のディスプレイボックス領域を無効にします。ウィンドウは、次回、オペレーティングシステムによってウィンドウが更新される際（たとえば、ユーザがディスプレイボックスのサイズを変更したとき）に、更新されます。

インタラクティブなディスプレイボックスにおける Run Script()

Run Script(0|1) は、ディスプレイボックスの変化に反応するスクリプトを <<Set または <<Set Selected メッセージの後で実行するかどうかを制御します。これまで、インタラクティブなディスプレイボックスのスクリプトは、Set メッセージの完了時における実行が一貫していませんでした。

Run Script(0|1) を、Set メッセージの第 3 の引数として指定してください。

```
New Window( "好きな果物を選んでください :",
   db = List Box( {"りんご", "梨", "バナナ"},
   Print( "選んだ果物", db << Get Selected ) )
);
db <<Set Selected( 2, 1, Run Script( 1 ) );
```

Run Script(1) を指定すると、Set メッセージの完了後にスクリプトが実行されます。値に変化がない場合でも実行されます。（ユーザが同じ値を選択した場合は、スクリプトは実行されません。）Run Script(0) を指定すると、スクリプトは実行されません。

インタラクティブなディスプレイボックスのほとんどでは、Run Script() を使用しない場合、スクリプトは実行されません。ただし、List Box() では、以前の動作と同様、デフォルトでスクリプトが実行されます。

スクリプトの実行を示すブール値の引数が廃止されました。次のような構文を使うと、ログにエラーメッセージが出力されます。

```
db <<Set Selected( 2, 1, 1 );
```

「ブートストラップ」の積み重ねたテーブル

「ブートストラップ」は、データテーブルを分割しない場合があります（たとえば、「一変量の分布」の要約統計量）。JMP 10 で作成したスクリプトがテーブルの分割されるものとして書かれている場合は、Bootstrap に次の引数を追加してください。

```
Discard Stacked Table if Split Works( 0 )
```

テーブルの表示形式

`Table Style` と `Table Row Style` は使用できなくなりました。代わりに、`Set Underline Headings` や `Set Shade Cells` といったテーブルプロパティを使用し、ブール値を指定してください。

アプリケーションビルダーの「Table Style」プロパティも削除されています。

JMP で［**ヘルプ**］>［**スクリプトの索引**］を選択し、Table Box を検索すると、すべてのテーブルプロパティを含むリストが表示されます。

付録 B

用語集
用語、概念、表記

| 構文の要約において、「|」は「または」を意味し、複数のオプションを分けるために用いられます。通常、「|」で分かれたオプションは互いに排他的です。つまり、選べるものは1つだけで、いくつも選ぶことはできません。

引数 引数とは、JSLの演算子、関数、メッセージなどの括弧の中で指定するものです。たとえば、`Open("Big Class.jmp")`の引数は、Big Class.jmpです。

多くの場合、引数は、指定された位置によって、その意味が決められます。たとえば、`size(200, 100)`において、**200**と**100**は位置指定の引数であり、第1引数は幅、第2引数は高さと常に解釈されます。**名前付き引数**の項も参照してください。

ブール値。 ブール値とは、はい／いいえのような2値のことです。たとえば、オン／オフ、表示／非表示、真／偽、1／0、または、はい／いいえです。**ブール演算子**は、真か偽か（または欠測しているか）を評価する演算子です。

col 構文の要約で、データテーブルの列への参照を意味する表記です。例: `Column("年齢")`。

コマンド 処理を行うJSLステートメントの一般表現です。このマニュアルでは、**演算子**、**関数**または**メッセージ**のように、具体的な用語の方をできるだけ使っています。

現在のデータテーブル 現在のデータテーブルとは、`Current Data Table()`で指定されているデータテーブルのことです。

現在行 スクリプト操作の対象となる行の番号です。デフォルトではゼロ（行なし）になっています。`Row()`や`For Each Row`などで現在行を設定できます。

データベース この用語は非常に広い意味で使われますが、JMPでは、JSLの`Open Database`コマンドを使いODBCを通じてアクセスできる、あらゆる外部データソース（SQLなど）を意味します。

データフィード データフィードとは、リアルタイムデータを連続して読み込む方法の1つです。リアルタイムデータとは、たとえばシリアルポートに接続された測定機器から読み込んだデータです。

db 構文の要約で、ディスプレイボックスへの参照を意味する表記です。例: `report(Bivariate[1])`

dt 構文の要約で、データテーブルの列への参照を意味する表記です。例: `Current Data Table()`、`Data Table("Big Class.jmp")`。

省略（eliding）演算子 省略演算子は、両側のオペランドをまとめて評価する演算子で、厳密に左から右へ評価する場合とは結果が異なります。たとえば、`12<a<13`は*a*が12～13の範囲にあるかどうかを評価する式で、JMPでは式全体を読み込んでから評価します。<が省略演算子でない場合、式は左から右へ評価さ

れます。たとえば、(12<a)<13は、括弧の中の比較がまず評価され、真の場合は1、偽の場合は0が戻されます。そして、その戻り値が13より小さいかを評価します。そのため、(12<a)<13の結果は、常に1（真）となります。object<<message のように使われる演算子 << も、省略演算子です（これは、Send(object, message)と等価）。

