THE POWER TO KNOW®

Statistical Discovery.™ From SAS.

JMP® Extensibility Synergy with MATLAB

Case Studies Using the JMP® Interface to MATLAB

WHITE PAPER

Table of Contents

John Salmon, PhD, Research Engineer, Aerospace Systems Design Laboratory,
Georgia Institute of Technology

Daniel Valente, PhD, Communications Specialist, SAS

Abstract

Extensibility is one of the hallmarks of JMP. In JMP 11, the provisions exist for integrating JMP with MATLAB. From JMP Scripting Language (JSL), you can send data from JMP to MATLAB, execute MATLAB functions and return results from MATLAB to JMP for further analysis and visualization. This paper will present three case studies that apply this new functionality. The examples shown will illustrate functionality where JMP and MATLAB have features that do not overlap. In the first case, a MATLAB program is executed to transform data from the time to frequency domain and return the results to JMP. The second example shows how to design an experiment in JMP using the Custom Designer, run trials of a simulation model in MATLAB, and then return results for analysis and profiling in JMP. The third case uses a genetic algorithm in MATLAB to optimize a multidimensional objective space interactively from JMP.

Introduction

JMP is a desktop, in-memory statistical analysis and data visualization software product from SAS. In addition to an extensive collection of capabilities for performing statistical analyses and graphing data, it also includes a rich scripting language, JSL. In recent versions of the software, JMP users have been able to connect to other software to extend the capabilities of JMP. Currently, users can connect to SAS®, R and – most recently in JMP 11 – MATLAB, which is the focus of this paper.

MATLAB, a software solution from MathWorks, is a matrix programming language and interactive programming environment for numerical computation, data analysis, visualization, and building custom engineering models and applications. It includes a rich set of add-on modules, known as toolboxes, which give MATLAB users access to functions to perform specialized modeling and design work as needed.

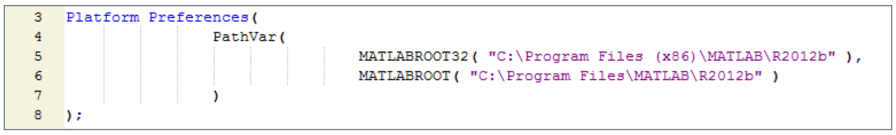
The JMP interface to MATLAB provides access to MATLAB with JSL through a set of functions that will be described in detail in the next section of this paper. The basic execution model is to first initialize a MATLAB connection, perform the required MATLAB operations and then terminate the MATLAB connection. Similar to the R interface in JMP, the connection utilizes a separate instantiation of MATLAB.

Combined with the other development tools within JMP, including the Application Builder and the Add-In Builder, both experienced and novice users of JSL can build custom applications, which can leverage engineering models or functionality in MATLAB. The applications can give colleagues who don't use MATLAB or understand its programming language access to MATLAB models through a JMP application. These applications can be easily deployed across an organization as an add-in.

The three cases highlighted in this paper give the reader an overview of the some of the possible interaction and integration points between JMP and MATLAB. Each example focuses on a single integration paradigm with the idea that each of these JSL programs could serve as a starting point for a more advanced customized application. Each discusses the “what” and the “why,” often with only broad outlines of the “how” (specifically in the details of the MATLAB functionality). We will, however, introduce the scope of the integration interface and present a list of the available JSL functions and the complete JSL code for a simple example in the introductory section. Also, the code for most of the examples presented in this paper can be downloaded from the JMP File Exchange at: jmp.com/fileexchange.

MATLAB Functions in JMP®

MATLAB must be installed on the same computer as JMP, and JMP is not distributed with a copy of MATLAB. Because JMP is supported as both a 32-bit and 64-bit Windows application, you must install the corresponding 32-bit or 64-bit version of MATLAB. JMP delays loading MATLAB until a JSL script requires access to it. When JMP needs to load MATLAB, it is located based on the current search path using a `PathVar` declaration. This should be set using the platform preferences of the JMP interface to MATLAB. Setting the `MATLABROOT` to the path of the MATLAB installation on your computer will make sure JMP knows where to find MATLAB (See Figure 1).



```
Platform Preferences(
                PathVar(
                        MATLABROOT32( "C:\Program Files (x86)\MATLAB\R2012b" ),
                        MATLABROOT( "C:\Program Files\MATLAB\R2012b" )
                )
);
```

Figure 1. MATLAB interface platform preferences.

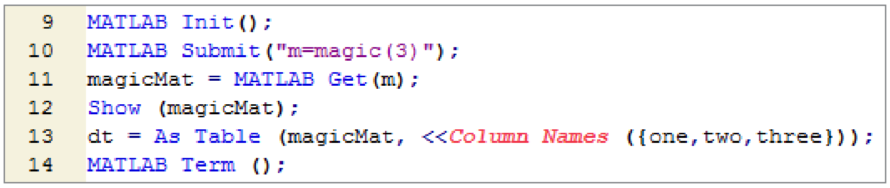
After the path has been set in the platform preferences, you can perform a quick check to make sure that your MATLAB connection is working by running a `MATLAB Init();` function. If the connection has been successful, the function will return 0 to the log. If the operation was not successful, an error message will be returned.

Table 1 shows the available JSL functions for the JMP interface to MATLAB. All of these functions are listed in the Scripting Index within the JMP help system (*Help > Scripting Index*).

Table 1. Listing of available MATLAB integration functions available through JSL.

JSL	Function
MATLAB Connect ( );	Returns a MATLAB connection scriptable object.
MATLAB Control ( );	Changes the control options for MATLAB.
MATLAB Execute ( );	Sends a list of inputs, executes statements and returns a list of options.
MATLAB Get ( );	Returns data from MATLAB, where the name argument can represent any of the following MATLAB data types: numeric, string, matrix, list, data frame.
MATLAB Get Graphics ( );	Returns the last graphics object written to the MATLAB graph display window in a graphics format specified by the format argument.
MATLAB Init ( );	Initializes the MATLAB interface.
MATLAB Is Connected ( );	Returns 1 if there is an active MATLAB connection; otherwise, returns 0.
MATLAB JMP Name to MATLAB Name ( );	Maps a JMP variable name to a MATLAB variable name using MATLAB variables naming rules.
MATLAB Send ( );	Sends data to MATLAB, where the name argument can represent any of the following JMP data types: numeric, string, matrix, list, data table.
MATLAB Send File ( );	Sends a data file to MATLAB where the filename argument is a string specifying a pathname to the file to be sent to MATLAB.
MATLAB Submit ( );	Submit statements to MATLAB. Statements can be in the form of a string value or list of string values.
MATLAB Submit File ( );	Submit statements to MATLAB using a file specified by the path argument.
MATLAB Term ( );	Terminate the MATLAB interface.

A first example to demonstrate how the interface works is shown in Figure 2. This example shows each stage of the interface workflow, so it serves as a good prototype of more complex scripts. The first step is to initialize the MATLAB connection with a `MATLAB Init();` call.

