Paper 307-2010

The Power of Personalisation with JMP®

Ian Cox, PhD, JMP Marketing Manager

ABSTRACT

Whether it's used alone or in conjunction with other SAS® technologies, JMP is often promoted as a point-and-click environment whose unique features allow informed users to get value from their data quickly and easily. Sometimes, though, users are intimidated by the sheer power and versatility of JMP, and the question becomes how it can meet specific needs, experience levels and application-oriented patterns of use. Through a series of examples, this presentation shows how the JMP scripting language, JSL, can be used to unleash the power of personalisation, from simply combining report elements in a desired way to building complete applications.

INTRODUCTION

As we begin, it is worth noting that there is no sense in which this paper can be exhaustive. The examples are intended as exhibits that may help to convince you that personalisation of JMP via JMP Scripting Language (JSL) is worth further consideration or study. By drawing attention to the completeness of JSL and some of its more advanced features, we seek to make the material pertinent for those who are already hard-core developers and programmers, as well as those who are just starting down this path. So although we do reference other useful resources, this paper is necessarily mostly about "what" and "why," with often only broad outlines of "how." The code for most of the examples shown can be downloaded from the JMP File Exchange at: http://www.jmp.com/community/.

JMP® AND STATISTICAL DISCOVERY

As its tag line proclaims, the key strength of JMP lies in its ability to support statistical discovery, the process by which users who understand their data can interact with it quickly and easily to arrive at answers that are valuable in their specific context. JMP is built to support an unfolding rather than preplanned style of investigation, in which the emergence of new and interesting questions through the course of the analysis can be as important as obtaining answers to the questions posed before the analysis started. To support this pattern of use, the functionality of JMP is organised and surfaced through a variety of platforms that, typically, consume a single JMP table to produce a report that the user can interact with further. The report contains hot spot or "red triangle" menu options from which the user selects after an initial review of the output, which is almost always shown in a graphical form. The ability to select and show data conditionally from directly within a report supports statistical discovery. For highly dimensional data, this support is further enhanced because reports are non-modal, and multiple reports generated from the parent table are linked to one another via this table.

As powerful and liberating as it is, this "free-form" usage of JMP as a toolbox can be intimidating for some users, particularly because after 20 years of continuous development JMP is so very functional, and is becoming more so with each release. Perhaps more fundamentally, although statistical discovery involves both exploration ("Uncover Relationships") and modelling ("Model Relationships"), modelling is not always required to find useful and valuable answers from data (see Figure 1).

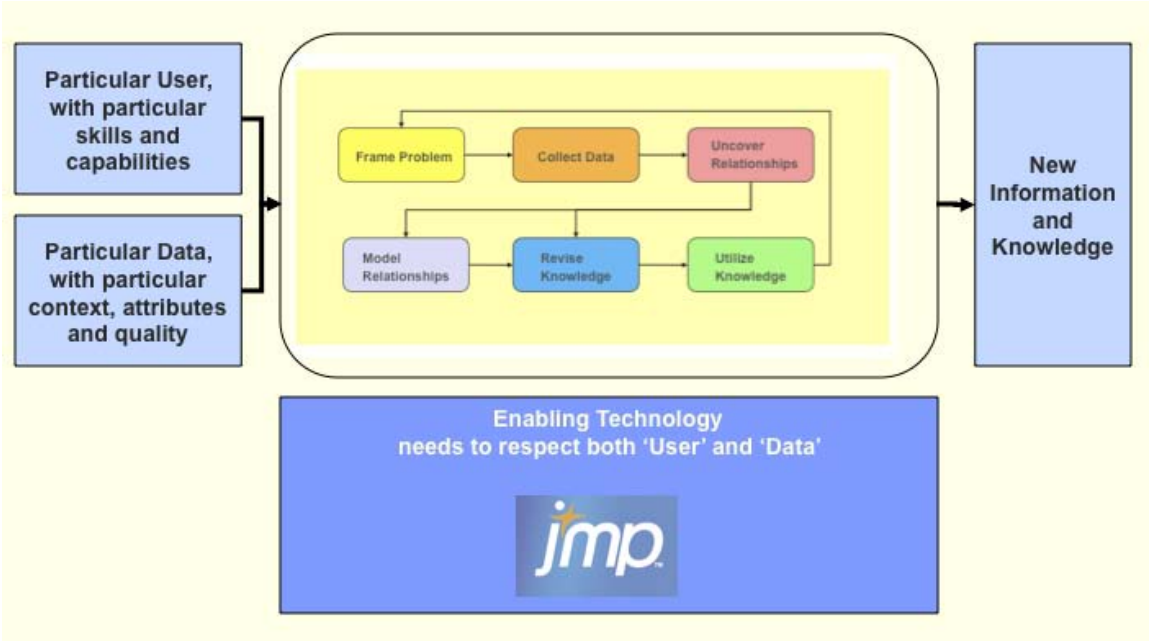

Figure 1. Statistical Discovery as a Transformational Process

No matter how the details are drawn, Figure 1 emphasises that, overall, statistical discovery can be seen as a transforming process with two inputs:

- The user, with particular skills, aptitudes and training.
- The data, with particular dimensionality, structure and quality.

and one output:

- New insight and knowledge useful within the specific context.

Clearly, for this overall process to function well, JMP has to respect both the nature of the user and the data the user want to work with. So, in addition to the obvious use of JMP as a toolbox, we should consider how well we can adapt JMP to more application-oriented usage patterns.

PROGRAMMING, APPLICATION BUILDING, PERSONALISATION AND MASS CUSTOMISATION

The purpose of programming is to create a blueprint for certain desired behaviour in a specific setting, and there is an ongoing debate about the extent to which the writing of programs is an art, a craft, or an engineering discipline [1]. Unfortunately, gaining consensus about exactly what this desired behaviour should be is not always easy, particularly if there are several points of view or if the setting is ambiguous. Furthermore, even when there is consensus about the desired behaviour, there is almost always more than one way to deliver it. Some ways will be good, others less so. Finally, it is almost inevitable that knowledge of requirements and how they can be met changes as implementation unfolds, because it's usually not economical to strive for a perfect plan in advance. All of these issues help to explain the rise in interest in agile development [2], where the application evolves under the influence of the real-world experiences gained as implementation proceeds.

Used originally in relation to Web development, personalisation is narrowly taken to mean that the user's experience of an application is based on attributes such as his department, functional area or role. In contrast, customisation refers to the ability of users to modify the application layout or specify what content should be displayed. As we shall see, JMP can easily support customisation and personalisation based on group membership, through an appropriate mix of implicit personalisation (in which no user intervention is needed) and explicit personalisation (under user control).

Mass customisation has been defined as "a strategy that creates value by some form of company-customer interaction at the fabrication and assembly stage to create customised products with production cost and

monetary price similar to those of mass-produced products." [3] More generally, it is a way to postpone the task of providing differentiating functionality for a specific customer until the latest possible point in the supply network.

Given that:

- JMP installs as a toolbox.
- An agile approach assures that an application is fit for purpose.
- Users seem to value personalisation.

… it is clear that mass customisation is of interest in fostering JMP usage so long as it is technically feasible and easy to accomplish. This is particularly so given the extraordinary inventiveness of users in finding new applications for JMP, and the fact that even if it were possible, meeting all foreseeable needs with a monolithic product would not be economic. Fortunately, and as we hope to demonstrate through the following examples, in addition to the customisation and personalisation mentioned previously, JMP also allows mass customisation by third parties or end users. It is in this more general sense that 'personalisation' appears in the title of this paper, and is used subsequently.

No matter what technical environment JMP is embedded in, and with or without the help of other SAS technologies, personalisation of JMP requires some familiarity with JMP's programming language, usually, but not necessarily, working in conjunction with the built-in platforms. So let's look at some code.