関数 関数名に続く括弧の中に引数または引数のリストをとります。たとえば、二項演算子 + は Add() と等価です。また、ステートメント 3 + 4 と Add(3, 4) は等価です。JSLの演算子のすべてに、等価な関数がありますが、関数の中には等価な演算子がないものもあります。たとえば、Sqrt(a) は関数でしか表現できません。関数をある名前で保存する Function も参照してください。

グローバル変数 グローバル変数とは、セッションの中で存在し続ける名前です。グローバル変数には、たとえば、数値、文字列、リスト、オブジェクトの参照など、さまざまなタイプの値を入れることができます。グローバル変数と呼ばれるのは、特定のコンテキストだけでなく、ほとんどどこででも参照できるためです。

二項演算子 二項演算子は、両側に1つずつオペランドをとるものです。たとえば数値演算3 + 4 の +、代入演算 a=7 の = などです。

左辺値 左辺値（L-value）とは、値を代入することができる式のことです。このマニュアルにおいて、「左辺値」の式とは、現在の値を戻すこともできるが、代入演算によって値を設定することもできる式を指します。たとえば、Row() 関数は、現在行の通し番号を求め、これを x=Row() のように他の変数に割り当てることができます。しかし、Row() 関数は左辺値であるので、代入演算の左辺に置いて、Row()=10 のように値を設定することもできます。

リスト リストとは、項目をいくつも含むデータの型です。リストは、中括弧（{}）を用いた表記か、List 演算子で作成します。リストを使えば、たくさんの項目をスクリプトで一度に扱えます。

matrix 行列は、数値の行と列から成る長方形配列で、JMPのデータの型の1つです。JSLでは、行列は大括弧（[ ]）表記か、Matrix 演算子で作成します。

メッセージ メッセージはJSLのステートメントで、実行できる**オブジェクト**へ送られます。

メタデータ JMPのデータテーブルでは、メタデータとはデータを記述したデータのことです。たとえば、データの発生源、各変数のコメント、データと連携するスクリプトなどです。

マウスダウン マウスのボタンを押すと実行できるイベントです。「Handle」（492ページ）および「MouseTrap」（495ページ）を参照してください。

マウスアップ マウスのボタンを放すと実行できるイベントです。「Handle」（492ページ）および「MouseTrap」（495ページ）を参照してください。

名前 **名前**は、JSLオブジェクトへの参照です。たとえば、グローバル変数に3という数値を割り当てる a=3 というステートメントでは、「a」がグローバル変数の名前です。

名前空間 **名前空間**とは、一意の名前およびそれに対応する値の集まりです。名前空間は、異なるスクリプト間での名前の競合を回避するのに役立ちます。

名前付き引数 名前付き引数は、特定の名前によって明示的に定義される引数のことです。たとえば、Graph Box関数などにおける title("My Histogram") は、名前付き引数です。一方、New Window関数においては、title は位置指定の引数であり、第1引数の位置に必ず指定する必要があります。

ODBC データベース Open DataBase Connectivity(ODBC) はマイクロソフトによる規格です。JSL では、Open Database コマンドを使って、ODBC に対応したあらゆるデータソースにアクセスできます。

obj 構文の要約で、分析プラットフォームへの参照を意味する表記です。例: Bivariate[1]

オブジェクト オブジェクトは、JMP の動的なエンティティで、たとえばデータテーブル、データ列、プラットフォーム結果ウィンドウ、グラフなどがこれにあたります。ほとんどのオブジェクトは、自身に対して何らかのアクションを実行するよう指示するメッセージを受け取れます。

演算子 通常、演算子は1文字か2文字の記号で表されます。例: 加算を示す「+」や、以下を示す「<=」

POSIX POSIX は Portable Operating System Interface の頭字語で、IEEE の登録商標です。POSIX パス名は JMP が動作するどのオペレーティングシステムでも使用できるため、オペレーティングシステムごとにパス構文を使い分ける必要がありません。

接尾演算子 接尾演算子は、1ずつ加算する a++ や1ずつ減算する a-- のように、左側（演算子の前）にオペランドをとります。

予め評価された統計量 一度計算され、以後定数として使われる統計量です。

接頭演算子 接頭演算子は、否定の !a のように、右側（演算子の後）に引数をとります。

参照 スクリプトで表現可能な**オブジェクト**に**メッセージ**を送るために、そのオブジェクトを示す方法です。例:column("年齢")、Current Data Table()、Bivariate[1]。一般に参照は便宜上、**グローバル変数**で保存されます。

行の属性 データ行の属性のあらゆる組み合わせを格納するデータ要素の型のことです。属性には除外する、表示しない、ラベルあり、選択されている、色、マーカー、色の濃淡、色相があります。

スカラー 行列ではない、ただの数値のことです。

スコープ演算子 スコープ演算子は名前を特定のデータの型と解釈させます。たとえば、:name の演算子「:」は *name* が列であるとし、::name の演算子「::」は *name* がグローバル変数であるとします。

トグル ブールコマンドの引数（1または0）を省略した場合、コマンドは設定を切り替えます。つまり、オンがオフ、オフがオンに設定されます。このようなコマンドを繰り返し送ることで、オンとオフの間を往復します。ブール引数を指定すれば、コマンドが明示的にオンまたはオフに設定するので、同じコマンドを繰り返し送っても逆の設定にはなりません。