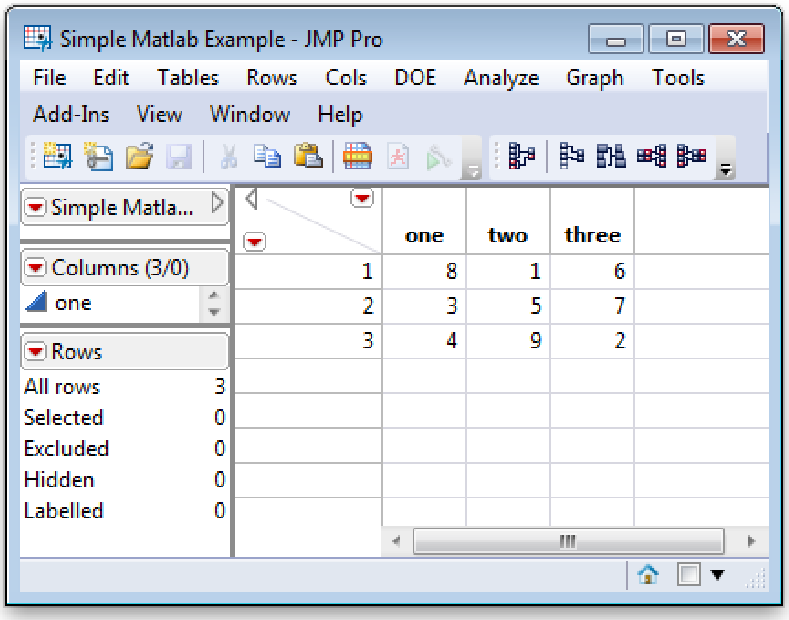


```
MATLAB Init();
MATLAB Submit("m=magic(3)");
magicMat = MATLAB Get(m);
Show (magicMat);
dt = As Table (magicMat, <<Column Names ({one,two,three}));
MATLAB Term ();
```

Figure 2. Simple MATLAB integration example.

Then, we use a `MATLAB Submit();` function to execute some MATLAB code in MATLAB. In this case, we are running the "magic" function with an input parameter of 3 and storing the results into a variable named *m*. In order to bring the *m* matrix into the JMP variable space for further use of the results in JMP, we run the `MATLAB Get();` function, which takes the data stored in *m* and assigns it to a JMP variable, which we have named: *magicMat*. To see the results of our small program, we write the contents of *magicMat* to the log and bring the data into a JMP data table as shown in Figure 3. And finally, we end our MATLAB session with a `MATLAB Term();` command.

The benefit here is that the data from our MATLAB program is now in a JMP data table and ready to be visualized, modeled or explored without any further MATLAB coding. We also gain the interactivity in JMP for exploring MATLAB model output and are able to use the pre-existing JMP tools for data visualization (e.g., via Graph Builder, Control Chart Builder, Distribution), data analysis (Fit Model, Partition, Screening, etc.) as well as further data cleanup and summarization (tools in the *Tables* menu, Tabulate, etc.).

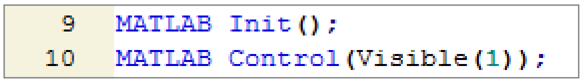


	one	two	three
1	8	1	6
2	3	5	7
3	4	9	2

Rows	
All rows	3
Selected	0
Excluded	0
Hidden	0
Labelled	0

Figure 3. A screenshot of the resulting data table from the example shown in Figure 2.

MATLAB users will be accustomed to working with the Command Window during the course of a MATLAB session. When using the JMP interface to MATLAB, you can still retain this active variable-exploration paradigm while working with your scripts and functions by using the JSL command shown in Figure 4.

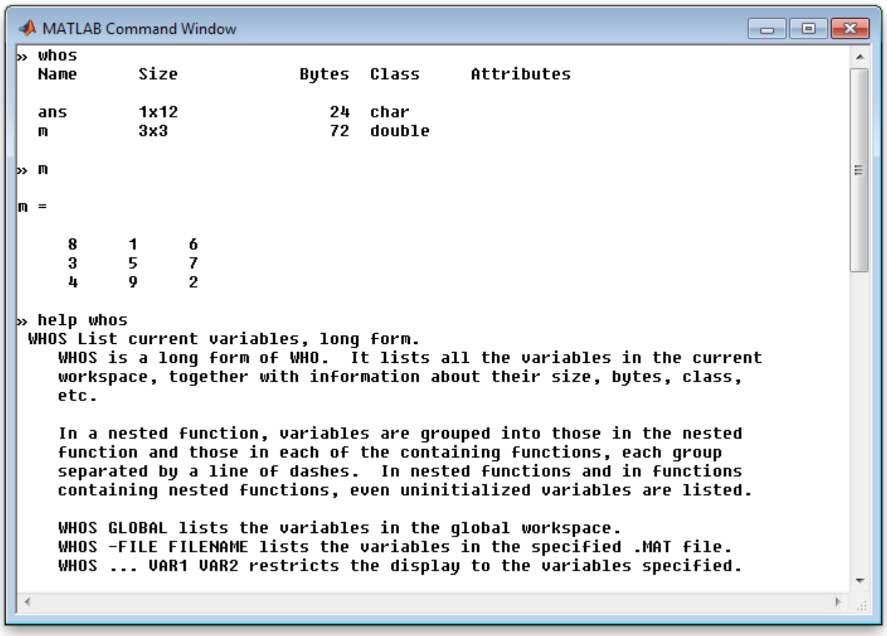


```
MATLAB Init();
MATLAB Control(Visible(1));
```

Figure 4. JSL to instantiate a MATLAB Command Window from JSL.

Executing a `MATLAB Control (Visible(1));` command will show a MATLAB Command Window that you can interact with from the instantiation of MATLAB that JMP creates. The default is to make this invisible so that MATLAB runs completely behind the scenes when called from JMP, but there are often situations during code development where it would be advantageous to use commands run from the Command Window.

Once the Command Window has been brought up, you have access to the full features of the command environment, namely the ability to probe into a variable space, access the help system or execute MATLAB commands. Going back to our simple example, we can examine how this Command Window may be used. Figure 5 shows the MATLAB Command Window accessed through JSL after running the example shown in Figure 2. From this Command Window we can look at the active workspace using the *whos* command, investigate matrix values or access the help system as needed.



```
>> whos
  Name      Size            Bytes  Class     Attributes

  ans       1x12               24  char
  m         3x3                72  double

>> m

m =

     8     1     6
     3     5     7
     4     9     2

>> help whos
 WHOS List current variables, long form.
    WHOS is a long form of WHO.  It lists all the variables in the current
    workspace, together with information about their size, bytes, class,
    etc.

    In a nested function, variables are grouped into those in the nested
    function and those in each of the containing functions, each group
    separated by a line of dashes.  In nested functions and in functions
    containing nested functions, even uninitialized variables are listed.

    WHOS GLOBAL lists the variables in the global workspace.
    WHOS -FILE FILENAME lists the variables in the specified .MAT file.
    WHOS ... VAR1 VAR2 restricts the display to the variables specified.
```

Figure 5. MATLAB Command Window instantiated when running the `MATLAB Control (Visible(1));` command from JSL.

If you want to run a current MATLAB program that requires the use of multiple custom functions in different files within a folder, you need to tell JMP where to access those custom functions. The easiest way to do this is to execute a `MATLAB Submit("CD 'Your FilePath Here'");` statement from JSL. After this has been submitted, you can call any of the included functions from within JSL, now that the current directory has been specified.

The basics of the JMP interface to MATLAB have been shown, and the next sections will explore some possible uses for the interface with three case studies.

Case I: Signal Processing/Fast Fourier Transform (FFT) Application

It is often the case that you need to convert data that represents a time-domain signal to the frequency domain in order to analyze its frequency spectrum. When signal vectors contain tens of thousands of samples, an efficient way to accomplish this Fourier transform is imperative. The Fast Fourier Transform functions in MATLAB accomplish this task and can efficiently perform this transform, even with hundreds of thousands of points of data. This example shows how we can build an application in JMP to send a time-domain signal to MATLAB, perform the FFT, and then return the resulting vector of spectral data back to JMP for visualization and further analysis.