A FIRST EXAMPLE – BUILDING A CUSTOMISED REPORT

Even when JMP is used interactively to make a report from a table, JMP generates JSL code behind the scenes that you can readily exploit. Saving this code allows you to repeat the same analysis, and this is one way to share your results with others. The saved script can also be used with new data so long as the columns to which it refers still exist in the current table. Usually, it is most convenient to save the script to the associated table (using *Script > Save Script to Data Table* from the report's red triangle menu), so that the script is carried along with the table. But to inspect the code itself, using *Script > Save Script to Script Window* opens up a JMP editor window containing the code that has been generated by pointing and clicking. Figure 2 shows the results of doing this by looking at the distribution of the variables *height* and *weight* in the Big Class sample data table.

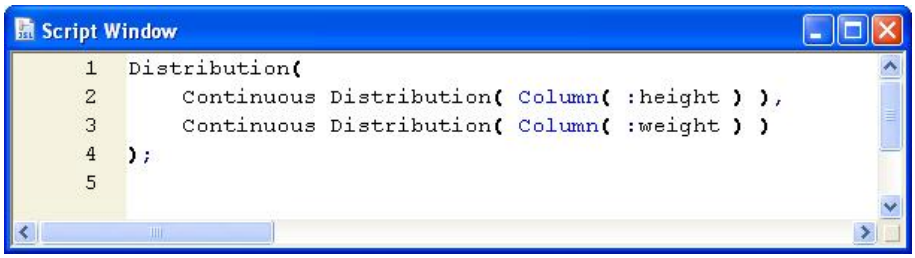

Figure 2. Automatically Generated Code from Distribution

Note these characteristics of JSL syntax:

1. The keyword 'Distribution' originating from the platform used.
2. The arguments bracketed by '(' and ')' and delimited by ','.
3. The prefix ':' indicating a reference to a column in a table.
4. The white space.
5. The terminating ';'.

Figure 3 shows the updated script window after following a similar procedure looking at *height* against *weight* in the *Fit Y by X* platform. Clicking the right mouse button on the script window brings up a context menu containing the command *Run Script*, which just makes two new versions of the Report windows that you already have open.

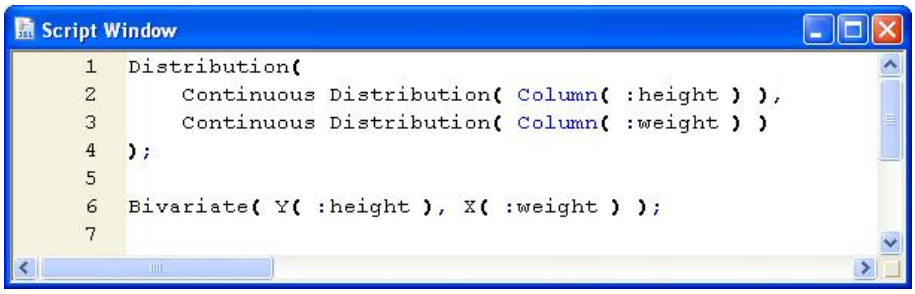

Figure 3. Automatically Generated Code from Distribution and Fit Y by X

But suppose you want the Reports to appear in the same window? Modifying the code in the Script Window as in Figure 4 easily does this. Note that because they are now within the scope of the *New Window* command, the *Distribution* and *Bivariate* expressions now have to be delimited by a ',' which can be effected by modifying No. 5 above. In fact, it is more appropriate to think of ';' as the *glue operator* in JSL, since it is used to glue expressions together. Thus the ';' in line 8 of Figure 4 could actually be removed with no effect because it glues the *Bivariate* expression to the empty expression that follows it.

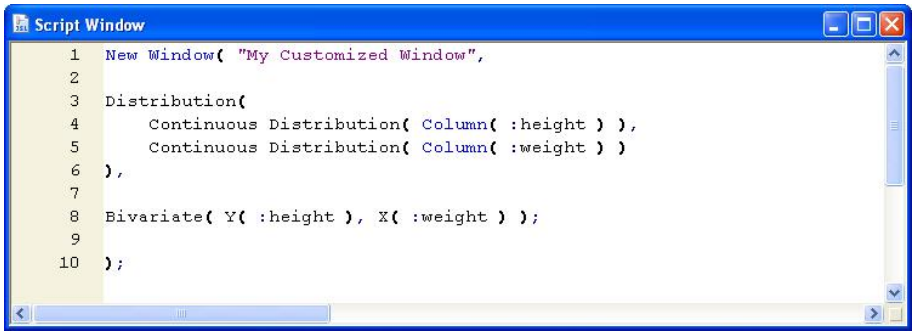

Figure 4. Putting Distribution and Fit Y by X Output in the Same Window

Running this code shows that the two report elements are arranged vertically. To obtain a horizontal arrangement, we need to know about the JSL expression *HListBox* (H stands for horizontal). Making a further addition to the code, and using the *Reformat Script* context menu command gives Figure 5, which achieves the desired effect.

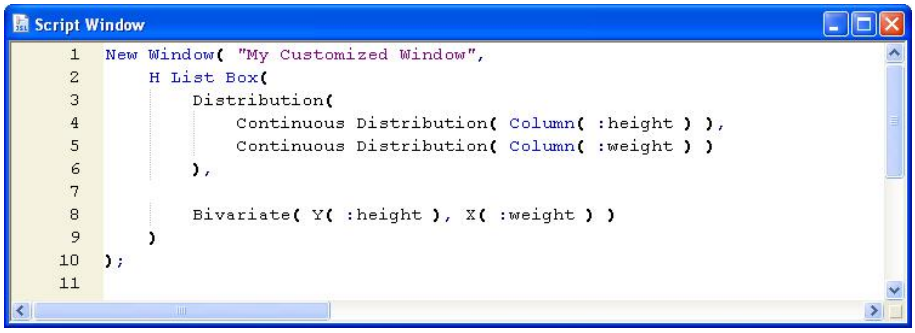

Figure 5. Using HListBox to Make a Horizontal Arrangement

Suppose now that we want the histograms to have the same orientation as their respective axes. A little experimentation with the red triangle menu options in the *Distribution* report soon allows you to come to the code in Figure 6, which gets close. But running the code shows that the alignment of the axes is not very precise. Although this can be improved, closer scrutiny of the options in the *Fit Y by X* report allows you to find the much neater solution shown in Figure 7.

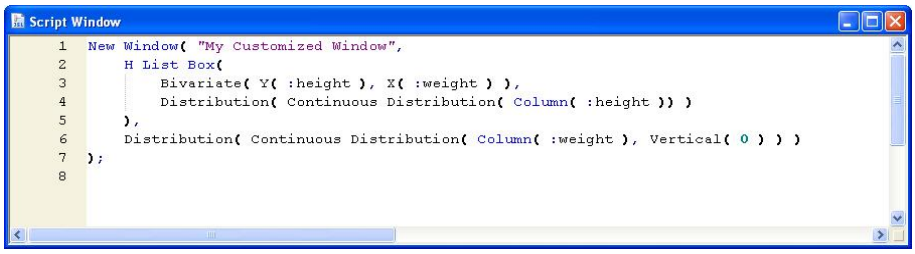

Figure 6. Imperfect Alignment of Axes

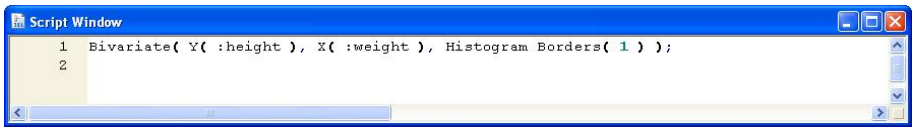

Figure 7. Using Inbuilt Functionality to Align Axes Correctly

Although this example served to introduce the rudiments of JSL and how to build upon code automatically generated by JMP, the moral of the story is really that you are well advised to fully investigate what's already provided in the JMP toolbox! A thorough understanding of what's possible interactively will always stand you in good stead. Similarly, close and repeated scrutiny of *Help* > *Books* > *JMP Scripting Guide* and *Help* > *indices* will pay dividends as you start to write JSL for yourself.