ベクトル 1列だけ、または1行だけから成る行列のことです。

索引
スクリプトガイド

記号

__ 244
; 82, 86
: 94, 338–339, 496
:: 93–94, 496
:* 161
'" 82
‘ 164
, 81
" 266
) 81
[ ] 153, 156, 385, 425, 458, 475
[...] 83
{ } 147
/! 83
//! 46, 85
\!" 83
\!\ 83
\!0 83
\!b 83
\!f 83
\!N 83, 253
\!n 83
\!r 83
\!t 83
\!U 119
+ 140
<< 34, 259, 357–358, 363, 387, 389
<<Get 445
= 86
== 337
>? 140
>> 140
| 36, 639
|/ 163, 185
|/= 164
|| 141, 163, 185
||= 164
$ALL_HOME変数 120
$DESKTOP変数 120
$DOCUMENTS変数 120
$ENGLISH_SAMPLE_DATA変数 120
$GENOMICS_HOME変数 120
$HOME変数 120
$SAMPLE_APPS変数 121
$SAMPLE_DATA変数 121
$SAMPLE_IMAGES変数 121
$SAMPLE_IMPORT_DATA変数 121
$SAMPLE_SCRIPTS変数 121
$TEMP変数 121
$USER_APPDATA変数 121

A

Action 361
Add From Row States 327
Add Multiple Columns 297
Add Rows 312
Add Script 368
Add To Row States 327
All 110, 160
And、欠測値 111
Any 110, 160
Append 402
Arc 479
ArcBall 507, 510
Arg 206
Arrow 476
As Column 94
As Global 94
As Row State 327, 336

As Table 166
Assign 86
Associative Array 187
automatic recalc 364
Axis Box 453

B

Back Color 487
Background Color 487
Bayesian Estimateスクリプト 631
Beep 253
Begin 511
Begin Data Update 339
Big Class.jmp 255, 362
BlendFunc 530
Border Box 409
Braces.jmp 374
Bring Window to Front 365
Bullet Point 404
Button 449
Button Box 381, 404, 406, 411, 453, 491, 498
By 353, 355–356, 366

C

Call List 507, 520
CallList 510
Caption 252, 254
Char 137, 209–210, 302
Check Box 411, 451
Cholesky 179
Choose 107
Circle 480
clear 509
Clear Column Selection 303
Clear Globals 92
Clear Select 316
Clear Selection 414
Clips1.jmp 373
Close 260, 274, 387
Close Window 364
Coating.jmp 372
col 294, 361
　用語集 639
Col List 449
Col List Box 411
　Get Items 411
Col Maximum 343
Col Mean 343–344
Col MeanとMean 343
Col Minimum 343
Col N Missing 343
Col Number 343
Col Quantile 343
Col Standardize 343
Col Std Dev 343–344
Col Sum 343
Color 502–503, 520
Color By Column 316
Color Of 326–327, 329, 331, 335
Color State 327, 329, 331, 334–335
Colors 316
Column 94, 294
Column Dialog 436, 446–447
Column Name 302
Columns 447
Combine States 327, 329, 331, 478
Combo Box 412, 451
Concat 137, 163–164, 185
Concat Items 138
Concatenate 288
Contains 149
Contour Function 469
Convert File Path 123
Copy From Row States 327
Copy To Row States 327
Current Data Table 271, 341
　用語集 639
CV 343
Cylinder 520

D

Data Browser Box 274
Data Table Window 352, 364
Data Type 304
datafeedのデータ値 542

Day 127
Day Of Week 127
Day Of Year 127
db
ディスプレイボックスの参照 384
用語集 639
.dbfファイルの読み込み 270–271
Delete Column Property 343
Delete Columns 301
Delete Formula 343
Delete Property 308, 343
Delete Rows 313
Delete Table Property 343
Delete Table Variable 343
Delete（ディスプレイボックス）403
Derivative 246, 248
Deselect 387
Design 172
Design F 172
Design Nom 186
DesignNom 172, 185
DesignOrd 173
Det 177
Diag 170
Dialog 446–448, 465
DialogとNew Window 437
Dif 320
Direct Product 174
Disable 537
Disk 520
Dispatch 390
Divide 161
DLL 547
DOE Bayes Diagonal 370
DOE K Exchange Value 370
DOE Mixture Sum 370
DOE Sphere Radius 370
DOE Starting Design 370
DOE Starts 370
Drag Line 496
Drag Marker 496
Drag Polygon 496
Drag Rect 496
Drag Text 496
dt
データテーブルの参照 384
用語集 639

E

Edge Flag 520
Edit Number 450
Edit Text 450
Editable 465
Eigen 178
EMult 161
Enable 515, 519, 527, 529, 537
End 511
End Data Update 339
Eval 203, 205, 209–210, 361
Eval Coord 520
Eval Expr 209–210
Eval Formula 306
Eval Insert 208
Eval List 147, 151, 209–210, 448
Eval Point 520
EvalFormula 342
EvalMesh1 533
EvalMesh2 535
evalと計算式 205
exception_msg 236
Exclude 322
Excluded 327, 329–330
Excluded State 327, 329, 331
Expr 203, 205, 209–210, 214, 218
Expr As Picture 435

F

Factorial 238
Files In Directory 398
Fill Color 487
Fill Pattern 489
First 99
Fitness.jmp 473
Font Color 487
For 395, 426, 468

