Automated Report Creation: From Data Import to Publication

JMP Discovery Conference – Tucson

Brian Corcoran – Software Development Director, SAS

JMP provides a variety of methods to get data into the product. It has a lot of facilities to export finished reports for viewing. However, it is often difficult to envision the end-to-end workflow. A question that often comes up is: How do I take my complete report workflow and automate it so I can come to work in the morning and already have my finished report waiting for me? This paper will give an example of how to accomplish this. We will import data from a public data source, which will give a brief survey of one of JMP's import capabilities. Next, we will perform some exploration in JMP to produce the report that conveys our findings. We can then export that output. While we do that work, we will accumulate the JSL scripts that we need to recreate the output. Finally, we will automate the tasks to produce our report daily.

Imagine that we have air quality data that gets periodic updates. The data comes from a variety of countries and states and is aggregated on one site. Any of the cities involved can update the data on a given day, so the data can change frequently. Because of this, we would like to produce a graphic each morning with the updated results.

The air quality data comes from a website and organization called **openaq**[1]. The site provides an API available via HTTP access. A REST interface is provided for retrieving the data. JMP 14 introduced HTTP Request, a mechanism to work with interfaces just like this. JSL is required to use this, so it is definitely a more advanced topic. There are more and more websites that are offering data in this format, and these sites are often official government sites. The World Bank data that I have used in previous talks is getting migrated to a databank accessible through REST interfaces. The goal with this paper is just to make you aware of such sites, and the increasing use of them to provide data to the public.

To make a request to a site like openaq, the basic requirements are a URL to the API that provides the data, and an associative array of parameters to pass to the site to tell it what kind of data you want to retrieve. An associative array in our case is just an array of name value pairs, like "Tucson":"Ozone". JMP will take the parameter arrays and transform it into JSON, a syntax specification for name/value pairs among other things. It will then pass this request via the web protocol HTTP to the site, which will usually return a block of data also in JSON format. JSL can then be used to break apart this data and put it into a data table. The key JSL snippet for this example is the following:

```
query = [=>];

query["country"] = code;
query["limit"] = 200;
request = New HTTP Request (
```

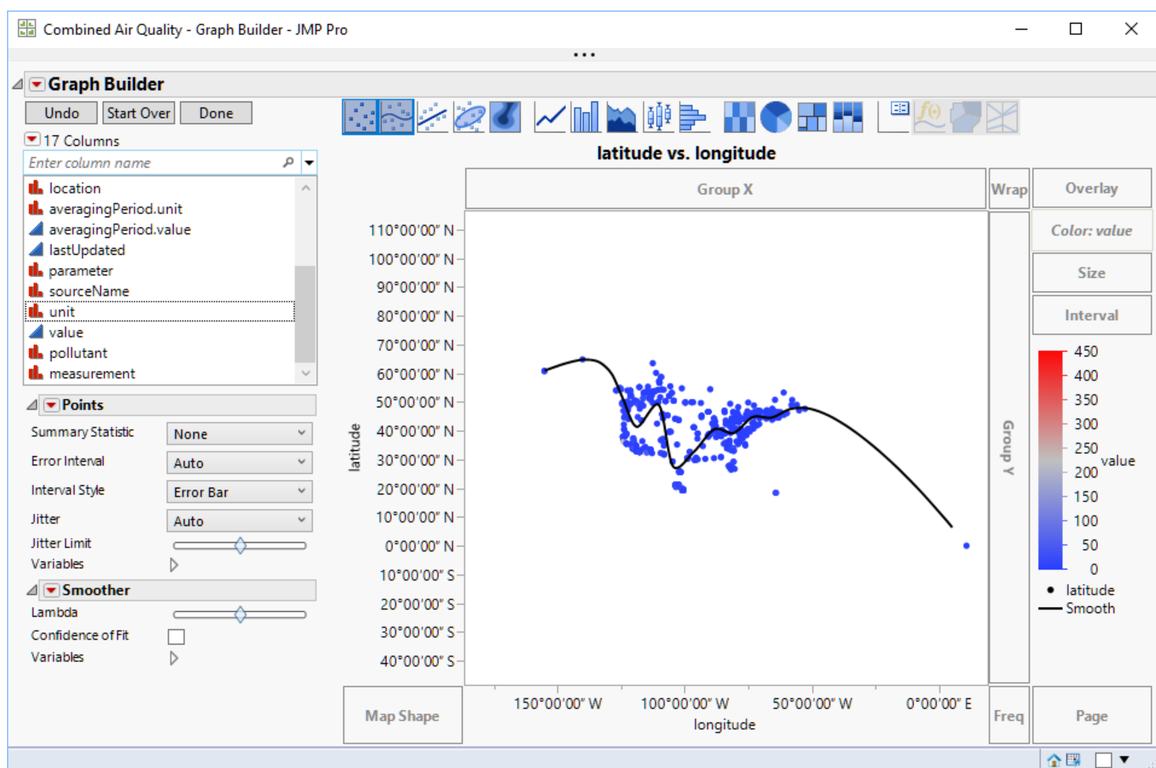
[1] openaq. *Air Quality Data*, openaq, n.d. Web. August 22, 2019. https://openaq.org . Obtained under Create Commons CC BY 4.0 license. Web API data obtained from https://api.openaq.org.

```
        URL("https://api.openaq.org/v1/latest"),
        Method("Get"),
        Query String(query)
);
json = request << Send();
```