A SECOND EXAMPLE – OFFERING ROLE-BASED FUNCTIONALITY

The initial state of a report window can be customised by each user using the *Platforms* option under *File* > *Preferences.* The options offered here generally map to red triangle menu items in the report itself, so that if for some reason the initial state specified is not actually the most suitable, it can easily be changed interactively. *Edit* > *Customize* > *Menus and Toolbars* does what is described, bringing up a window in which menu branches can be modified or pruned, or new branches or items added (usually giving the user access to new or repackaged functionality involving JSL).

Another way to give access to specific functionality is through a JMP Journal file, as exemplified by the JMP Starter Window (*View* > *JMP Starter*). Using a journal has the advantage that, in comparison with customized menus, it is easier to offer "in-line" commentary or guidance to a user, and to build a more obvious workflow by making appropriate use of the hierarchical outline nodes. Such journals can be built by hand and then made available to a user or group of users, or can be built by JSL if needed. This section looks at both of these options and concludes with a look at what is possible with a little time and energy.

When JMP initialises, it looks for a file called *jmpStart.jsl* in the appropriate *Builtin Scripts* folder of the install. If this file exists, the JSL code therein is executed automatically, which opens up a number of possibilities. For example, the code in Figure 8 can be used to "bootstrap" the current session, in this case by running all the JSL files in a second folder, perhaps on a shared drive that is easier to administer than the C:/ drive of each desktop. Note that the variable *files* in line 9 is returned as a list, a very common data structure in JSL. List items are delimited by ',', and are between an opening '{' and a closing '}'. The *for* construct in line 11 lops over all the items in this list (all the files in the specified folder) and tries to run the code in each one by using *Include*. Note that this approach is far from robust, because the specified folder could contain other types of file, or the JSL in any one file may not be syntactically correct. JSL allows for throwing and catching exceptions, and, as usual, the work that goes into foreseeing and coding for every eventuality is a function of the intended use. Note that *files* is evaluated at run time, giving an easy way to manage the shared initialisation(s) by modifying, adding or deleting the JSL files in the folder referenced.

JMPStart

```
1  // JMPStart.JSL must reside in the 'Builtin Scripts' folder of the JMP install:
2  // For a standard (English) install on Windows the path would be:
3  //              C:\Program Files\SAS\JMP\8\Support Files English\Builtin Scripts
4
5  // Auto-include all scripts in a particular folder
6
7  // Edit this path variable for the local environment
8  SetPathVariable("$JSL_AUTO_INCLUDE", "/D:/JSL_Include");
9  files = FilesInDirectory("$JSL_AUTO_INCLUDE");
10 If (IsEmpty(files), Throw());;
11 for (i = 1, i <= NItems(files), i++, Include("$JSL_AUTO_INCLUDE"||"/"||files[i]));
```

Figure 8. Using jmpStart.jsl to Bootstrap a JMP Session

Suppose that one of the included files contains the code shown in Figure 9, which shows a simple way to get the ID of the named user licensed to use this JMP install. The method relies on parsing the value of *$HOME,* a built-in path variable used by JMP. The code also shows:

- An example of a JSL function definition. The '{}' in line 1 indicates that arguments are passed to a function as a list, in this case with no items since we don't need *getUserID* to take an argument.
- The fact that a JSL function is an example of an expression that can be evaluated. Line 4 prints this expression to JMP's Log Window (check the option *View* > *Log*, then select *Window* > *Log*). To have the ID shown in the log, we need to evaluate the expression (line 5).

getUserID

```
1  getUserID = Function({}, w = Word(3, ConvertFilePath("$HOME"), "/") );
2
3  ClearLog();
4  Print(getUserID());
5  Print(Eval(getUserID()));
```

Figure 9. Crude Way to Get User Information

We shall see that the fact that a JSL expression is any combination of variables, constants and functions, linked by operators, *which can be evaluated,* is a very powerful feature of the language. In this case the expression is simply the definition of a function, but it could be all or part of an entire script. Depending on your prior experience with other programming languages, it's also fair to say that this feature can be confusing at first, so it is worth investing time to understand it fully. The Winter 2010 *JMPer Cable* article, *Expression Handling Functions – Part 1*[1], is an excellent jumping-off point, and tinkering with code that already works is a great way to learn more.

Relying on *$HOME* may not be the most robust solution, so Figure 10 shows an alternative that makes use of the ability to call DLL files from within JMP (requires JMP 8 or a later release). As in the previous version, *getUserInfo* is a function that does not require an argument. The *{Default Local}* indicates that variables defined therein will be local to that function, and in line 16 the '<<' is the JSL syntax for sending a message to an object (in this case, the DLL that was loaded in the previous line). Note also that if a function has more than one output, these are returned as a list. In this case, there are two outputs (line 32), and *EvalList* forces the evaluation of every item in the list so that each value is available to the calling program. Finally, the JSL syntax '[ ]' always denotes a subscript that allows you to index into a list or matrix and retrieve specific items or elements (see line 36).

getUserInfoInJMP8

```
getUserInfo = Function({}, {Default Local},
    // Constants from MSDN for EXTENDED_NAME_FORMAT enumeration
    // See http://msdn2.microsoft.com/en-us/library/ms724435.aspx
    NameUnknown = 0;
    NameFullyQualifiedDN = 1;
    NameSamCompatible = 2;
    NameDisplay = 3;
    NameUniqueId = 6;
    NameCanonical = 7;
    NameUserPrincipal = 8;
    NameCanonicalEx = 9;
    NameServicePrincipal = 10;
    NameDnsDomain = 12;

    securDLL = LoadDll("Secur32.dll", AutoDeclare(false) );
    securDLL << DeclareFunction("GetUserNameExW",
                                Arg( Int32 ),                          // Name format (use code from above)
                                Arg( UnicodeString(80), output ),      // Name buffer
                                Arg( UInt32, update )                  // Buffer length
                               );

    UserName = "";
    Userid = "";

    NameLen = 80;
    securDLL << GetUserNameExW(NameDisplay, UserName, NameLen);

    NameLen = 80;
    securDLL << GetUserNameExW(NameSamCompatible, Userid, NameLen);

    // Return the results as a two item list
    EvalList({UserName, Userid})
);

user = getUserInfo();
Print("User Name is "||user[1]||"   UserID is "||user[2]);
```

Figure 10. Using a Microsoft DLL to Get User Information

Once we have the user information, we can easily take conditional action as we initialise JMP for that user: We could maintain a JMP table that details this action, or access a LDAP server to learn more about that user.

As mentioned above, one scenario is to present a user-specific journal that guides and perhaps constrains that user in an appropriate way. So the question naturally arises as to how such a journal can be built and maintained. To illustrate this, we slightly modify an example from the aforementioned *JMPer Cable* article. Figure 11 shows the desired result, namely a Journal with some buttons that perform specified actions.