For Each Row 98, 319, 326
Format 125, 131, 305
Formatメッセージと Format関数 305
Formula 205, 345
Frame Box 454
freeze all 276
freeze frames 276
freeze frames with scripts 276
freeze pictures 276
Frustum 506
Function 237

G

Get 395
Get All Columns As Matrix 165
Get As Matrix 165, 167, 303
Get Column Names 302
Get Data Table 274
Get Data Type 304
Get Format 305
Get Formula 303, 305
Get Items
およびCol List Box 411
Get List Check 307
Get Lock 308
Get MM SAS Data Step for Formula Columns 556
Get Modeling Type 304
Get Name 272, 303
Get Picture 435
Get Properties List 308
Get Property 308
Get Range Check 307
Get Rows Where 165, 315
Get SAS DATA Step for Formula Columns 556
Get Script 307, 341, 364
Get Selected Columns 300
Get Selected Rows 165, 315
Get Table Variable 339
Get Text 420
Get Values 303
Get Window Position 365
Get Window Size 365
GInverse 175
Global Box 381, 404, 406, 408, 454, 491
Glue 98
Go To 300
Go To Row 314
Gradient Function 471
Gram-Schmidt法 181
Graph Box 465, 468

H

H List Box 438
Handle 405, 491–496, 498
Has Data View 273
HDirect Product 174
HeadName 207
Hex to Char 135
Hidden 327, 329–330
Hidden State 327, 329, 331
Hide 322
HierBox 381, 454
HLine 475
HList 449
HList Box 380, 400
Hour 127
Hue State 327, 329, 334–335

I

Icon Box 455
Identity 170
If 104
Ignore Columns 263
Ignore Platform Preference 364
In Days 129
In Format 125
In Hours 129
In Minutes 129
In Polygon 484
In Weeks 129
In Years 129
Include 239
Parse Only 239
Index 159, 171

Insert 213, 217
Insert Into 213, 215, 217
InsertInto 213
Interpolate 108
Intersect、共通の値を調べる 195
Inval 636
Invalid Row Numberエラー 95, 338
Inverse 174–175
Invert Expr 248
Invert Row Selection 314
Invert Selection 282
invisible 362–363
Is Empty 262
Is List 152
Is Matrix 160
Is Missing 112
Is Scriptable 262
ISO 4217コード 133

J

J 170, 185
JMP Version() 115
jmpStart.jsl 621
jmpStart.jsl 621
JMPウィンドウのさくせい 377
JMPスターター 28
JMPスターターウィンドウを閉じる 28
JMPチュートリアル 26
Join 289
Journal Box 412
Journal Window 366, 434
JSL Encrypted 245
JSL Quote 204
JSL、定義 29

L

Labeled 327, 329–330
Labeled State 327, 329, 331
Lag 320
LELE 306
LELT 306
Light 526
Light Model 528
Line 474, 477
Line Stipple 515
Line Style 489
Line Up 449
Line Up Box 414
Line Width 515
LINE_LOOP 513
LINE_STRIP 512
LINES 512
List 147, 152
List Box 414, 449, 455
List Check 306
Load Matrix 525
Load Text File 239
Loc 149, 167
Loc Max 168
Loc Min 168
Loc Sorted 168
Local 90, 92
Local Here 223
Lock 308
Lock Globals 91
Look At 509
LTLE 306
LTLT 306
L-value
　用語集 640

M

Mail 252, 255
Make SAS Data Step 556
Make SAS Data Step Window 556
Map1 533
Map2 535
MapGrid1 533
MapGrid2 535
Marker 335, 477–478
Marker by Column 316
Marker Of 327, 329, 331
Marker Size 477
　任意のサイズ 477
Marker State 327, 329, 331

Markers 316
Match 106
Material 520
MATLAB 562
Matrix 155–156, 475
Matrix Box 400–401, 455
Matrix Mult 161
Max 160
MaxCol 447
Maximize 249
Maximize Window 365
Maximum 343
Mean 343
Microsoft Excelファイルの読み込み 268
Min 160
MinCol 447
Minimize 249
Minimize Window 365
Minimum 343
Minute 127
Modeling Type 304
Month 127
MouseTrap 405, 491, 495–496, 498
Move Rows 316
Move Selected Columns 300
Move Window 364
Mult Matrix 525
Multiply 161
Munger 138–139

N

N Items 152
Name Expr 137, 218
NameExpr 203, 205, 210, 215
names default to here 219
Names Default To Here(1) 90
NaN 84
NArg 206
NCol 159, 319
New Column 295, 345
New Table 260, 263
New Table Variable 340
New Window 352, 400, 465
New WindowとDialog 437
Next Selected 316
NMissing 343
Normal 520
Normal Contour 470–471
NRow 159, 313, 319
Num 210
Number 343
Number Col Box 455
NumDeriv 247–248
NumDeriv2 247–248

O

obj
　オブジェクトの参照 384
　用語集 641
ODBC 553
ODBCデータベース
　用語集 641
On Open 341
On Openスクリプトの環境設定 341
Open 260–261
Open Database 271, 553
OpenGL 501
Or、欠測値 111
Ortho 181, 506
Ortho2D 506
OrthoPoly 182
OR演算子 36
OS 115
Outline Box 380, 386, 400–401, 456
Oval 482