Here we create an HTTP request object in JSL, and then ask it to transmit our query string to the web API call. The returned information is stored in the **json** variable. This is then split apart and used to populate the data table. I've included the script to openaq, developed by my co-worker Bryan Boone, in the conference materials. Don't get overwhelmed with the length or complexity. One key takeaway from this tutorial is that you can assemble data from lots of places and repeat it using standard techniques. I'm going to copy the openaq script into a **WorkingScript.jsl** file that we will use to collect all of our work. This will form the basis for the automated task that we develop.

If we run the HTTP Request script, it will take a few seconds to fetch the data, and we can see the data table populate. We will end up with several hundred rows of air quality observations. Unfortunately, the pollutant measurements are all mixed up. There is numeric value for each pollutant, and a column that specifies the type of pollutant. It would be nice to have all of these separated out or sorted.

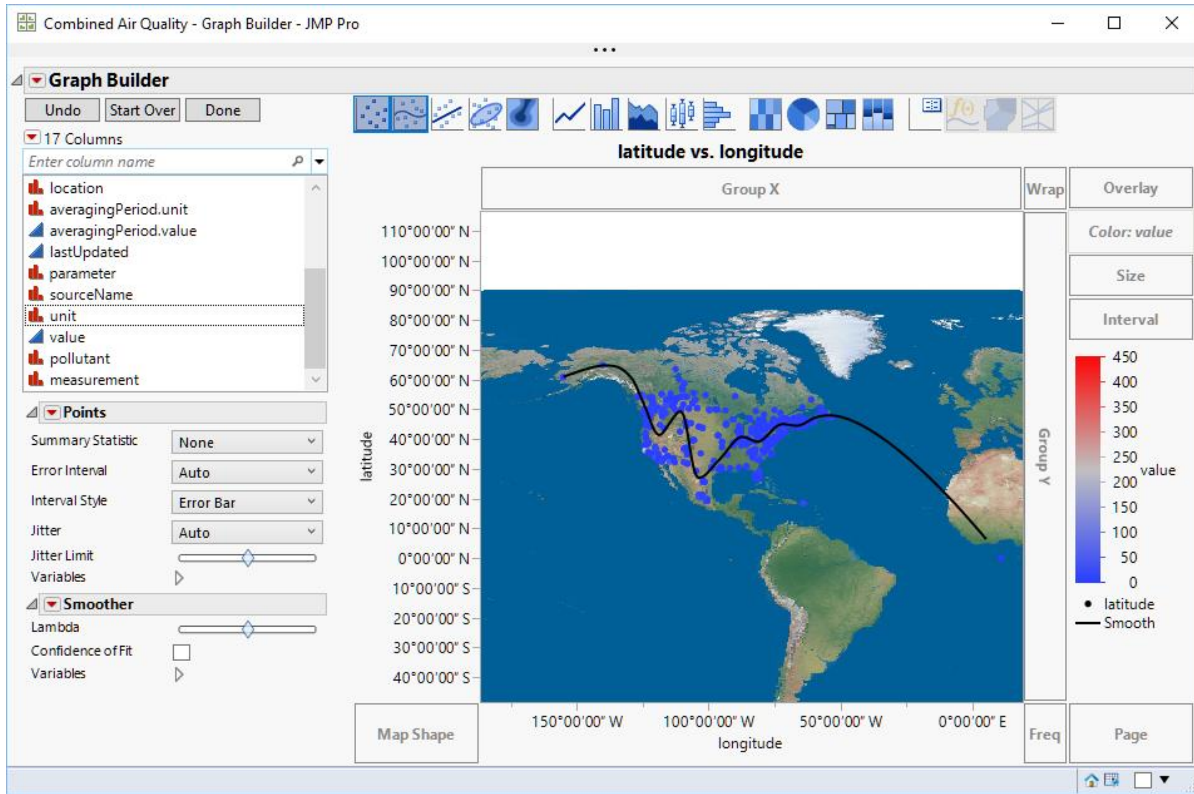
The data contains latitude and longitude data. If we drag these into Graph Builder, it will give us an idea of the quality of the data.