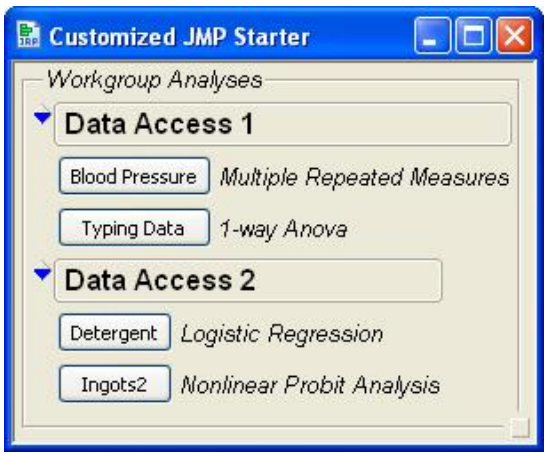

Figure 11. A Simple User Interface in a Journal

The code to achieve (part of) this result by brute force is shown in Figure 12. This shows some new display boxes, including *ButtonBox*, whose second argument is the expression detailing the action to take when the button is pressed (in this case, simply opening a particular file). It's clear that this approach could soon become unwieldy, so take a look at the alternative in Figure 13. The user interface is assembled from the specification detailed in *layout*, which can be easily maintained without affecting the subsequent code. The *addnode* function

iterates through the items in *layout*, using '[]' to index items appropriately. The inner loop over *y* builds a command (an expression . . .) *cmd* that appends a new *ButtonBox* using the << *Append()* message, after which a *TextBox* containing the description of that *ButtonBox* is also appended. Both display boxes are added to a *LineupBox*, and the option *NCol(2)* gives the neat arrangement in two columns. Note that casting *cmd* as an expression is crucial to the correct functioning of the code, else there is a danger that all the *ButtonBoxes* that are built end up opening the same table (the final one referenced as we iterate). For a detailed explanation, please see *Expression Handling Functions – Part 1* [1].

bruteForceUI

```
1  //Brute Force Method
2  NewWindow( "Customized JMP Starter",
3      PanelBox("Workgroup Analyses",
4          OutlineBox("Data Access 1",
5              LineupBox(
6                  NCol(2),
7                  ButtonBox( "Detergent", Open( "$SAMPLE_DATA/Detergent.jmp" ) ),
8                  TextBox( "Nominal Logistic Regression" ),
9                  ButtonBox( "Ingots2", Open( "$SAMPLE_DATA/Ingots2.jmp" ) ),
10                 TextBox( "Logistic Regression" )
11             )
12         )
13     )
14 );
```

Figure 12. Brute Force Way to Produce Figure 11

elegantUI

```
1  // Create a list of lists that specifies the desired UI:
2  // Note the way the data is structured. The innermost lists each have two items,
3  // the first will be the name that appears on a script button the user can press
4  // the second the text of a description of what that button does.
5  layout =
6  {
7      {"Data Access 1",                                    // Outline node name
8          {"Blood Pressure", "Multiple Repeated Measures"},
9          {"Typing Data", "1-way Anova"}
10         },
11     {"Data Access 2",                                    // Outline node name
12         {"Detergent", "Logistic Regression"},
13         {"Ingots2", "Nonlinear Probit Analysis"}
14         }
15 };
16
17 // Function to dynamically build display boxes for a Journal:
18 //      ref is a reference to a 'containing' display box, specifically a PanelBox
19 //      lst is a list with the data structured as above
20 addnode = Function( {ref, lst},
21                 For( x = 1, x <= NItems(lst), x++,
22                     ref << Append( OutlineBox( lst[x][1], lbx = LineupBox( NCol(2) )) );
23                     For( y = 2, y <= NItems(lst[x]), y++,
24                         tableToOpen = "$SAMPLE_DATA/" || lst[x][y][1] || ".jmp";
25                         cmd = Expr( lbx << Append( bbx = ButtonBox( lst[x][y][1], Open(Expr(tableToOpen)) ) ) );
26                         Eval(EvalExpr(cmd));
27                         lbx << Append(TextBox(lst[x][y][2]))
28                     );
29                 );
30             );
31
32 //Create a containing PanelBox in a new window
33 New Window( "Customized JMP Starter", pbx = PanelBox( "Workgroup Analyses" ));
34 //Invoke function
35 addnode( pbx, layout );
```

Figure 13. More Elegant Way to Produce Figure 11

To conclude this section, take a look at the screenshots in Figures 14 and 15. These are from a pre-release version of JMP Genomics (which requires JMP 9, due September 2010). Building on the techniques shown already, this JMP Genomics Starter is built (using JSL) by parsing the XML specification held in a *.jmpmenu* file. The starter configures access to JMP Genomics functionality according to user role or persona, and personas can be selected from the drop-down list at the top. New personas can be defined (by the user, if necessary) through a point-and-click interface, and these can be saved and restored.

Figure 14. More Complex UI (Persona 1)

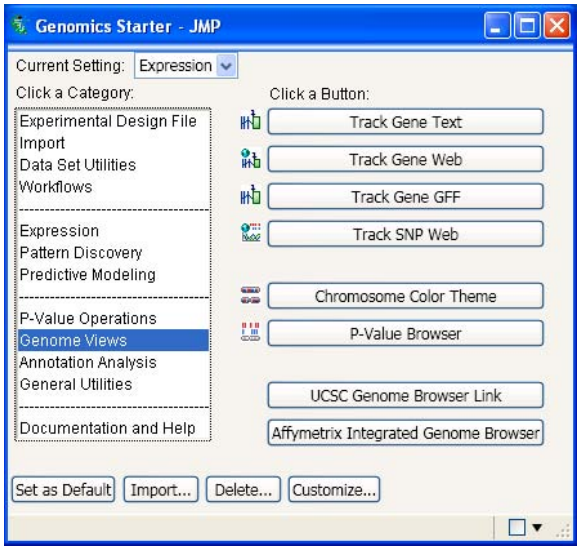

Figure 15. More Complex UI (Persona 2)

A THIRD EXAMPLE – PRE-FILTERING SPECTRAL DATA WITH THE SAVITZKY-GOLAY METHOD

A spectrum consists of a set of intensities measured at various wave lengths or frequencies. Such spectral data occurs in many areas of application, and it is quite common to pre-filter the data prior to any attempts at statistical modelling via techniques such as partial least squares. A variety of filtering and smoothing methods are routinely used, one being the Savitzky-Golay method that replaces each measured value with one interpolated from a local polynomial fit. In this case, there are three parameters that affect the filtering, namely the order of the polynomial chosen and the offset of the left and right edge of the filter window from the point for which we are calculating the filtered value. Any such filtering is inherently subjective, so rather than rely on batch-oriented processing, there is some advantage in providing immediate feedback as the user manipulates the filtering parameters. In other words, this is a natural usage pattern for JMP, but in the context of a specific, albeit small, application.

Although small, the application serves to illustrate some further important features of JSL that enable:

- Building a user interface that allows one to assign columns in a table to specific roles in an analysis.
- Using matrix manipulations in JSL.
- Building a custom graphics display that can be updated under user control.

The application was originally developed as a proof of capability and a learning exercise. It was never really "designed" as such, and no special effort was made to optimise its performance. As with any language, there are many possible approaches, and there is always the fundamental interplay between algorithms and data structures. It has been said that "the only proof is working code," and it is in this spirit that this final example is presented. As mentioned previously, the full code is available for download, so we focus here on briefly discussing code snippets that exemplify the three areas mentioned above that are of general interest. Before doing this, we show briefly what the application does, and in rough terms, how this functionality maps to the program.

Assuming the specimen data table *SG_Data* is active, running the script *Simple_SG.jsl* produces the dialog shown in Figure 16. Apart from the first two columns (which can be ignored), each column in *SG_Data* is a particular wave length, so the spectra are stored row-wise in the starting table. The dialog allows columns selected in the left-hand list to be moved to the right-hand list for analysis (giving the means to use only selected wave lengths). When the user hits the *OK* button, we will need to unload the dialog to find out what they actually chose, and perhaps do some error checking. Note that *SG_Data* contains 392 columns (390 wave lengths) and 47 rows (spectra).