Figure 6 shows an impulse response (IR) measured in a medium-sized office. The resulting data file is 25,000 samples, roughly corresponding to about 0.5 seconds of audio data (at a sampling rate of 48 kHz). The x-axis corresponds to time (displayed in numbers of samples) and the y-axis is the amplitude of the response at that time.

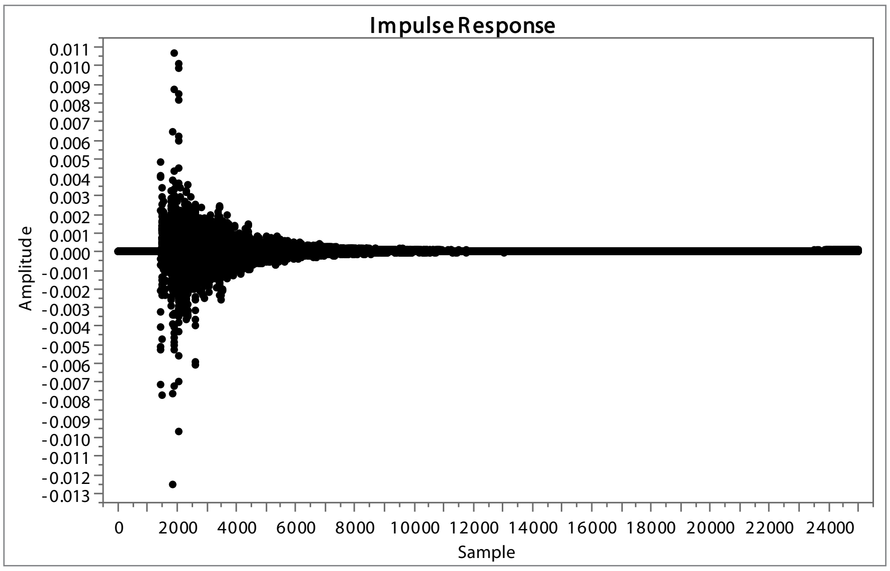


Figure 6. Impulse response data measured in a medium-sized office.

To convert this IR from the time domain to the frequency domain, we can use a pre-existing MATLAB function that we have written, which will convert these data, and also return the proper frequency vector given the sampling frequency of our audio file (so that the resulting x-axis will be correct when plotted).

To make this program more convenient for future use, we can also create a column dialog that will ask the user for the column, which includes the time data that needs to be converted to the frequency domain.

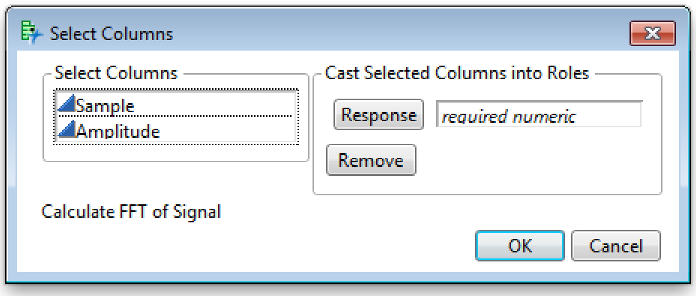
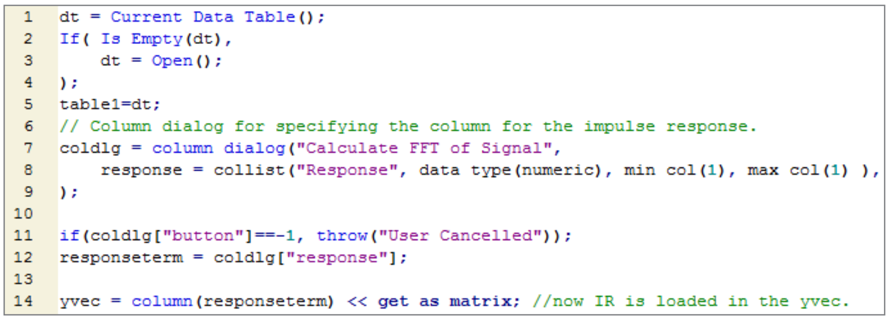


Figure 7. FFT application launcher.

The code to produce Figure 7 is shown in Figure 8. The `Column dialog();` creates a new window that lists all the columns in the current data table. We can use the `collist();` to collect the column that includes the data. This column will contain data in the time domain for converting to the frequency domain in our MATLAB function. Then we store the data in the column that the user selected into a variable called *yvec*.



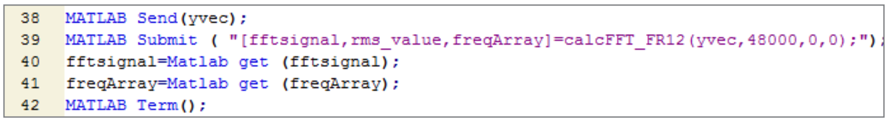
```
dt = Current Data Table();
If( Is Empty(dt),
    dt = Open();
);
table1=dt;
// Column dialog for specifying the column for the impulse response.
coldlg = column dialog("Calculate FFT of Signal",
    response = collist("Response", data type(numeric), min col(1), max col(1) ),
);

if(coldlg["button"]==-1, throw("User Cancelled"));
responseterm = coldlg["response"];

yvec = column(responseterm) << get as matrix; //now IR is loaded in the yvec.
```

Figure 8. Code to produce Figure 7.

The code to run the FFT function in MATLAB is shown in Figure 9. Our *calcFFT_FR12* function takes several input arguments: the matrix that contains the input signal, the sampling frequency and two switches. For this example we have set the switches to 0 because we simply want to calculate the FFT and frequency array vector, then return the results to JMP. The extra functionality built into this program is unneeded for our use here.



```
MATLAB Send(yvec);
MATLAB Submit ( "[fftsignal,rms_value,freqArray]=calcFFT_FR12(yvec,48000,0,0);");
fftsignal=Matlab get (fftsignal);
freqArray=Matlab get (freqArray);
MATLAB Term();
```

Figure 9. Code to run an FFT function in MATLAB.

The function returns three variables: The matrix containing the FFT data, an RMS value and a matrix that contains each FFT bin's frequency value in Hz. In JMP, we are just interested in the `fftsignal` and `freqArray`. We can use the `Matlab Get();` function to bring those variables from the MATLAB variable space to that of JMP. As in our simple example (Figure 2), now that the data is in the JMP variable space, we can bring the data into a JMP data table. Then we can use the Graph Builder to visualize the frequency response of the IR (see Figure 10). As in all JMP platforms, the graphs are interactive, so a region of the response that seems to be more intense in energy than the surrounding frequencies can be selected, and these points are automatically selected back in the data table. Markers or coloring can be added to the data points, and we can drill down into this region to study it in more detail – all without having to write any additional plotting code or run subsequent analyses in MATLAB.