P

Panel Box 415
Parse 209–210
Partial Disk 520
Password 270
Patch Editor.jsl 535
Pen Color 487
Pen Size 490
Perspective 502, 505

pick 537
Pick Directory 397
Pick File 397
　Multiple 398
Pickles.jmp 373
Picture Box 381, 456
Pie 479
Pixel Line To 490
Pixel Move To 490
Pixel Origin 490
Plot Col Box 456
Point Size 515
POINTS 512
POLYGON 512
Polygon 483
Polygon Mode 517
Polygon Offset 519
Pop Matrix 510
Popup Box 415
　例 416
POSIX 123
　用語集 641
Prepend 402
Preselect Role 307
Previous Selected 316
Print 203, 252–253
Print Window 273, 365
Product 102
Push Matrix 510, 523
P管理図 373

Q

QR 182
QUAD_STRIP 513
Quadric Draw Style 521
Quadric Normals 521
Quadric Orientation 521
QUADS 513

R

R_Home
　設定 565
Radio Box 417
Radio Buttons 450
Random Reset 306, 342
Range Check 306
Rank 169
Ranking Tie 169
Rect 481, 486
Recurse 238
recursive、ディレクトリ内のファイルをリストする 399
Redo Analysis 351, 364
Remove 213, 217
Remove From 213, 215, 217
Repeat 139
Report 363, 366, 384, 428
Reshow 387
Reverse 213, 217
Reverse Into 213, 217
Revert 272
Rotate 503, 507, 517
Row 96, 98, 319
Row Legend 465, 473
Row State 326–328, 331
Run Formulas 306, 342
Run Model 368
Run Script() 636

S

SAS Connect Libraries 559
SAS DATA Step for formula columns 556
SAS Model Manager、スコアリングコードの作成 557
SAS Name 557
SAS Open For Var Names 557
SASデータセットの読み込み 269
SASのライブラリとの接続 559
SASマクロ変数 557
SASメタデータサーバー 558
Save 272
Save ByGroup 356
Save Database 554
Save Picture 435
Save Script 363

Save Script for All Objects 352, 364
Save Script to Datatable 341, 351, 364
Save Script to Report 351, 364
Save Script to Script Window 352, 364
Save Script to Window 351
Save Text File 239
Scene Box 501
Script Box 457
Scroll Lock 309
Scroll Window 365
Second 127
Select 387
Select All Matching Cells 315
Select All Rows 313
Select Columns 263
Select Matching Cells 315
Select Rows 314
Select Where 314
Selected 327, 329–330
Selected State 327, 329, 331
Send 259, 295, 357–358
Send To Report 390
Sequence 320
Set Data Table 274
Set Each Value 296, 343
Set Formula 303, 305–306
Set Label Columns 308
Set Lock 308
Set Matrix 166
Set Name 271, 303
Set Property 308
Set Row States 336
Set Scroll Lock Columns 309
Set Selected 299, 422
Set Style 418
Set Table Variable 340
Set Values 296, 303, 404
Set Wrap 404
Setメッセージと Getメッセージ 303
Shade Model 529
Shade State 327, 329, 334–335
Shape 172
Shift 213, 217
Shift Into 213, 217
Show 203, 252
Show Arcball 511
Show Globals 90
Show Properties 260–261, 294, 359, 363, 388, 392
Show Tree Structure 382
Show Window 364
.shpファイルの読み込み 270
Sib Append 403
sigma
　プロパティ引数 311
Simplify Expr 249
Size Window 365
Slider Box 381, 404–405, 408, 491
Solve 175
Sort 286, 341
Sort Ascending 169
Sort Descending 169
Sort List 213, 217
Sort List Into 213, 218
Speak 252–253
Sphere 521
Spline Coef、例 367
Spline Eval、例 367
Spline Smooth、例 367
Split 287
Stack 287
StatusMsg 252, 255
Std Dev 343
Step 109
String Col Box 457
String Col Edit Box 440
Subscribe 292
Subscript 156, 385, 425
Subset 281
Substitute 214–215, 218
Substitute Into 214, 216, 218, 429
Sum 343
Summarize 277, 280, 318
Summary 277, 280
Summation 101
Suppress Formula Eval 342
SVD 180

Sweep 176–177, 185

T

Tab Box 418
　例 418
Table Box 380, 458
Text 486, 503, 522
Text Box 419, 458
Text Edit Box 381, 420, 458
textコマンド 474
thisApplication変数 590
thisModuleInstance変数 590
Throw 235–236
Time Of Day 127
Title 366
title 362–363
Trace 171, 179
Translate 503, 507, 517
Transpose 164, 288
TRIANGLE_FAN 513
TRIANGLE_STRIP 513
TRIANGLES 512
Try 235–236, 262
Type 112

U

Unicode、使用
　JMPで 119
　上付き文字と下付き文字 120
　例 119
Unlock Globals 91
Unsubscribe 292
Update 290, 502
Use Value Labels 304

V

V List Box 438
Value Labels 304
Values 296
VConcat 163–164, 185
VecDiag 171
VecQuadratic 171
Vertex 520
VLine 475
VList 449
VList Box 379, 498

W-Z

Wait 253
Wait、ディスプレイボックス内 636
Washers.jmp 373–374
Webページ、読み込み 269
Week Of Year 127
While 100, 468
Write 203, 252–253
X Function 469
Y Function 468
Y2K日付処理 129
Year 127
Zoom Window 365, 390