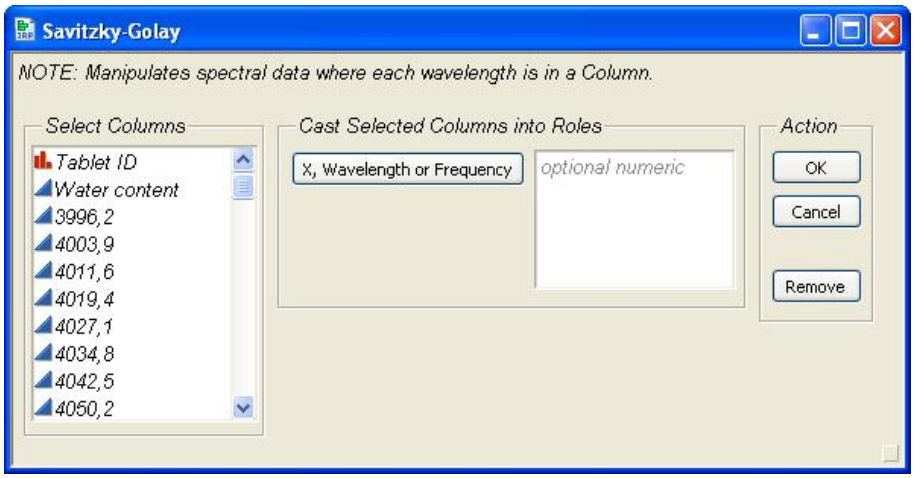

Figure 16. UI to Assign Columns to a Role

Selecting all the columns apart from the first two and hitting *OK* gives three new windows, only two of which are shown here as Figures 17 and 18. Figure 17 shows the measured values of each spectrum as a red line, while the corresponding smoothed values are shown as a blue line. At the bottom of the window are slider bars that allow the user to change the default filter parameters and see all three windows update. Figure 18 shows the first derivative of the filtered values (which can also be used for analysis), and the third window generated shows the values of the second derivative. The button at the bottom of each window allows the user to save the filtered values into a new table for further analysis. Figure 19 shows the effect on the smoothed vales of changing the filter parameters. Note that the regular JMP tools can still be used to manipulate the displays. For example, the user can drag a rectangle with the magnifying glass to zoom in to a particular range of wavelengths.

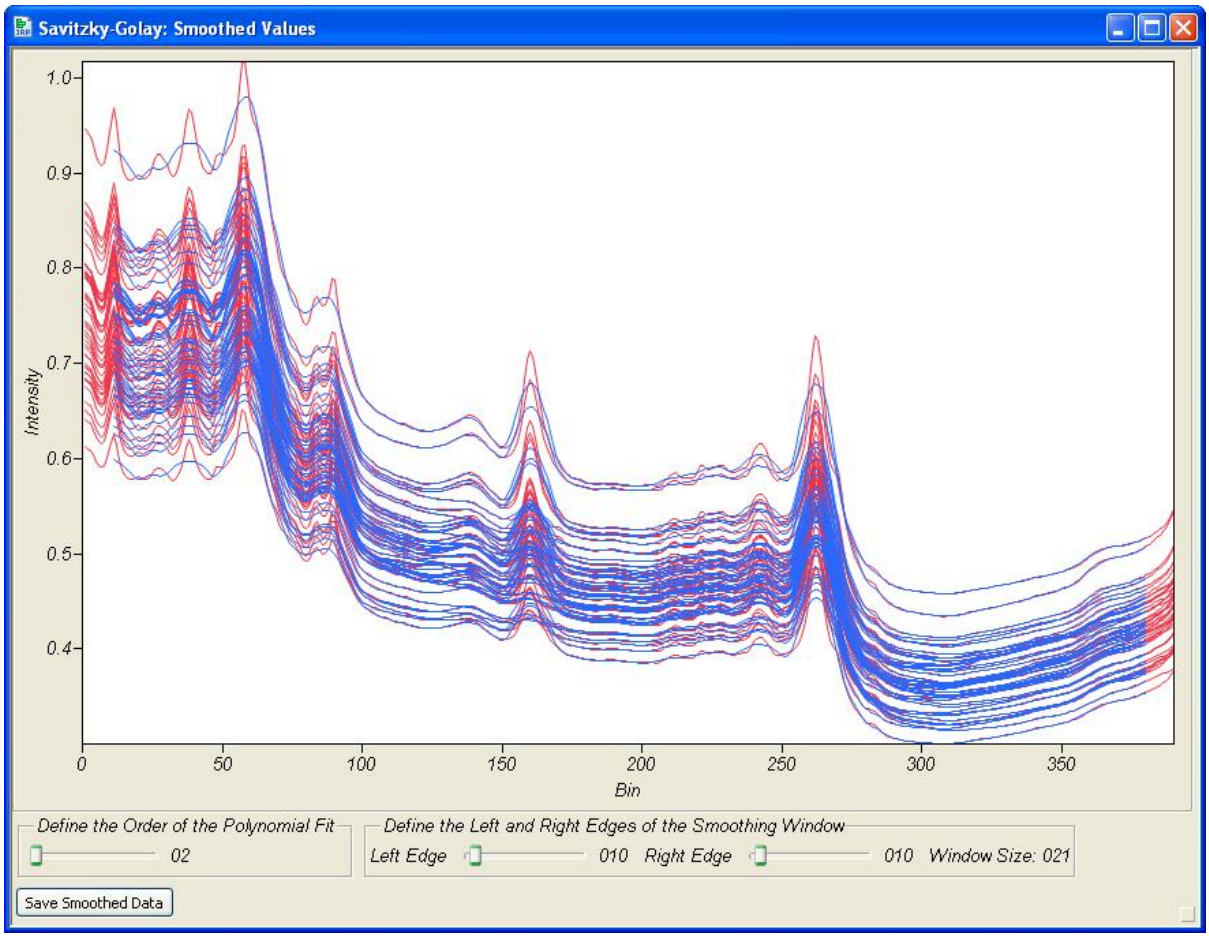

Figure 17. Raw Data and Filtered Spectra (Second Order, Window Size 21)