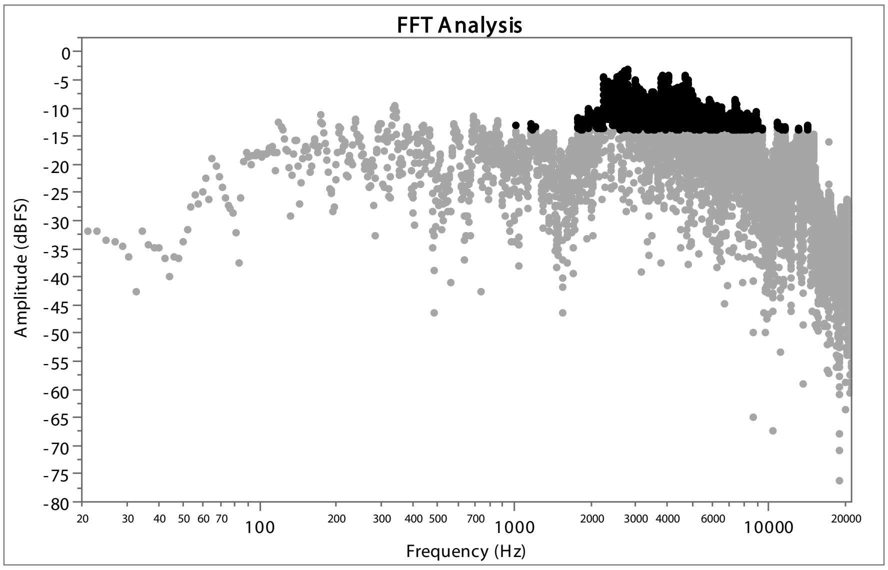


Figure 10. FFT analysis of impulse response data.

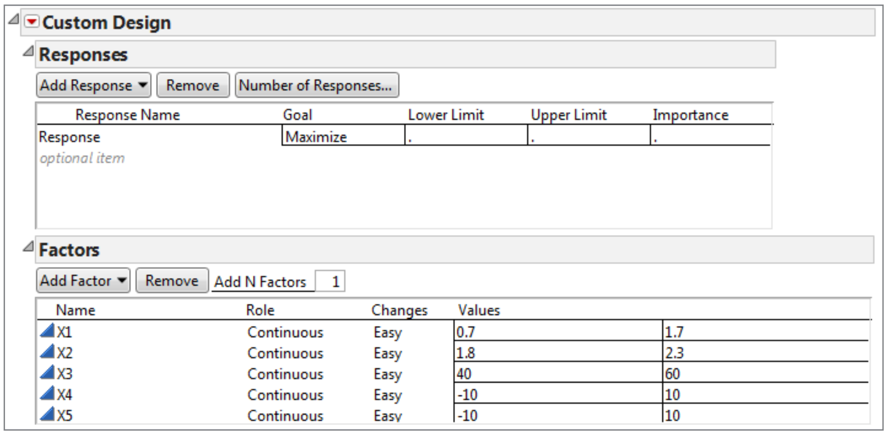
Now that this time-to-frequency conversion program is created in JMP, we can add it to the set of available tools during analysis any time that this functionality is needed. We have effectively expanded the scope of analysis that JMP can perform by utilizing this MATLAB function.

Case II: DOE for Computer Simulation Model Optimization

The Custom Designer in JMP is an easy way to set up an experimental design to fit the specific problem you are facing. It can be customized to the unique set of factors, constraints and questions you are trying to address through experimentation.

Adopting a systematic design for optimizing the output of a computer simulation model or matching the response of a model to collected behavioral data are two areas where using design of experiments (DOE) in JMP makes algorithm development in MATLAB more efficient. We can use JMP to set up experimental conditions, perform simulation runs and analyze or explore a model's opportunity space interactively with the JMP Profiler.

In the next example, we will consider exploring a simulation model built in MATLAB, which includes five input parameters. Our goal is to maximize the output of the model by tuning each of these five parameters.



Response Name	Goal	Lower Limit	Upper Limit	Importance
Response	Maximize	.	.	.
optional item				

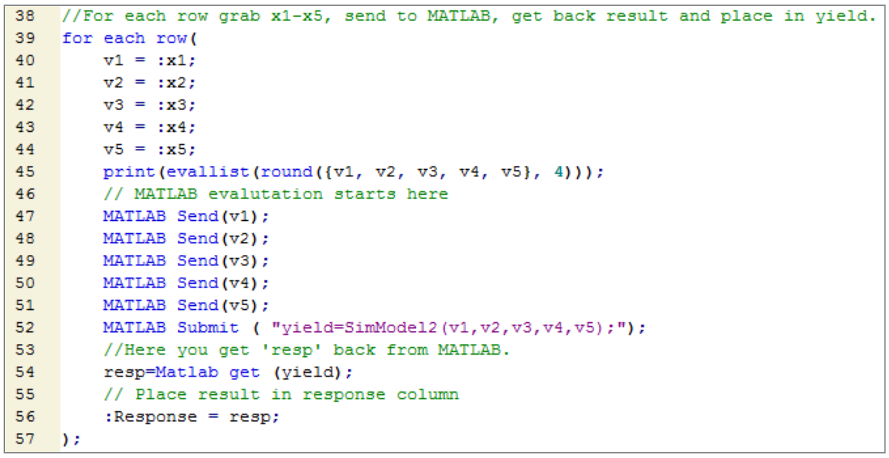
Name	Role	Changes	Values	
X1	Continuous	Easy	0.7	1.7
X2	Continuous	Easy	1.8	2.3
X3	Continuous	Easy	40	60
X4	Continuous	Easy	-10	10
X5	Continuous	Easy	-10	10

Figure 11. Custom design for a MATLAB simulation model.

Figure 11 shows the setup of the response and the five factors we are interested in exploring. Setting up this experiment in JMP using a Custom Design will produce an optimal design of run conditions for our model, as well as the Fit Model script to analyze the results once they are returned from our MATLAB model.

The JSL to run our experiment is shown in Figure 12. We will use the `for each row ();` function to run through each of the rows in the JMP data table, send factor values to MATLAB and then execute our model with this factor setting. As in the previous example, we can use the `MATLAB Send ();` function to send each of the factor settings v1-v5 from the JMP variable space to the MATLAB variable space, and then execute our MATLAB model. This returns the yield output given our set of input variables. To return the yield value to the row corresponding to our trial run, we can use the `MATLAB Get ();` function to bring the yield variable into the JMP variable space. Finally, we can use the `:Response = resp;` code to write the resulting yield into the response column in the row that corresponds to the condition we have just run.

This code is flexible; if we decide to change the number of experimental conditions, add factor constraints or adjust our factor settings, we can just rerun the code and collect new data from our MATLAB model.