Initial Look at Air Quality Measurements

This seems like a strange pattern. Do we have some outliers or incorrectly entered values? The easiest way to get more insight is to put a map behind the measurements. If we right click in the graph

frame and select Graph->Background Map and then select "Simple Earth" under **Images**, we see the following:

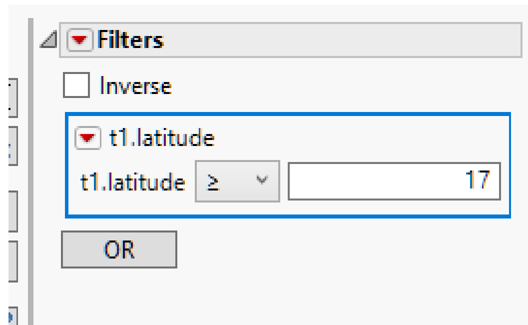


Data with Map

Incorrectly mapped points often show up in clusters by the poles or equator. In fact, it turns out the issue is that a measuring station in Saskatchewan is returning 0,0 for longitude and latitude. It would be nice to exclude this point. It's time for Query Builder for JMP files. Before we do this, we need to put a JSL snippet into our scripting to save the data to disk. This would look like:

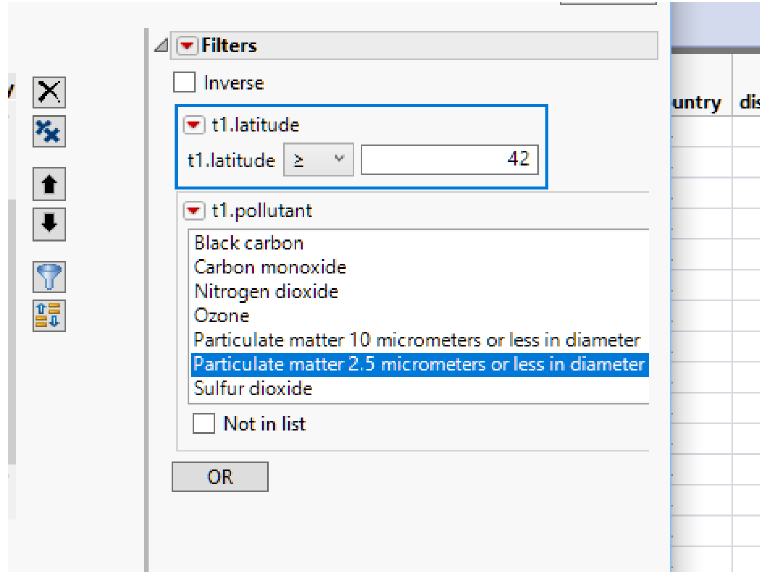
```
dt << Save("c:\Discovery Demo\Tucson\Combined Air Quality.jmp");
```

Now we can use Tables->Query Builder on the result table from our HTTP Request script. The first thing to do is to apply a filter to the latitude. Dragging *"t1.latitude"* over the filters area and providing a value of 17 reduces our observations to Puerto Rico and areas north of Puerto Rico.



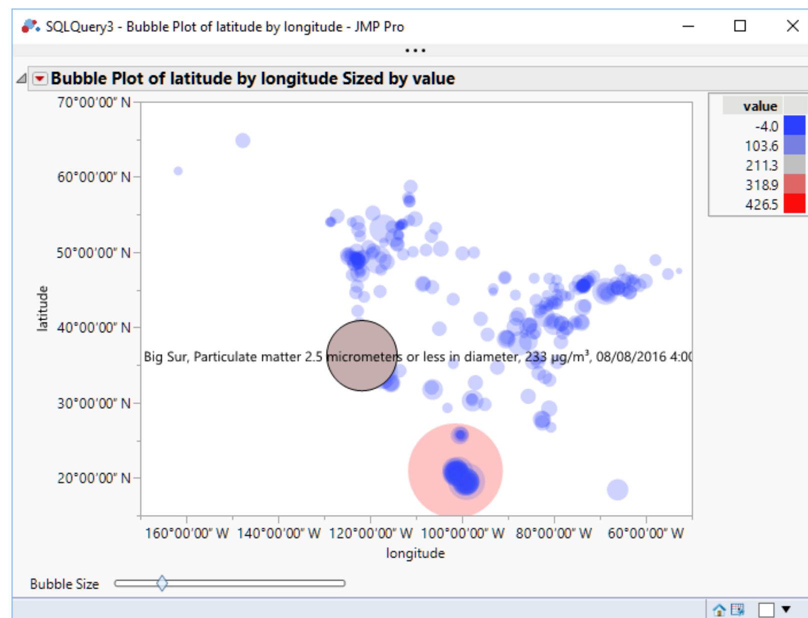
Reduce Observations to Puerto Rico and North