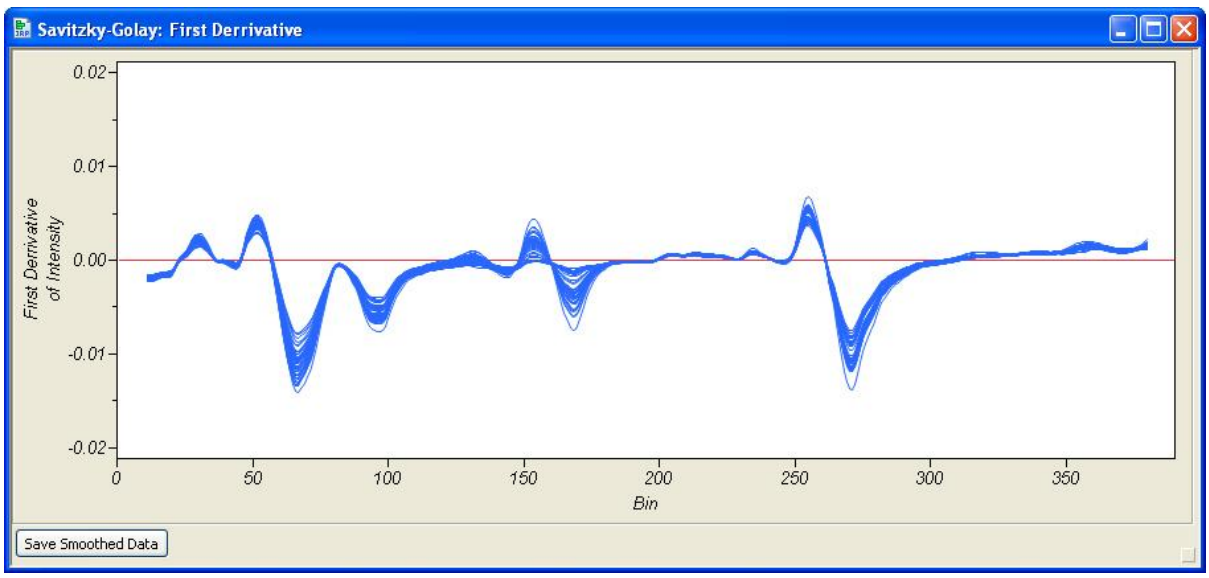

Figure 18. First Derivative of Filtered Spectra in Figure 17

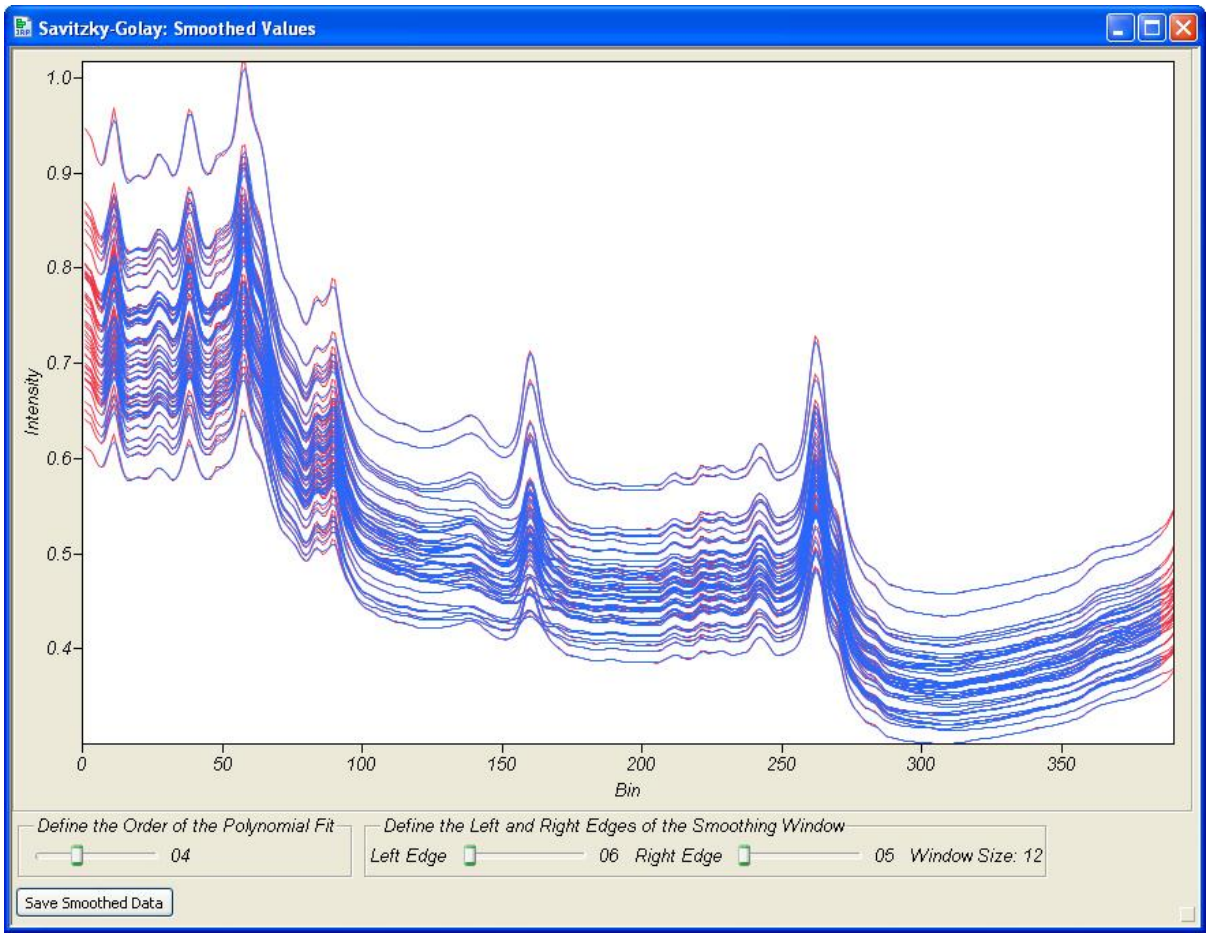

Figure 19. Raw Data and Filtered Spectra (Fourth Order, Window Size 12)

In *Simple_SG.jsl*:

Lines 1 to 95 contain some functions and an expression for later use. The Savitzky-Golay (SG) method itself is implemented via the functions *doSG* and *doSGCoeff*, while matrixToTable is used when the user wants to save the current filtered values, or those of the first or second derivative. The expression *fixWindow* is used to ensure that there are no inconsistencies between the specified size of the filtering window and the polynomial order.

- Lines 98 to 136 produce the dialog shown in Figure 16 and are discussed below (Example 3A).
- Lines 142 to 316 are the expression *OKScript* that is invoked when the user hits *OK* in Figure 16.
- After the dialog choices are unloaded, *OKScript* builds a matrix of the required data and transposes this matrix for use with *doSG* and *doSGCoeff* (lines 157 to 161).
- Using the initial values of the filter parameters, we compute the smoothed values and those of the first and second derivatives (lines 163 to 183).
- The three windows produced are referenced by *nw0*, *nw1*, *nw2*: The expression *refreshScript* (lines 198 to 235) is called when one of the sliders for a filter parameter in *nw0* is adjusted by the user. It interrogates the slider positions to get the new values, recomputes the filtered values and redraws *nw0*, *nw1* and *nw2* accordingly.
- Lines 237 to 274 make the window *nw0*, which uses a graphics script to draw the unfiltered and filtered values as lines (Figure 17). This is discussed further below (Example 3C).
- Similarly, lines 276 to 294 (296 to 314) make the window *nw1* (*nw2*) that contains the plot of first (second) derivative values shown in Figure 18.

EXAMPLE 3A – ASSIGNING COLUMNS TO ROLES

Figure 20 shows the code that produces the dialog in Figure 16. There are some display boxes that should be familiar (*VListBox*, *PanelBox, TextBox*) and some that are new, like *BorderBox* and *ColListBox*. *BorderBox* just provides some visual spacing and *ColListBox* (with the use of the option *All*) lists all the columns in the current data table (in this case referenced as *dt1*, and equal to *DataTable("SG_Data")* if *SG_Data* was selected when the script was run).