ア

[アクション] 261, 360
[アクションの選択] 360
値の色
　プロパティ引数 310
値の順序
　プロパティ引数 309
値のラベル
　プロパティ引数 309
アドイン 609
アドインビルダー 609
アプリケーション
　作成 594–605
　サンプル 593
　スクリプトの記述 602–605
　編集と実行 605
　保存 606
アプリケーションの暗号化 601
アプリケーションビルダー 585, 587
　データテーブルの指定 601
アプリケーションビルダー内の名前空間 590
予め評価された統計量 96, 343

イ

一元配置 357
イメージデータタイプ 435
色 487–488
因子の変更（DOE）
　プロパティ引数 311
因子の役割（DOE）
　プロパティ引数 311
インタラクティブなグラフ 404, 491–498
インプレース演算子 213, 215
インプレースでない演算子 214–215
引用符 266

ウ

ウィンドウ
　作成 377
　操作 377
　メッセージ 363
ウィンドウ、カスタム 399
ウィンドウの検索 390
ウィンドウのスクリプト 377
ウォッチ、デバッガに追加 70

エ

エスケープシーケンス 83, 253
エディタ、テキストのドラッグ＆ドロップ 56
エラー
　デバッガで判別 63
　名前の解決 96–97, 633
　無効な行番号 96
　列名の解決 94–95
エラー処理 235
エラーをスローする 262
エンコーディング 135
演算子 86, 246–248
　等価の関数 86
　優先順位 86
　用語集 641
演算子、定義 34

オ

オートコンプリート 52
応答変数の限界（DOE、満足度プロファイル）
　プロパティ引数 311
大文字 36
オブジェクト 257, 260, 294, 357, 384, 387, 425
　子オブジェクト 359, 363
　用語集 641
オブジェクトとメッセージ、定義 34
オプション 350
オプションの引数、定義 35

カ

カーソルの位置まで実行 69
回帰 183–184
回帰分析 183
改行 83, 266
改行ポイント、制御 404
改行文字 83
回転 178
改ページ 83
数 84, 119
カスタマイズ 463
カスタムウィンドウ 399
カスタムグラフ 465
カスタムプラットフォーム 429
カスタムプロパティ
　プロパティ引数 312
カスタムマーカー 490
下線と変数名 244
括弧の一致 54
括弧の揃え 54
括弧の対応 54
カラーグラデーション
　プロパティ引数 310
カラーテーマ、値の色列プロパティ 310
空の行列 155
空の添え字 338
空のテキスト 438
空の文字列、コンボボックス内 632
環境設定カスタマイズファイル 121
関数
　等価の演算子 86
　ローカル変数 237
関数の定義 34

用語集 640
カンマ
とループ 100
カンマ（,）81
管理限界
プロパティ引数 310
管理図
P 373

キ

キーワードを引用符で囲む 633
機器、接続 543
起動時のスクリプトの実行 621
起動スクリプト 621
逆引用符 164
逆回転 179
逆行列 182
逆行列、更新 182
行ごとの関数 344
行の終わりを示す文字 266
行の順序の水準
プロパティ引数 310
行の属性 321
用語集 641
行の属性スクリプトチュートリアル 337
行の属性の組み合わせ 329
行の例 96
行ベクトル 154
行列
空 155
逆 182
行と列の削除 157
行と列の範囲 159
作成 155
算術 161
算術演算子 246
式からの作成 155
順位 169
数値関数 163
正規化 178
線形システムを解く 174
添え字 156
対角線 164
データテーブル 164–167
手引き 154
転置 164
特殊な作成法 170
並べ替え 169
範囲チェック 160
比較 160
分解 178
用語集 640
列の要約 166
レポートから取得 167
連結 163
論理演算子 160
霧 531
切り換え 318, 357

ク

空白 36, 83–84, 86
空白文字 85
クォート演算子 203
区切り、日付時間 129
グラフィックの基本要素 511
グラフ理論と連想配列 195
グローバル変数 89, 92, 94
インタラクティブな表示要素 406
オブジェクトを参照 357
および expr 204, 215
および Global Box 454
およびインプレース演算子 213
および関数 239
および行列 178
接頭（prefix）演算子 338
非表示 244
用語集 640
列名 338
列を参照 294
グローバル変数の非表示 244

ケ

計算式
および eval 205
ピクチャー 435

評価 342
計算式、列 296
計算式のイメージ 435
計算式のグラフィック 435
計算式の適用範囲指定 95
計算式をイメージで保存 435
計測単位
プロパティ引数 311
結果を戻す 445
結合 82
欠測値
行列 160
欠測値のコード
プロパティ引数 309
現在の行と選択された行 318
現在の行の番号 98, 319
用語集 639
現在のデータテーブル 271
現在のデータテーブル、指定 271
現在のテーブル行 96
減衰、光源 527

コ

コード変換（DOE）
プロパティ引数 310
コールスタック、デバッガ 67
光源の減衰 527
工程能力分析 427
子オブジェクト 359, 363
コマンド 350
用語集 639
コマンド vs. メッセージ 260
小文字 36
固有値分解 178
コロン 93, 339, 496
コンテキスト 306
コンテナ、アプリケーションビルダー 590
コンボボックス 437
コンボボックス、空の文字列 632