Now we can examine regional data for particular pollutants by dropping in a filter for "*t1.pollutant*". We can then generate a few query results after selecting individual pollutants to examine. We could make this a prompted filter to allow user input as to the choice of pollutant to examine, but this would not fit well with our desire to automate our final results. Let us try *"Particulate matter less than 2.5 micrometers in diameter".*

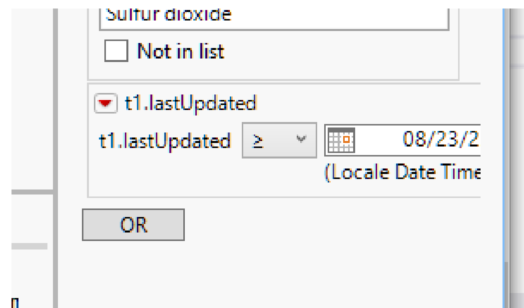


Filtering on Pollutant

Now if we run this query, we get just the values for *"Particulate matter less than 2.5 micrometers in diameter".* A bubble plot would be a good visualization of this data. We can use *latitude* as Y, *longitude* as X, and *value* for Size and Coloring. Before we even put a map on this, we see two large values.



I'm not sure about the reading in Guanajuato, but I recognize the Big Sur reading as a time when there was a fire in the area. The thing is, we don't really want to be viewing data from 2016. I'm going to go back to the query and put in a filter for date, selecting Today in the date picker.



Add the Date

It is worth looking at the query now, which we can get from the Source script in our resulting data table.

```
New SQL Query(
	Version( 130 ),
	Connection( "JMP" ),
	JMP Tables(
		["Combined Air Quality" =>
		"\c:\Discovery Demo\Tucson\Combined Air Quality.jmp"]
	),

	<MUCH OMITTED HERE>

	From( Table( "Combined Air Quality", Alias( "t1" ) ) ),
	Where(
		GE(
			Column(
				"latitude",
				"t1",
				Numeric Format( "Latitude DMS", "0", "NO", "" )
			),
			17,
			UI( Comparison( Base( "Continuous" ) ) )
		) & In List(
			Column( "pollutant", "t1" ),
			{"Particulate matter 2.5 micrometers or less in diameter"},
			UI( SelectListFilter( ListBox, Base( "Categorical" ) ) )
		) & GE(
			Column(
				"lastUpdated",
				"t1",
				Numeric Format( "Locale Date Time h:m:s", "0", "NO", "" )
			),
➔			3649401299,
			UI( Comparison( Base( "Continuous" ) ) )
		)
	)
➔) << Run
```

Two things are worth noting here. Even though we said "Today" in the date picker, the script shows a numeric value for today's date. We don't want to be frozen in time for that date, we want only

the current day's data. We can change this value manually to **Today()**, subtracting one day to make sure we get everything in the last 24 hours.