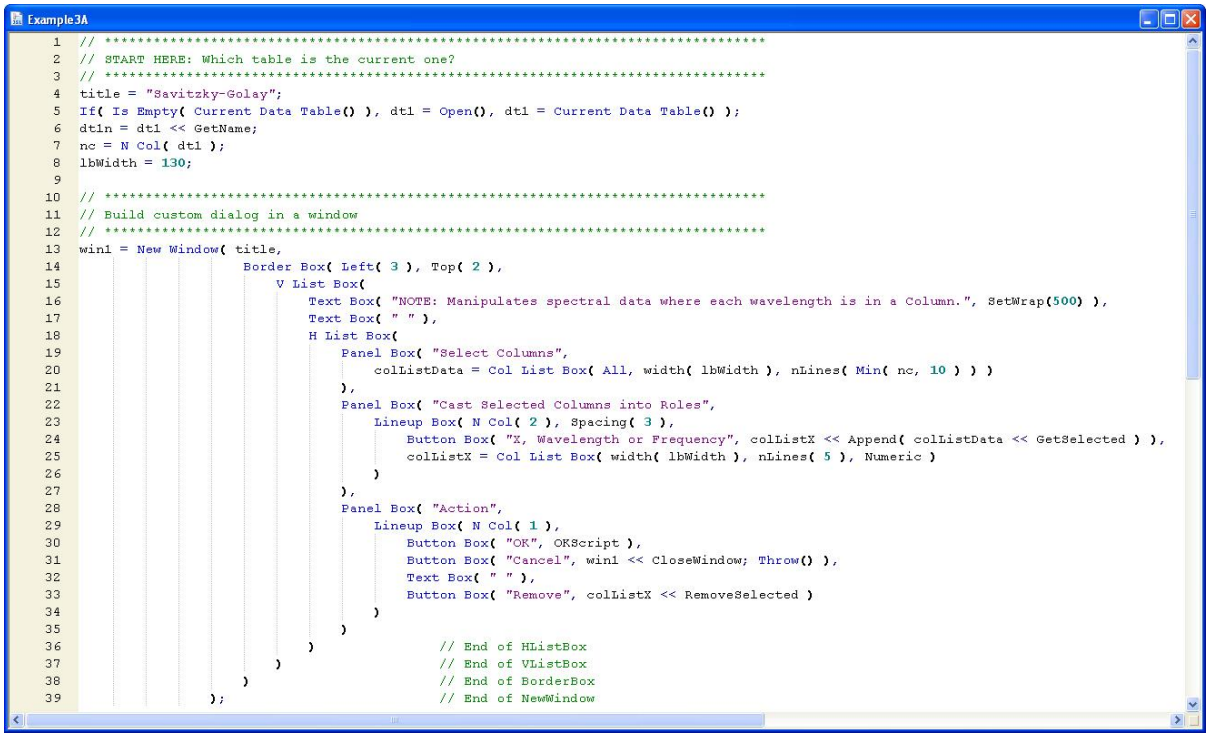

Figure 20. Code to Produce Figure 16

As seen in Figure 16, we need to provide two lists, and the means to shift selected columns between them. In this case the "source" list is called *colListData* (line 20), and the "destination" list is *ColListX* (line 25), which, because of the options chosen, is initially empty. The button labelled 'X, Wavelength or Frequency' is used to move selected columns from the source to the destination list, (line 24) while the button labelled 'Remove' removes selected columns from the destination list should the user make a mistake (line 33). Note that in line 24 the expression tied to the *ButtonBox* sends a << *GetSelected* message to the source list to see which columns are currently selected (highlighted), then adds these to the destination list using an << *Append()* message. This expression could be more sophisticated (for example, updating other UI elements to reflect the columns chosen), but note that we do not need additional code to prevent the occurrence of duplicate entries in the destination list.

The *Cancel* button closes the dialog window referenced as *win1* and terminates the script, whereas the *OK* button calls *OKScript*, which does all the work when the user is happy with the selected columns. OKScript closes *win1*, and unloads the display box *ColListX* into a list called *lx* by sending it a << *GetItems()* message. It then checks to see that at least one wave length column was selected before continuing. Clearly, the error checking could be made more meaningful, and it would be better to associate the corresponding logic with the UI elements in *win1*, rather than dismissing this window first.

When building user interfaces via JSL, all of the visual elements that you see as you click around JMP are available as so-called *display boxes.* For a listing see *Help* > *Indices* > *DisplayBox Scripting*. Each display box is an object that you can manipulate by sending messages to, and display boxes can be arranged in more or less any way to achieve the intended effect. For example, the JMP Scripting Guide contains an example showing how you can build a facsimile of the launch dialog that is shown when you select *Analyze* > *Multivariate Methods* > *Cluster*. Although at first glance complex, building a user interface in JMP is routine in the sense that there are just a few design patterns that you will soon get used to if you need to do it for yourself.

EXAMPLE 3B – USING MATRIX MANIPULATION IN JSL

JSL provides facilities to work with matrices with one subscript (row or column vectors) and two subscripts. Subscripts start from 1 (not 0), and if there are multiple rows, ',' is used as a delimiter between them. Where it makes sense, JSL functions operate with matrices as well as scalars, so if *X* is a matrix, *Power(X, 3)* takes the cube of every element in *X*. There are also many functions that work specifically with matrices: For example *GInverse(X)* gives the Moore-Penrose generalised inverse of *X*. Generally, if the algorithm of interest lends itself to the use of matrices as data structures, then the efficiency gain can be considerable. Usually, if your code contains lots of similar *For()* loops, it may be worth investigating the use of matrices.

Figure 21 shows a slightly modified version of *doSGCoeff()* which computes the SG coefficients used to smooth the data and is one of the functions in *Simple_SG.jsl* involving matrices. As the figure shows, this function takes the three filtering parameters (shown here as *M*, *nL* and *nR*) and *order*, which allows the function to also compute the coefficients used in the calculation of the first or second derivative value. As mentioned before, these function arguments are supplied as the items in a list, and in this case they are all scalars (assumed to be positive). Note that the intention here is not necessarily that you understand how the SG smoothing has been implemented, but rather that you appreciate some of the mechanics of how to work with matrices in JSL.

Example3B

```
1  // ******************************************************************************
2  // Function to get the SG Coefficients:
3  // See page 651 of Numerical Recipes in C (note we use their method, but not
4  // their code!) -
5  //      M is the desired order of the polynomial fit
6  //      nL is the distance to the left edge of the window
7  //      nR is the distance to the right edge of the window
8  //      order is the order of the vector of coefficients to return
9  // (order = 0 gives SG smoothing, order = 1 (=2) gives SG first (second) derrivative).
10 //  Note that, in the matrix X below, order = 0 corresponds to row = 1 and so on.
11 // ******************************************************************************
12 doSGCoeff = Function( {M, nL, nR, order}, {Default Local},
13     v = (-nL :: nR)`;                                          // A useful column vector
14     // Build the Design Marix A[i,j] = Power(i,j) columnwise:
15     // The first column in A, jj = 0 is all unity
16     A = J( nL + nR + 1, 1, 1 );
17     For( jj = 1, jj <= M, jj++,
18         A = A || Power( v, jj )
19     );
20     X = Inverse( A` * A );                                     // The matrix we need
21
22     If (order==0, Print(v, A, X));                             // v, A and X don't depend on order so show them once
23
24     c = J( nL + nR + 1, 1, . );                                // Will be the vector of coefficients
25     For( r = -nL, r <= nR, r++,                                // Calculate and assign each coefficient
26         c[r + nL + 1] = Sum( X[order + 1, 0] :* A[r + nL + 1, 0] );   // Row-wise dot product
27     );
28     If( order == 2, c = 2 * c );                               // Need to multiply by 2! for 2nd derrivative
29     // Return the vector of coefficients
30     c;
31 );
32
33 ClearLog();
34 Show(Round(doSGCoeff(2, 2, 2, 0), 3));
35 Show(Round(doSGCoeff(2, 2, 2, 1), 3));
36 Show(Round(doSGCoeff(2, 2, 2, 2), 3));
```

Figure 21. Compute SG Coefficients

Line 13 constructs a column vector, *v*, which has an integer element for each value in the filtering window (run the code, and look in the JMP Log): The '`' operator forms the transpose of the row vector that the index function '::' produces. Lines 16 and 24 show examples of the '*J()*' constructor used to build a matrix: The first (second) argument is the required number of rows (columns), and the final argument is the value assigned (for the design matrix *A* in line 16 this is '1', while for *c*, the vector of coefficients, this is the missing value '.').

Lines 17 to 19 loop to build the design matrix column wise (using the '||' operator to concatenate columns side by side). And line 20 uses a matrix-specific function *Inverse()* to get a new matrix *X*. Lines 25 to 28 calculate the actual coefficients *c*: The subscript '0' is used to mean a whole column or a whole row (recall that subscripts indexing specific elements start at 1), and the ':*' means "multiply corresponding elements." In conjunction with *Sum()*, this gives each coefficient as the scalar or dot product of a column from *X* and a column from *A*.

Line 30 returns the vector of coefficients we need to the calling script, and line 34 asks for the coefficients needed for a quadratic fit with a symmetric filtering window of size 5 (the *Round()* function just makes the output in the JMP Log easier to read). Similarly line 35 (36) returns the coefficients used to calculate the first (second) derivative from the raw data.

EXAMPLE 3C – BUILDING A DYNAMIC CUSTOM GRAPHICS DISPLAY

Figure 22 shows the code in *Simple_SG.jsl* that makes the window *nw0* (see Figure 17). Note that in this case it is difficult to give code snippet that stands alone. There are some familiar display boxes, and two new ones, *GraphBox()* and *SliderBox()*, which we deal with in turn.