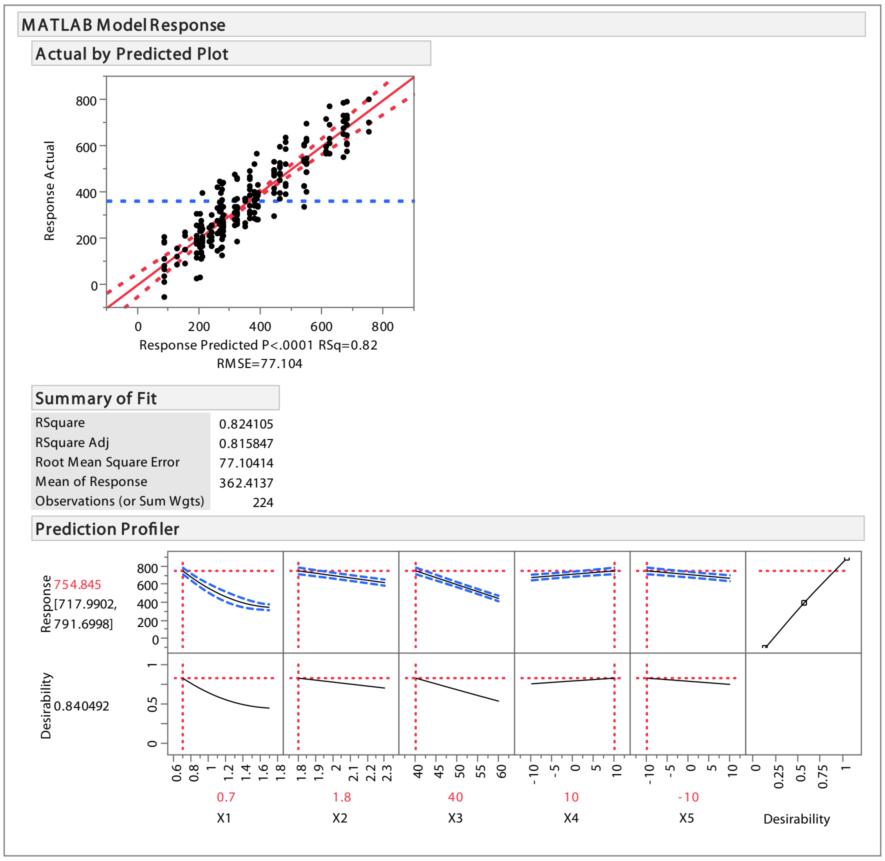
```
//For each row grab x1-x5, send to MATLAB, get back result and place in yield.
for each row(
    v1 = :x1;
    v2 = :x2;
    v3 = :x3;
    v4 = :x4;
    v5 = :x5;
    print(evallist(round({v1, v2, v3, v4, v5}, 4)));
    // MATLAB evalutation starts here
    MATLAB Send(v1);
    MATLAB Send(v2);
    MATLAB Send(v3);
    MATLAB Send(v4);
    MATLAB Send(v5);
    MATLAB Submit ( "yield=SimModel2(v1,v2,v3,v4,v5);");
    //Here you get 'resp' back from MATLAB.
    resp=Matlab get (yield);
    // Place result in response column
    :Response = resp;
);
```

Figure 12. JSL code to run each row in the JMP data table as a condition of the simulation model and return results back to the yield column. This code is robust to new experimental designs and number of conditions.

The resulting model can be fit using the Fit Model script generated by the experimental design. This is shown in Figure 13. While multiple regression can be performed using MATLAB, the interactivity in JMP afforded specifically by the JMP Prediction Profiler is an attractive option to MATLAB users, both for exploring the opportunity space of the model as well as for communicating results to others. With the Profiler, you can drag a factor, change its setting and immediately see the impact on the response and other factors. By using the Maximize Desirability command in the Profiler, you can quickly identify the factor settings, which amount to the greatest yield response.

Another useful tool in Fit Model is the “assess variable importance” option, which is also accessed via the Prediction Profiler. This is shown in Figure 14. The “assess variable importance” method can be used to see the impact each factor is having on a response. It is also a way to compare factor effects across multiple modeling techniques. This technique can also be used to compare new versions of MATLAB simulation models using a common method.

By using JMP to design experiments, run MATLAB simulation models, return results and analyze them in JMP, you speed up the learning cycle in designing your MATLAB models and optimize resources – especially when MATLAB model simulation run times are long. Implementing a systematic way to run an experiment using an optimal design allows you to extract the most information from a simulation model for the same amount of experimental runs.



MATLAB Model Response

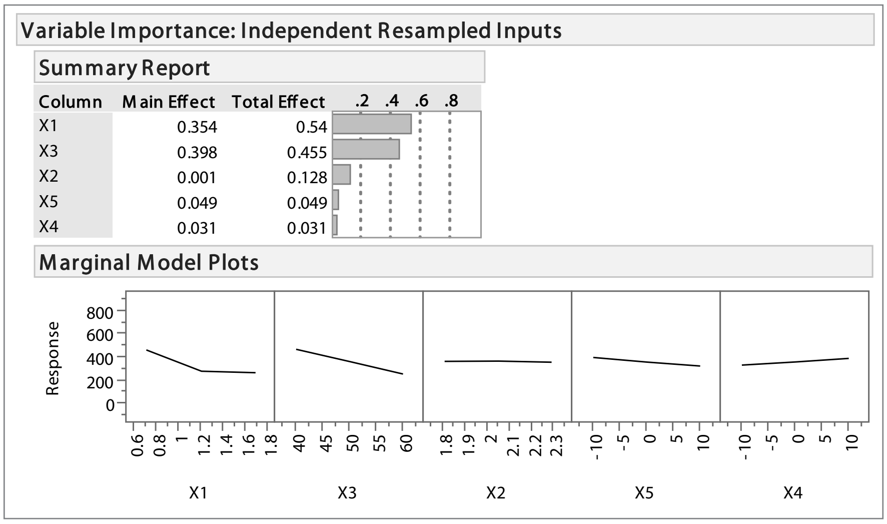
Actual by Predicted Plot

Summary of Fit

RSquare	0.824105
RSquare Adj	0.815847
Root Mean Square Error	77.10414
Mean of Response	362.4137
Observations (or Sum Wgts)	224

Prediction Profiler

Figure 13. Results of Fit Model from running the simulation model in MATLAB. We can then use the interactive Profiler to explore the model, optimize variables and maximize our yield response.


Variable Importance: Independent Resampled Inputs

Summary Report

Column	Main Effect	Total Effect
X1	0.354	0.54
X3	0.398	0.455
X2	0.001	0.128
X5	0.049	0.049
X4	0.031	0.031

Marginal Model Plots

Figure 14. Assessing variable importance from the Profiler. This method lets us compare main and total effects of each of our factors on the response. It can also serve as an omnibus method for comparing the impact of variables across iterations of our MATLAB simulation model, giving a clear view of the input variables that most affect the response.

Case III: Genetic Algorithm Optimization Application

A final example of the JMP interface to MATLAB is to utilize capabilities that are not included in JMP but are found in MATLAB. An example of this is a multivariate optimization employing a genetic algorithm (GA).

Suppose a model is developed in MATLAB that calculates the impact of implementing different sets of technologies on an aircraft to: 1) decrease fuel consumption, 2) reduce emissions and pollutants (i.e., NOx), and 3) reduce noise during takeoff and landing. These three objectives are further joined by a desire to keep the risk low (by adding as few technologies as possible to the aircraft design) while minimizing the cost. Of course, these objectives may conflict. The trade-offs between the objectives can result in different optimal technology sets based upon the decision maker's objective function. Since this objective function is unknown *a priori*, a Pareto frontier, or the set of nondominated points, is sought to explore and quantify the designs that would be optimal for different objective functions.

With more than 90 technologies within the entire technology portfolio from which to choose, exhaustively testing all 2^{90} combinations is infeasible. Therefore, a genetic algorithm is employed in MATLAB to assist in discovering this Pareto frontier for further decision analysis. Given that an optimizer implementing a genetic algorithm may require significant time to execute multiple generations with large populations of candidate designs, a user may want to perform intermediate analysis in between each generation, stop the execution if errors are found, or exit early if a sufficient Pareto frontier is defined. Furthermore, checking the behavior of the optimizer without running a full set of generations may be more useful for validation and verification purposes. Lastly, since a user or the end decision maker will want to explore the Pareto frontier dynamically after the model has been executed, using JMP to perform the post-processing and data analyses can facilitate the decision making.