サ

最下位 382
[サブテーブル] 261, 360
左辺値 93, 213, 215, 217, 319, 496
左右の引用符 266
算術、行列 161
参照 257, 294, 384, 387, 425
用語集 641

シ

シェープファイルの読み込み 270
時間の単位
プロパティ引数 311
式 212, 218
式、定義 36
式のクォート 203
式の操作 212–218
式を文字列としてクォート 204
軸
プロパティ引数 310
事前計算される統計量
用語集 641
事前計算される統計量と Summarize 引数 343
自動実行スクリプト 621
自動スクロール 593
自動的に実行 46, 85
ジャーナル 434
出力
非表示 363
仕様限界
プロパティ引数 310
条件付き関数 104
条件付きロジック 86
省略演算子 358
用語集 639
初期化されていない変数 113
[新規エンティティ] 360

ス

数値編集ボックス 441
数値列編集ボックス 440
スカラー 155
用語集 641
スクリプト

1つにまとめる 44
新しいデータテーブルの作成 42
暗号化と暗号解読 243–246
オートコンプリート機能 52
関数のツールヒントの表示 52
データテーブルに保存 41
ファイルの読み込み 43
スクリプトエディタ
環境設定 59
フォントの設定 59
スクリプトエディタの環境設定 59
スクリプトの暗号化 243
スクリプトの暗号解読 243
スクリプトの再フォーマット 56
スクリプトの実行 51
スクリプトのスタイル 56
スクリプトのデバッグ 63
[スクリプトの場合のみ] 261
スクリプトのフォーマット 56
スクリプトを再フォーマット、スクリプトエディタ内 56
スケーリング 178
スコープ、定義済み 222
スコープ演算子 93
用語集 641
ステップごとに実行、デバッガ 64
スペース 84
すべて中断 64
スライス行列 157

セ

正規化 178
正規直交化 181
正規表現 143
正規表現、検索 56
西暦2000年対応の日付処理 129
整列ボックス 438
接頭（prefix）演算子 86, 94, 338
用語集 641
接尾（postfix）演算子 86
用語集 641
説明 85
セミコロン 82
とループ 100
選択
四角形のコードブロック 55
[選択肢] 360
選択ダイアログボックス 397

ソ

相違点を要約した行列 292
相対ディレクトリ 122
添え字 148
列の参照 294
添え字、空 338
ソケット
コマンド 551
使用法 550
ストリーム 550
データグラム 550
メッセージ 552
例 550

タ

ダイアログボックス
JMPビルトインダイアログボックスの使用 397
作成 436
選択ダイアログボックス 397
ダイナミックリンクライブラリ 547
縦棒 36
縦方向リストボックス 379
タブ 83
タブボックス 381
タブボックス、スタイルの設定 418

チ

チェックボックス 438
地図の役割
プロパティ引数 311
チュートリアル 26
カスタムレポート 426
行の属性 337
行列 183–184
表示 406
品質管理図 335

プラットフォーム 392
注意事項
OpenGL、転置した行列 523
プラットフォームとレポート 360
中括弧 147
中止、ループ 100
直交多項式 182

ツ

ツールヒント 27
通貨コード 133
通信設定 543
ツリー構造の表示 382

テ

データテーブル
Open で列を指定する 263
アプリケーションビルダーで指定 601
印刷 273
行の属性のスクリプト 321
行列 164–167
計算 343
最後に保存した状態に戻す 272
作成する 263
スクリプトで作成 42
閉じる 274
名前 271
非表示にする 263
開いているかどうかのテスト 262
開く 261
保存 272
読み込み 264
データテーブルの比較 292
データテーブルを非表示にする 273
データフィード
管理図の例 545
データフィードオブジェクト 541
データ読み込みの例 545
メッセージ 543, 546
用語集 639
リアルタイムデータの取り込み 541
データフィード。「通信設定」を参照
データフィルタ
コマンド 321
ローカル 283
データベース
開く 553
用語集 639
テーブルスクリプトの暗号化 245
テーブル変数 339
ディスプレイボックス 377, 379, 384
添え字 386
ディスプレイボックスツリー 382
テキスト改行、制御 404
テキストのドラッグ＆ドロップ、エディタ 56
テキストの編集 440
テキスト編集ボックス 440
パスワードの形式 421
プレースホルダテキスト 420
テキストボックス内の HTML タグ 420
適用範囲が指定された名前 92
適用範囲が指定されていない名前 92
適用範囲が指定されていない列名 95
デザイン行列 172
デバッガ
ウォッチ変数の設定 66
オプション 66
環境設定 66
グローバル変数の表示 66
コールスタック 67
ステップオプション 64
名前空間の変数の表示 66
ブレークポイント 67
変数値 66
ローカル変数の表示 66
ログ 67
デバッガ内の状態 68
デバッグ 85, 498
デフォルト値、連想配列 188
デフォルトのディレクトリ 122
デフォルトの文字エンコーディング 135
電子メールでのテーブルまたはレポートの送信 243
天体球
定義 510
転置 179

点をつなぐ、管理図ビルダー 631

ト

投影
　透視 504
　平行 504
等高線 485
透視投影 504
特異値分解 180
匿名スクリプト 604
トグル
　用語集 641
トラブルシューティング 85, 498
　無限ループ 313