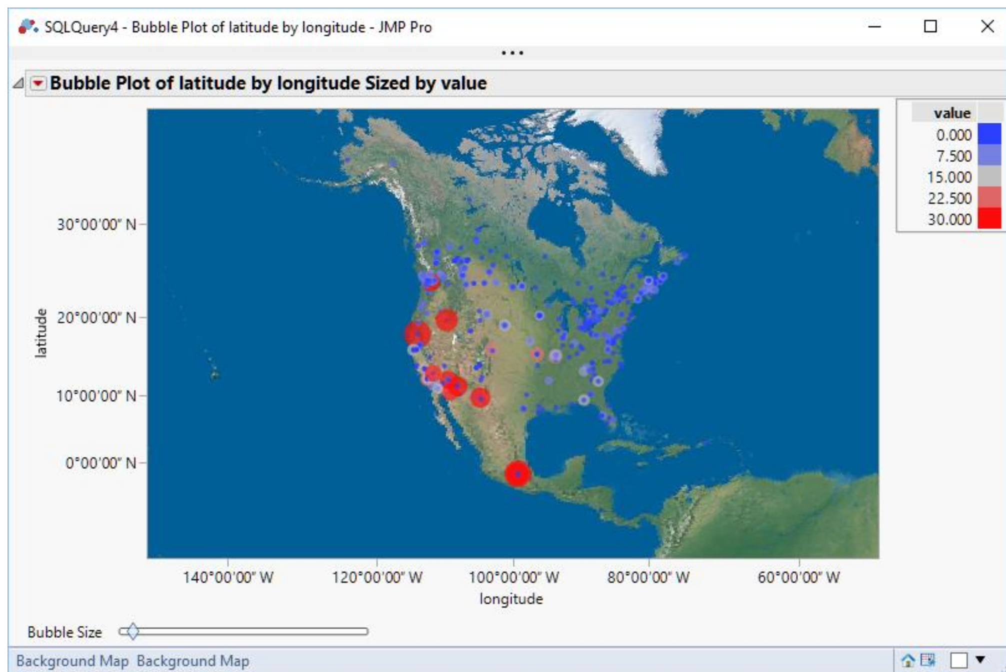
The last part in particular is also important. The **Run** command at the end will run the query either on a background thread, or in the foreground, depending on your preference settings. However, we are very order dependent with our scripting for this example. To force the query to run in the foreground, we can use the **Run Foreground** command instead. Now that last snipped looks like:

```
                    Date Increment( Today(), "Day", -1, "actual" ),
                    UI( Comparison( Base( "Continuous" ) ) )
            )
    )
    ) << Run Foreground()
```

We can now copy that script into our WorkingScript.jsl file, adding a semicolon to the end of the query. We also want to assign the result of the query to a data table variable, like:

```
qdt1 = New SQL Query(
```

Now we can remake our Bubble Plot with the current data and filters applied.

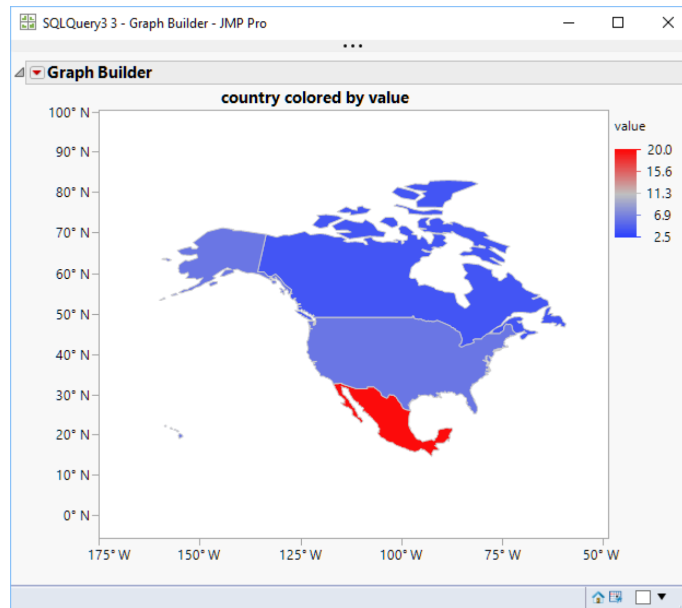


Final Bubble Plot

Our Bubble Plot script is relying on using the current data table. If we run several queries, it is possible for this reference to get confused and to get the wrong output. To avoid this, we can assign the output of the queries to data table objects, as that is the return object for Query Builder. Now, when we go to generate our reports we need to make sure that the correct data table is current. We do this by passing the data table reference to **current data table()**, like this:

```
current data table(qdt1);
```

It would be nice to have country-wide summary to show as well. We can use Graph Builder to do this, just putting *Value* in the Color role and *Country* in the Map Shape role.