The MATLAB model creates various output graphs shown in Figure 15. The associated data stored in variables or matrices can be exported to tables or other formats (e.g., CSV). However, since a user may want to interact with and perform additional analyses on these data to explore the technology sets linked to each design point, a simple interface can be developed within JMP, which ties directly to the MATLAB output data. Data selection, filtering and other interactive functionality within JMP easily facilitate the analysis required for good decision making.

The MATLAB model contains a main executable file with a number of initialization commands and required functions for the GA to operate correctly. This main file will make use of all other .m files containing function definitions as well as native functions specific only to MATLAB. Some of the user-defined variables include setting: 1) the technology readiness level required by feasible technology sets, 2) the maximum acceptable cost, and 3) the population size. Similarly, additional GA-specific parameters such as the crossover rate, mutation rate and maximum number of generations can be defined.

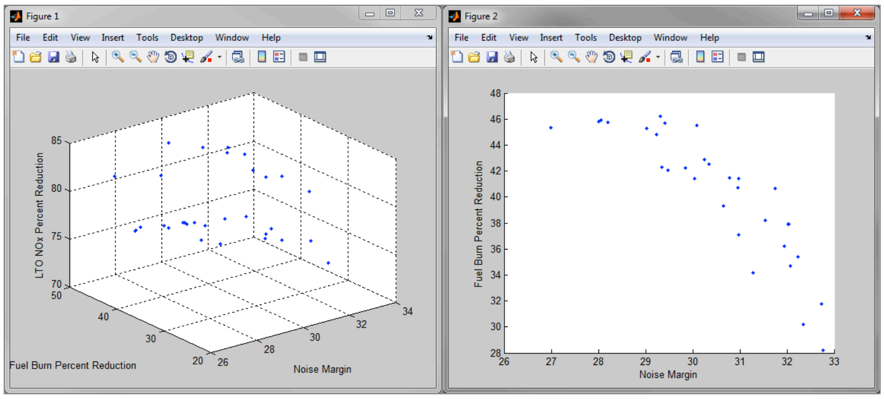


Figure 1
File Edit View Insert Tools Desktop Window Help
LTO NOx Percent Reduction
85
80
75
70
50
40
30
20
Fuel Burn Percent Reduction
26
28
30
32
34
Noise Margin
Figure 2
File Edit View Insert Tools Desktop Window Help
Fuel Burn Percent Reduction
48
46
44
42
40
38
36
34
32
30
28
26
27
28
29
30
31
32
33
Noise Margin

Figure 15. MATLAB output graphs from GA optimizer after one generation. Left: 3-D graph of Pareto frontier. Right: 2-D view of the same data.

To define these parameters from a JMP interface, a user can obtain access to various inputs (e.g., a number edit box), which are sent to MATLAB to update those specific variables. Other essential inputs to the model, such as the compatibility matrix (the matrix defining whether two technologies are mutually exclusive), could be likewise defined by the user in more complex interfaces, or by selecting the appropriate data file, but a default matrix is set from the MATLAB code directly. Similarly, an impact matrix, which defines each technology's settings for various aircraft design parameters used to calculate the noise, fuel burn and emissions of the specified design, could likewise be updated or entered from the JMP interface.

Once the MATLAB connection is made from JMP, the JMP interface initializes the first generation of data points (i.e., technology combinations) and evaluates the performance of each combination. Then the user is permitted to execute additional generations of the GA optimizers by sending the command through the connection to improve the set of combinations and approach the Pareto frontier of nondominated points. After each iteration (or generation), the JMP interface will display the data on a 3-D graph representing the output metric values for each of the three objectives for each technology set. Each generation is displayed in the interface with the capability to explore the full generation history as shown in Figure 16. With the data inside a JMP data table, data points can be selected, hidden, filtered or further analyzed in the numerous statistical and graphical platforms within JMP.

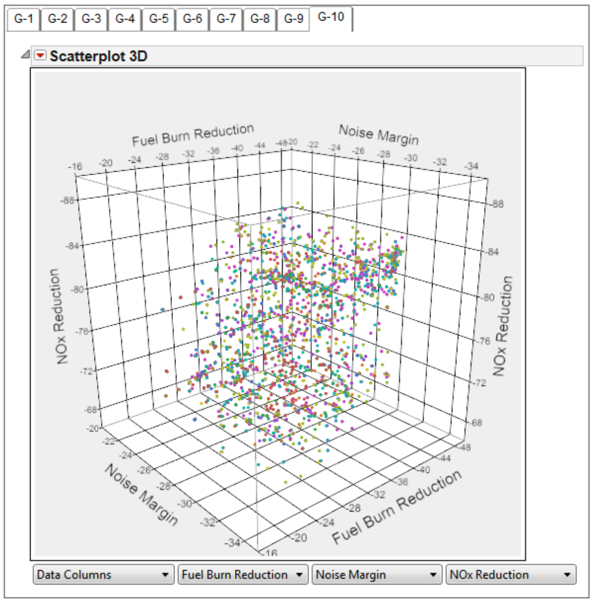


Figure 16. 3-D graph of the 10th generation of the GA optimizer for the three objectives fuel burn, noise and NOx.

One example of such a process is available within the interface directly. By clicking on the “Animate” button, a 3-D graph and scatterplot matrix are created with a dynamic filter prepared to cycle through and animate the relative exploratory nature of each generation in the GA optimizer. See Figure 17.

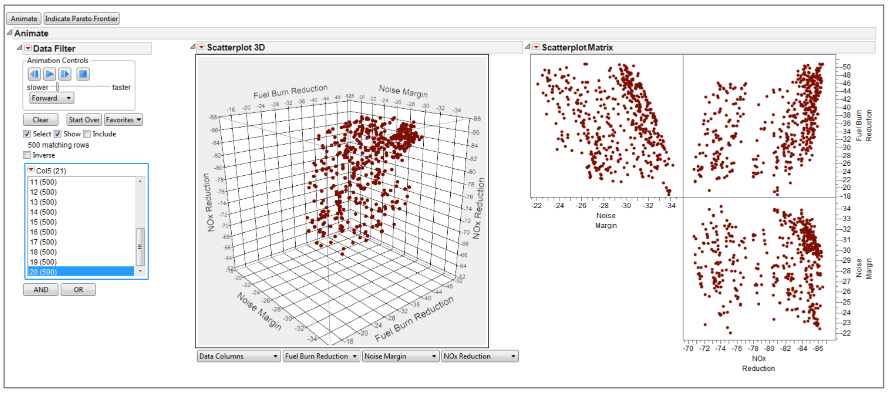


Figure 17. Implementing a filter with animation to explore the generation history of the GA optimizer.

After 15 generations, many members in the population approach the Pareto frontier. This is clearly visible in the Noise vs. Fuel Burn subplot in the scatterplot matrix of Figure 17. The edge of the nondominated points (top right corner of this subplot) reveals a trade between noise and fuel burn. Along this edge or Pareto frontier, seeking to improve one objective may require a decrease in the other. Other trades exist in the other subplots where the axes have been set such that the ideal solution is in the top right corner of each subplot (i.e., larger negative values are better).