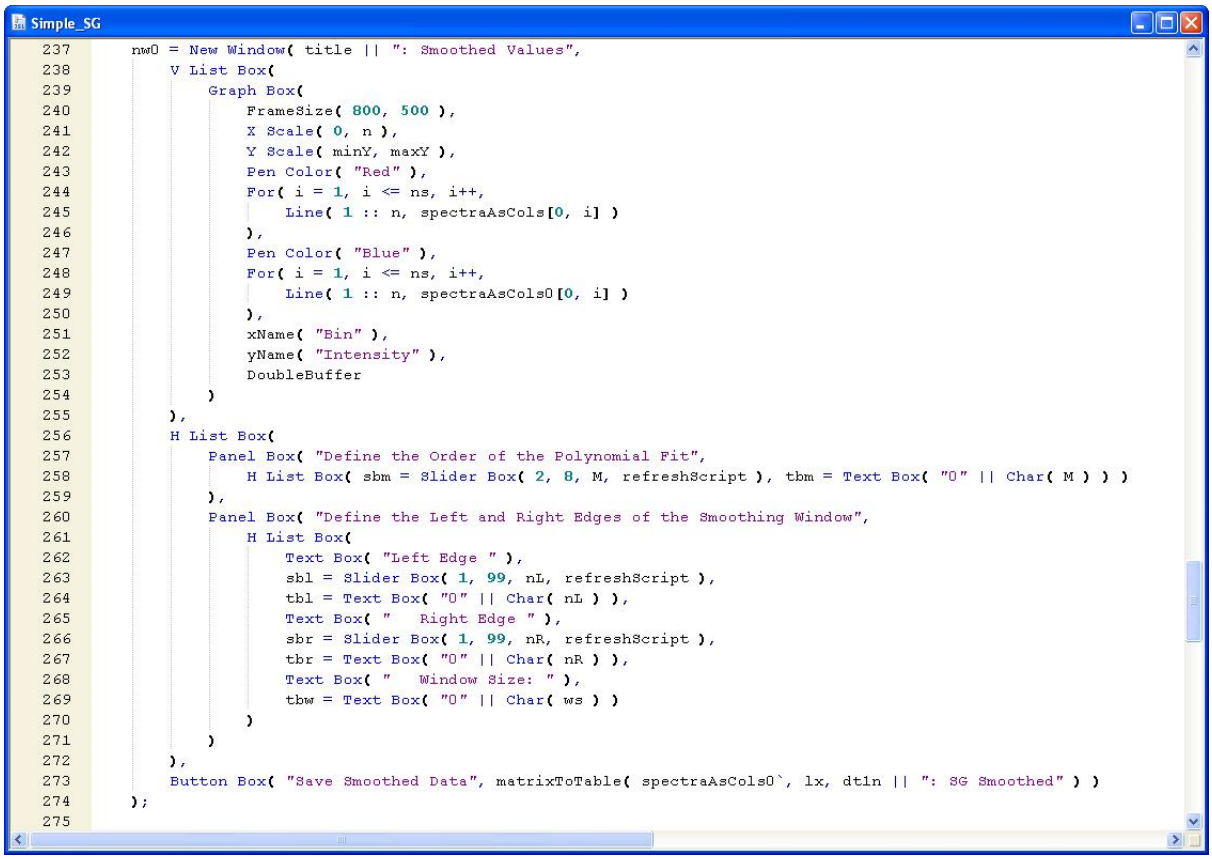

Figure 22. Code to Produce Figure 17

GraphBox() makes the white graphics frame in Figure 17, with the associated axes. JSL provides many graphics primitives that work wither in relation to the axes defined, or, if needed, at the pixel level. Here we just use *Line()* to draw each spectrum and the corresponding filtered values. Note that *spectraAsCols[]* contains the unfiltered data and *sepctarAsCols0[]* contains the filtered data (see line 180 in *simple_SG.jsl*). Furthermore, note that *Line()* can work with matrices, and we use the '0' subscript to access a whole row of values. *DoubleBuffer* (line 253) is used whenever we want a *GraphicsBox()* to update under user control.

SliderBox() (lines 258, 263, 266) allows the user to change the filtering parameters. The first (second) value is the minimum (maximum) permitted value of the global variable in the third position). The expression *refreshScript* (lines 198 to 235) associated with each *SliderBox()* is called whenever a slider is moved. As mentioned above, *refreshScript* looks at the new values of the variables *M*, *nL* and *nR*, recomputes all the required values, then updates the windows *nw0*, *nw1* and *nw2* by sending each a << *ReShow()* message (lines 224, 224 and 226 in *simple_SG.jsl*). Note that we also use a *TextBox()* to the right of each *SliderBox()* to show the current position of that slider.

The *ButtonBox()* in line 273 allows the user to save the smoothed values to a new table. The code that produces *nw1* (*nw2*), shown in lines 276 to 294 (lines 296 to 314) is very similar to that for *nw0*.

CONCLUSION

The three examples here only scratch the surface of what is possible with JSL, but may motivate you to investigate further. Although JMP is rightly held in high regard for its ability to support users in statistical discovery, the use of JSL opens up many new possibilities. In some ways, the use of "S" in JSL is unfortunate, since it can conjure up the idea of a weak language that does little more than capture and replay user actions. On the contrary, JSL is very full-featured, with the additional advantage that it is intimately tied to a very functional product. As a general rule, new functionality added to JMP is made scriptable, so the number of tools in the toolbox is increasing with every release. Also, and as mentioned in connection with the second example, the expression handling in JSL allows a natural way to express a high-level abstraction should you want to take advantage of this in your coding [2].

As well as building user interfaces, manipulating Platforms, Reports and Journals, or building whole new Platforms, JSL can also, amongst other things, parse messy data using Snobol pattern matching [3] and build interactive three dimensional scenes using Open GL [4]. The JMP Editor provides a number of convenience features (syntax highlighting, keyword completion and syntax reminders via tooltips) that make coding easier. JMP version 9 (due September 2010) will have a much more extensive object-scripting index, provide namespaces to support more robust development of an extensive codebase, and introduces an "add-in" architecture that streamlines the deployment of repackaged or new functionality. Perhaps more significantly, JMP 9 will also allow one to submit R code from a JMP session and retrieve the results. This adds to the existing capability of JMP to interoperate with other SAS technologies, and gives many options to distribute the data and analysis tasks at hand in the way that is most suitable and effective in the given setting. Naturally, such interoperation will be scriptable, providing further possibilities to use JMP as a "hub" for analytic applications in a "best of breed" approach.

The general premise of this paper is that personalisation can be important in providing more value to more users. Also, no matter who ultimately carries the cost, the concept of "time to value" is something that cannot be ignored. So probably "more quickly" is required in the previous sentence too. If JMP already has, or can easily access, capabilities "something like" those that are required, then with just a little knowledge, JSL provides the ability to rapidly tap into these capabilities and surface them in a consistent and unified way, often with a high level of interactivity.

REFERENCES

[1]. *Expression Handling Functions – Part 1*, JMPer Cable, Issue 26, Winter 2010.

[2]. *Concepts of Programming Languages*, RM Sebesta, Addison Wesley 1999.

[3]. http://en.wikipedia.org/wiki/Snobol.

[4]. http://en.wikipedia.org/wiki/Computer_programming.

RECOMMENDED READING

Agile development -- http://en.wikipedia.org/wiki/Agile_development.

Mass customization -- http://en.wikipedia.org/wiki/Mass_customization.

OpenGL -- http://en.wikipedia.org/wiki/Open_gl.

(Web pages accessed February 2010)

ACKNOWLEDGEMENTS

Thanks are due to John Schroedel of JMP for the code in Figure 10, and to Russ Wolfinger of JMP Genomics for the code leading to Figures 14 and 15. Finally, thanks also to Joseph Morgan of JMP for authoring the JMPer Cable article in Reference [1] above, and for showing leadership in the use of JSL.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Ian Cox, PhD

Enterprise: JMP Marketing Manager, SAS

Work Phone: +44 1628 4-86933

E-mail: Ian.Cox@jmp.com

Web: www.support.sas.com/visualsixsigma