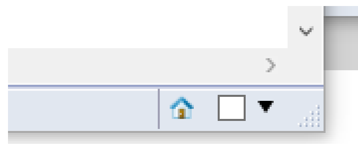


Average Values in Graph Builder

We can use "Save Script to Script Window" to save out the JSL we need to recreate this graph, and then we can transfer it to our WorkingScript.jsl file.

```
gbOutput = Graph Builder(
	Size( 534, 456 ),
	Show Control Panel( 0 ),
	Variables( Color( :value ), Shape( :country ) ),
	Elements( Map Shapes( Legend( 5 ) ) )
);
```

Now, it would be nice to combine these two graphs into a small dashboard. One way to do this is to use the UI Combine Windows feature. You can find a checkbox on the bottom right of the graph windows:



Combine Windows Checkbox

You can check the boxes for the two graph windows and then use the dropdown to select "Combine Windows…". This will bring up a UI for customization, but if you select the defaults you will end up with a dashboard containing the graphs.

Now you can use the dashboard red triangle menu to do "Save Script->To Script Window". The problem with this is that Application Builder, the JMP platform that creates the dashboard, is very thorough about making the script. It essentially duplicates much of the work that we have already done,

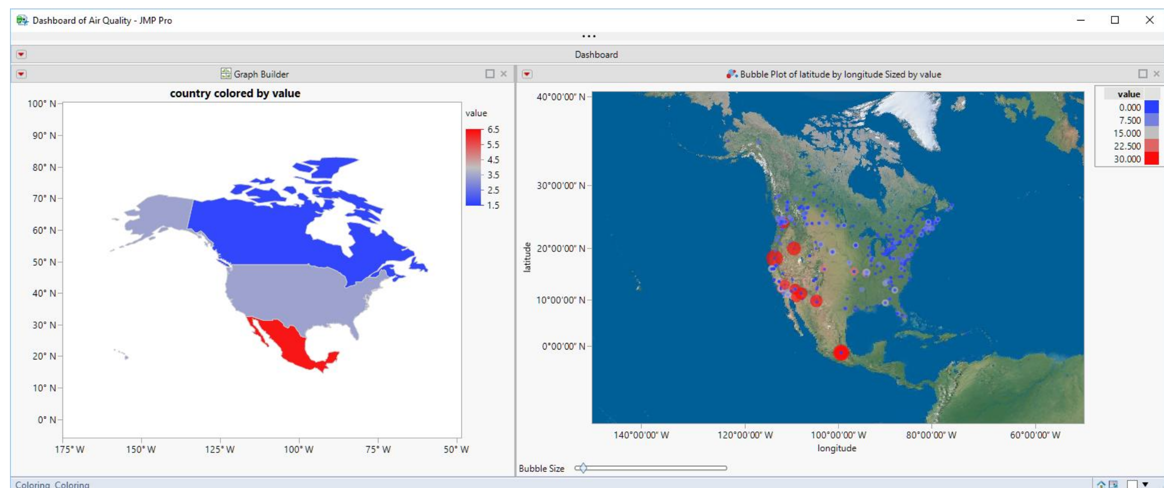
including references to the query output. This is overkill for this particular tutorial, since we already have our working script with everything we want in the order that we want it.

So, we can include a simple JSL snippet to do the Combine Windows. It looks like this:

```
app = JMP App();
app << Set Name( "Dashboard of Air Quality" );
app << Combine Windows( {gbOutput << Report, bubbleOutput << Report} );
(app << Get Modules)[1] << Set Window Title( "Dashboard of Air Quality" );
app << Run;
```

The basis of this script came out of the JSL Scripting Index under the Help menu, so remember that as a resource for tying together your work. All that is happening here is that we are getting something called the Report object out of each graph. It is this object that Combine Windows understands how to tie together. We then set the Window title, which is not absolutely necessary, and issue the **Run** to do the window combination.

This is the result:



Dashboard with Both Outputs

Typically to output graphs to other formats, you get the Report object from the analysis and pass it to the Save function that you require. The JSL Scripting Guide would give an example like this:

```
biv = bivariate( y( :weight ), x( :height ) );
rbiv = biv << report;
rbiv << Save PDF( "path/to/example.pdf" );
```