Furthermore, an analyst familiar with the interactivity inherent in JMP can use the software’s native functionality to isolate and identify the Pareto frontier in the last generation or in each of the generations. One way to accomplish this would be to use basic data table manipulation techniques, selecting the dominant rows from the Row Selection menu and unchecking the appropriate check boxes associated with each of the three objectives to indicate a desire for low values. A button to perform this process automatically also has been added to the interface beside the “Animate” button, the results of which are applied to the data and shown in Figure 18. Although various MATLAB functions exist to perform a similar action, a user may not be willing or qualified to seek out the necessary functions. Thus, bringing the data into JMP can make it easier to perform the desirable analysis since it doesn’t require the user to search for and learn new functions in MATLAB.

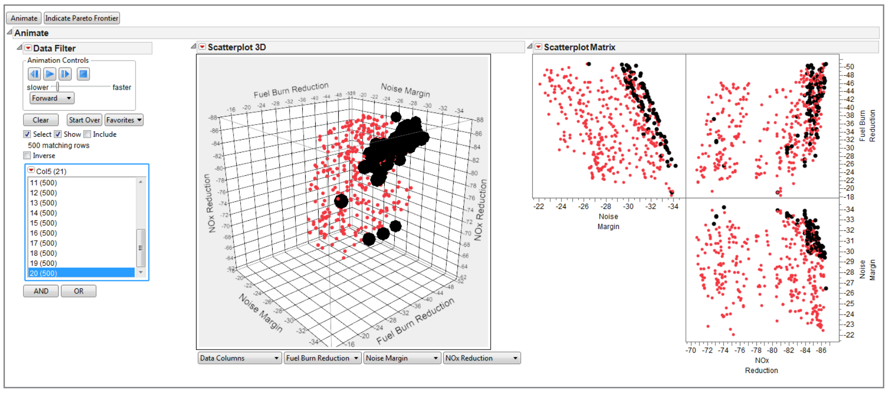


Figure 18. Application of native JMP functionality for selecting dominant points.

An animation with the Pareto frontier identified will cycle through the generations, showing the movement of the frontier to more optimal values. Similarly, these plots can be created to compare generations side by side, as shown in Figure 19, to study the intermediate Pareto frontier at each generation.

Finally, the interactivity of GUI elements in JMP enables dynamic exploration of the actual set of technologies associated with each design point. These data can be joined with the performance metrics and costs associated with any particular technology portfolio for further comparisons, analysis and, ultimately, portfolio decisions.

A number of methods, described below, can be implemented to facilitate this process of exploring the technology portfolios that compose the Pareto frontier of the last generation executed.

Figure 20 plots the data from the last generation with the number of technologies in each portfolio and the associated cost. In general, and as expected, more technologies in a portfolio result in a higher cost. However, with the JMP functionality to select points on the Pareto frontier of the three-objective space, the subdivision between dominated and nondominated points can be quickly applied and separated in Graph Builder, as shown on the right-hand graph of Figure 20.

The analysis suggests to reach the Pareto frontier, a relative cost just greater than 43 is required. Similarly, the minimum number of technologies at that same point to reach the Pareto frontier is 18.

However, this is only half the trade-off between cost, risk and performance. By adding the output metrics of the three objectives to the graph, the trade-offs between cost and performance are more readily visible, as shown in Figure 21.

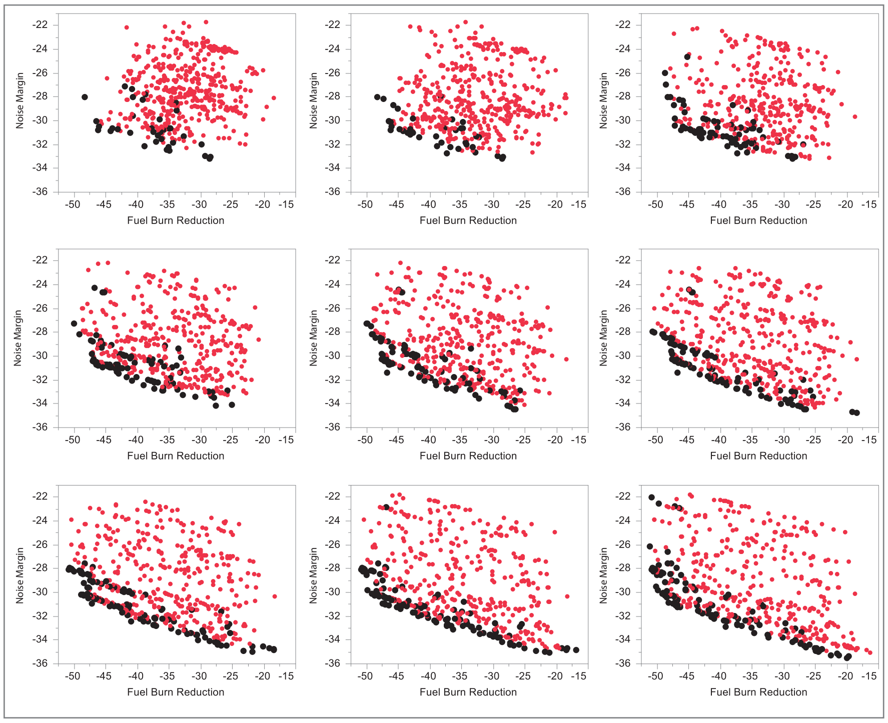


Figure 19. Comparing generations 0 (top left) through generation 16 (bottom right) of the GA optimizer (skipping every other generation), for the objectives of minimizing noise and fuel burn.

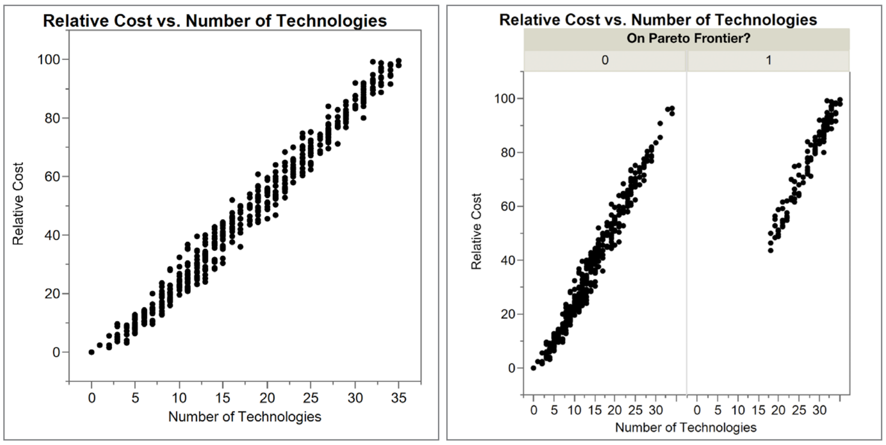


Figure 20. Number of Technologies vs. Relative Cost without (left) and with (right) categorizing data based on Pareto frontier status.