ナ

名前 84
　用語集 640
名前空間
　用語集 640
名前付きスクリプト 603
名前付き引数 35
　用語集 641

ニ

二項（infix）演算子
　用語集 640
二項（infix）演算子 86, 94, 339
二重引用符 82
二重引用符（" "）83
二重下線と変数名 244
二変量 359, 381, 385, 392, 471

ヌ

ヌル 83

ノ

ノート（列）
　プロパティ引数 309

ハ

配合（DOE）
　プロパティ引数 310
パイプ記号 36
パス形式 123
パス変数 120
パスワードで保護されたアプリケーション 601
パスワードの形式 421
パターン、大文字と小文字を区別 143
バックスラッシュ 83
バックスラッシュと感嘆符 83
パネルボックス 439
パラメータ、定義 35
範囲スライダボックス 381, 404–405
範囲チェック 306
　構文 306
　プロパティ引数 309
反復 99, 318
凡例、追加 473

ヒ

比較演算子
　行列 160
　例 109
引数 350
　定義 35
　名前付き、定義 35
　用語集 639
引数、定義 35
引数（Enable コマンド）537
非クォート演算子 203
日付時間
　2 桁の年 129
　区切り 129
日付時間形式 305
日付値
　セレクタ 608
非表示のデータテーブル 273
非表示レポート 362
微分 246, 248
評価機能
　1 次元 532

2次元 535
評価を促す演算子 203
評価を遅延させる演算子 203
表示ツリー 377
ナビゲート 379
表示ツリーの作成 399
表示ツリーのナビゲート 379
開いたデータテーブル
テスト 262
開いているデータテーブル、テスト 262
開く
JMP スターターウィンドウ 28
開く、データベース 553

フ

[ブール] 360
ブール値 353
用語集 639
ファイルパス形式 123
フィールド区切り文字 266
フィールドと行の区切り文字 266
フィールドの終わりを示す文字 266
フォント、スクリプトエディタ内の設定 59
プライベートデータテーブル 273
プラットフォーム
By グループ 353
スクリプト 350
プラットフォームとレポート 385
プラットフォームのスクリプト 347
インタラクティブに作成 351
構文 353
ブレークポイント、デバッガ 67
プロパティ 308
分解 178
分散分析 184
分析プラットフォーム
By グループ 353
スクリプト 350
分布
プロパティ引数 311

ヘ

平行投影 504
ベクトル 154
法線 528
用語集 641
ベジェ曲線 532
変更
Rインストールディレクトリ 565
変数 89
競合の解決 234
グローバル、非表示 244
パスの上書き 122

ホ

法線ベクトル 528
補間 208
保存された式 218

マ

マーカー、カスタム 490
マウスアップ
用語集 640
マウスダウン
用語集 640
マクロ 182, 204, 212, 237
丸括弧 81

ム

無限ループ、停止 100–101

メ

メタデータ 339
用語集 640
メッセージ
Report 384
アクティブなプラットフォーム 357
オブジェクト 257
表示 387, 425
プラットフォーム 350
プロパティの表示 260
用語集 640
列 294

メニューの区切り 411
メニューのヒント 27

モ

モーダルダイアログボックス 436
JMP ビルトインダイアログボックスの使用 397
作成 436
選択ダイアログボックス 397
文字エンコーディング 135
モジュール、アプリケーションビルダー 589, 595
モジュールインスタンス、アプリケーションビルダー 590
文字列 119
文字列編集ボックス 440
モデルのあてはめ 186

ヨ

要素ごと 87
要素ごとの行列演算子 161
横方向リスト 438
横方向リストボックス 380, 438
読み込み
Microsoft Excel ファイル 268
SAS データセット 269
Web サイト上のデータ 269
シェープファイル 270
データベース 271
パスワードで保護されたファイル 270
読み込みスクリプトの作成 43
読み込む
テキストファイル 264
読み込んだデータから引用符を外す 266

ラ

ラジオボタン 439
ラジオボックス 439
ラベル、テキストファイルの列見出し 267

リ

リスト 218
用語集 640
リストから取り出す 448
リストチェック 306
プロパティ引数 309
リストの結合 154
リストの操作 212–215
リストの連結 154
リターン 83, 266

ル

ループ、中止 100

レ

例外 235–237
列 361
Open で指定する 263
値の指定 296
グループ化 297
グループ化解除 297
除外する 263
新規作成 295
数値形式 296
列ごとの関数 344
列数、読み込みの引数 265
列での添え字 295
列内の行への添え字指定 338
列のグループ化 297
列のグループ化解除 297
列の計算式 296
列の参照 294
列の事前指定 447
列の並べ替え 300
列プロパティ 308
削除（JSL）308
取得（JSL）308
設定（JSL）308
列ベクトル 154
列名、適用範囲が指定されていない 95
列名、特殊文字 294
列名の添え字 338
列を除外する 263
レポートから行列を取得 167
レポートの非表示 363
連想配列 187

N Items 190
値の割り当て 187
キーおよび値の削除 190
キーおよび値の追加 190
グラフ理論 195
検索 193
作成 187
作成用関数 189
デフォルト値 188

ロ

ローカルデータフィルタ 283
ローカル引数 237
ローカル変数 89
関数内 237
ログ
埋め込み 63
デバッガ 67
論理演算子
行列 160

ワ

ワイルドカード 387