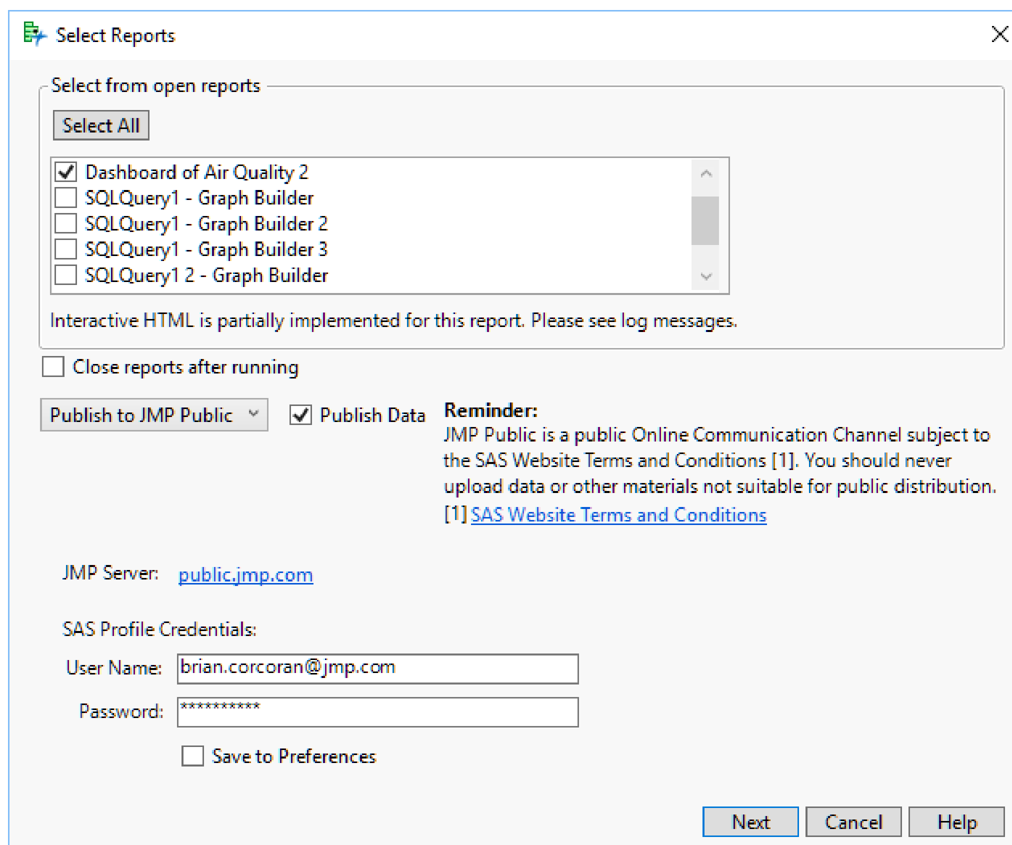
However, we have a JMP App object and need to get the report object from that. This might require some research, but I'll just go to the conclusion. You first need to get the window list from the dashboard. This should consist of one window. You can then use that window to save out the PDF file. Using the ***app*** object from our prior code snippet, we end up with:

```
winList = app << Get Windows();
```

```
dashWin = winList[1];
dashWin << Save PDF("c:\Discovery Demo\Tucson\DashboardOfPollutants.pdf");
```

Now we have a static, or unchanging, copy of our output saved to disk.

The next thing we will do is share our report with the world. JMP 14.2 introduced JMP Public, a report collaboration site. Now, any JMP 15.0 user can publish a report to the site. Anyone, JMP user or not, can view and interact with the reports at https://public.jmp.com . There is no charge to use the site. Just remember that, unless you uncheck "Publish Data", the data that is used to generate the report will be uploaded to the site to allow interaction with the report. It is possible to set the publication of the report such that only the publisher can see it, but by default it will be visible to anyone visiting the site. If you do not include data, the report will not be interactive. There is a nice UI to publish your reports if you are not trying to automate your output:



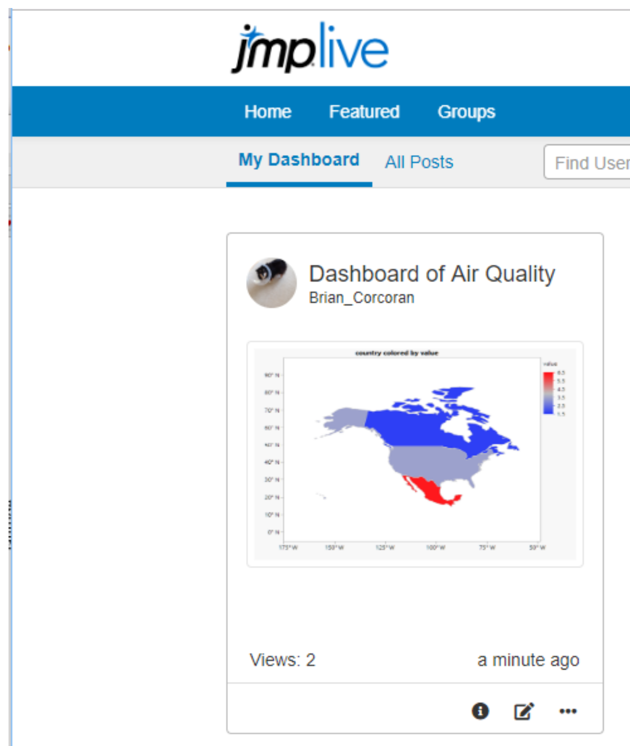
Publish UI for JMP Public

This is available from the File Menu. Select **Publish…** and within the UI select the dropdown for **Publish to JMP Public** instead of **Publish to File.**