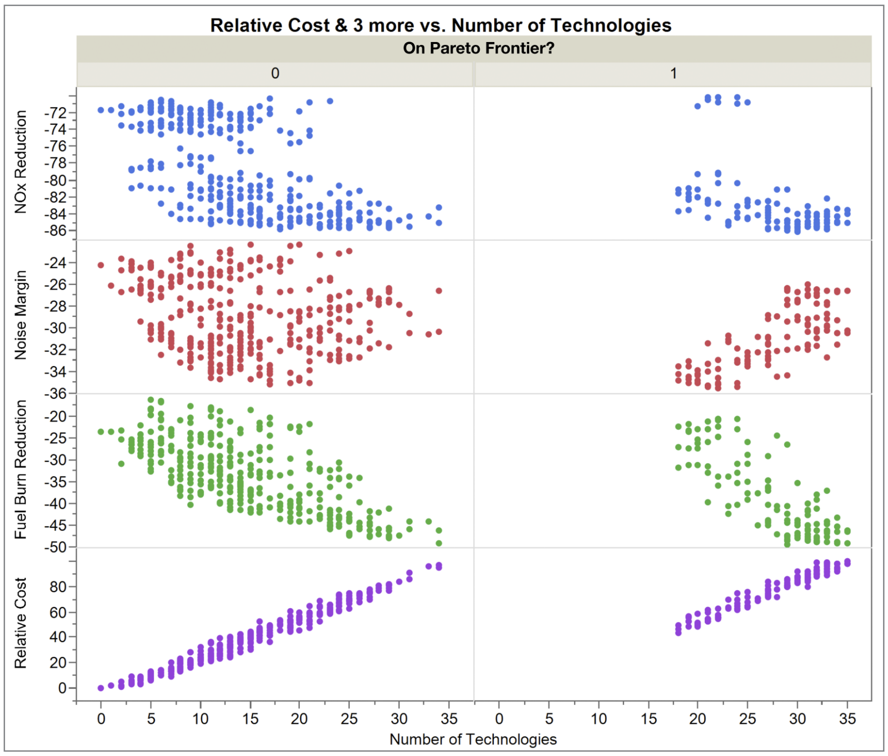


Figure 21. Number of Technologies vs. Relative Cost and performance objectives categorized by Pareto frontier status.

In this figure, the least expensive technology portfolio with a relative cost of 43 and with 18 technologies offers, comparatively, a lot of NOx and noise reduction but only 30 percent fuel burn reduction. Figure 21 suggests that with more technologies, fuel burn reduction could be improved up to 50 percent, but at the expense of noise, and, of course, at a higher relative cost with more risk (due to more technologies). This is evident by the general trend lines of the data added to the same graph and shown in Figure 22. Linear fits of the various relationships are shown with other fit metrics and could be used in turn for additional optimization processes.

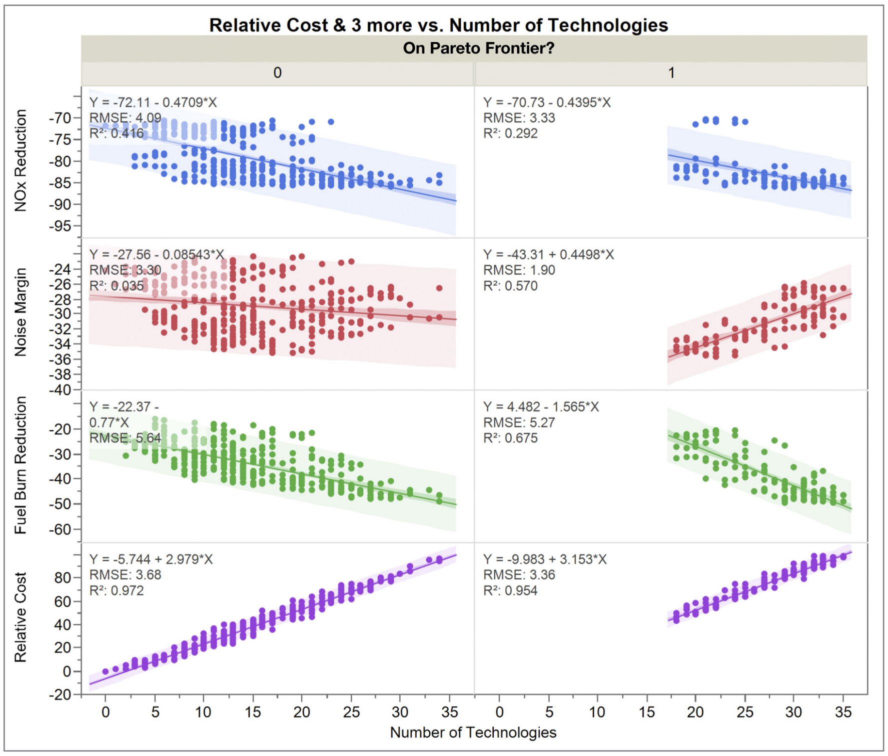


Figure 22. Number of Technologies vs. Relative Cost and performance objectives categorizing by Pareto frontier status with additional fit metrics.

Pursuing more than just a few technologies might still be too risky or too expensive, and thus it may be desirable to investigate which technologies occur most often on the Pareto frontier. Figure 23 shows the technologies sorted with respect to the frequency of occurring in the portfolios in the Pareto frontier. One potential decision mechanism could then be to include the most common technologies (e.g., the first five) and compare the performance of that portfolio to the full set. It would almost definitely not be on the Pareto frontier (which required at least 18 in this example) but may be more feasible in terms of affordability.

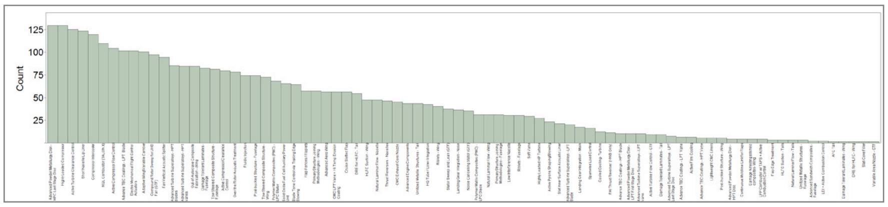


Figure 23. Occurrence for each of the technologies across all portfolios on the Pareto frontier.

Summary

This paper presented the JMP interface to MATLAB, introduced in JMP 11. The basic execution model, available functions and a simple example showing how to use the interface was shown. Three cases where using JMP with MATLAB functionality were then introduced. The first showed a custom application built in JMP to use MATLAB functionality, which extends the analytic options available for processing data in JMP – in this case, implementing an FFT algorithm to convert data in the time domain to the frequency domain. Next, a case where JMP is used to design an optimal experiment and run conditions of a simulation model built in MATLAB was shown. Results were returned to JMP for analysis, factor profiling and assessing variable importance. And finally, a custom application built in JMP to run a genetic algorithm in MATLAB to optimize a multidimensional object space interactively was presented. We've just scratched the surface of what is possible with the connection between JMP and MATLAB. Because of the flexibility and interactivity of JMP, as well as the ability to explore data from MATLAB in an ad hoc nature, it is possible to use JMP as a hub to reach out to pre-existing or new MATLAB functions, return data to JMP for analysis, visualization and exploration, and then communicate results and findings to others.

Acknowledgements

Thanks are due to Jeff Polzin of JMP for developing and implementing the MATLAB interface in JMP 11.

Contact Information

Comments and questions: Please contact Daniel Valente.
Email: Daniel.Valente@jmp.com
Phone: +1 919-531-1655

For more information on analytical application development with JMP visit:
jmp.com/applications/analytical_apps

About SAS and JMP

JMP is a software solution from SAS that was first launched in 1989. John Sall, SAS co-founder and Executive Vice President, is the chief architect of JMP. SAS is the leader in business analytics software and services, and the largest independent vendor in the business intelligence market. Through innovative solutions, SAS helps customers at more than 65,000 sites improve performance and deliver value by making better decisions faster. Since 1976 SAS has been giving customers around the world THE POWER TO KNOW®.

SAS Institute Inc. World Headquarters +1 919 677 8000

JMP is a software solution from SAS. To learn more about SAS, visit **sas.com**

For JMP sales in the US and Canada, call 877 594 6567 or go to **jmp.com**