We want to automate this though, so that no user interaction is required. The JSL Scripting Index is our friend again. We will want to look at **New Web Report**. This will show the syntax for creating a typical report. The **Publish** action allows us to push the report up to JMP Public. Since we already have the report object from our previous work, the job is somewhat simpler. The JSL looks like:

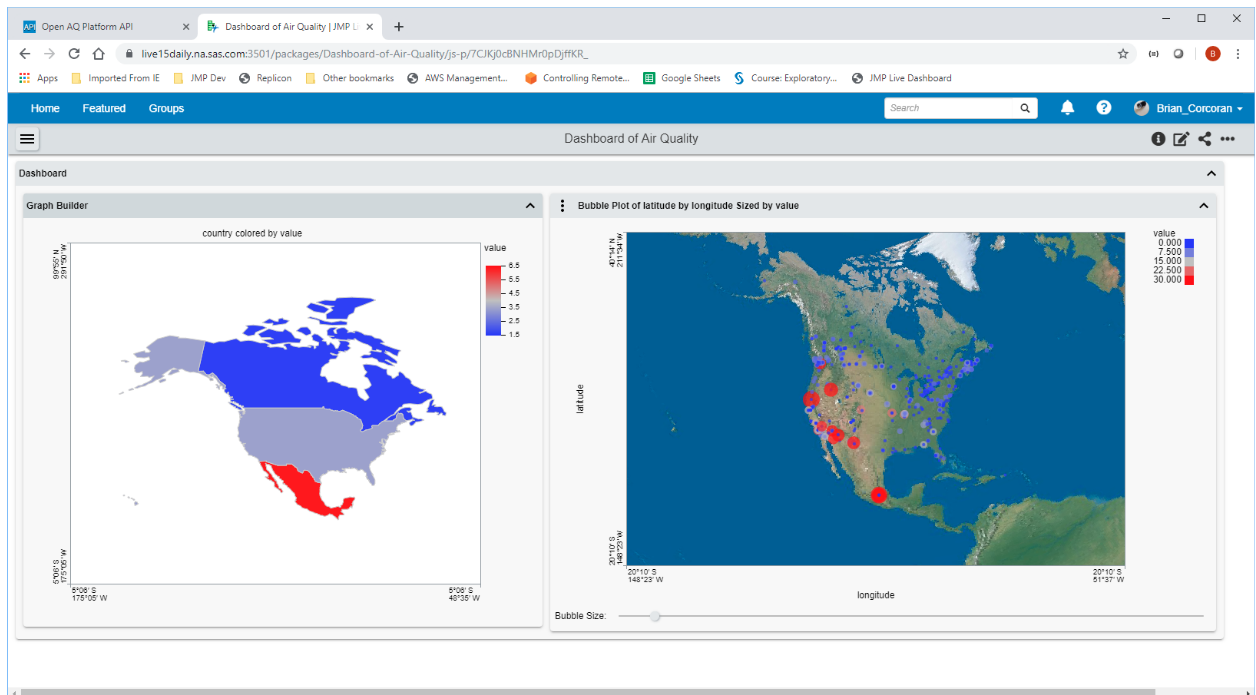
```
webreport = New Web Report();
webreport << Add Report( app, Title(“Dashboard of Air Quality”) );
use_data = "true";
url = webreport << Publish( URL("https://public.jmp.com"),
Username("Someone.somewhere@jmp.com"),Public(1),
Password("My_SAS_Profile_Password"), Publish Data(use_data) );
If( !Is Empty( url ),
      Web( url )
);
```

JMP Public requires you to have a SAS Profile already set up to publish to the site. You do not need this to view reports that are Featured, but you do for everything else. If you want to publish your reports so that only you can see them, either omit the Public(1) option or use Public(0). Upon running this script, JMP Public will update and you will see the report tile on the main page:



Report Tile

If you open the report, you'll see the dashboard in the same layout. You can hover over the maps to get tooltips with the underlying data.



Part of the JMP Public dashboard report

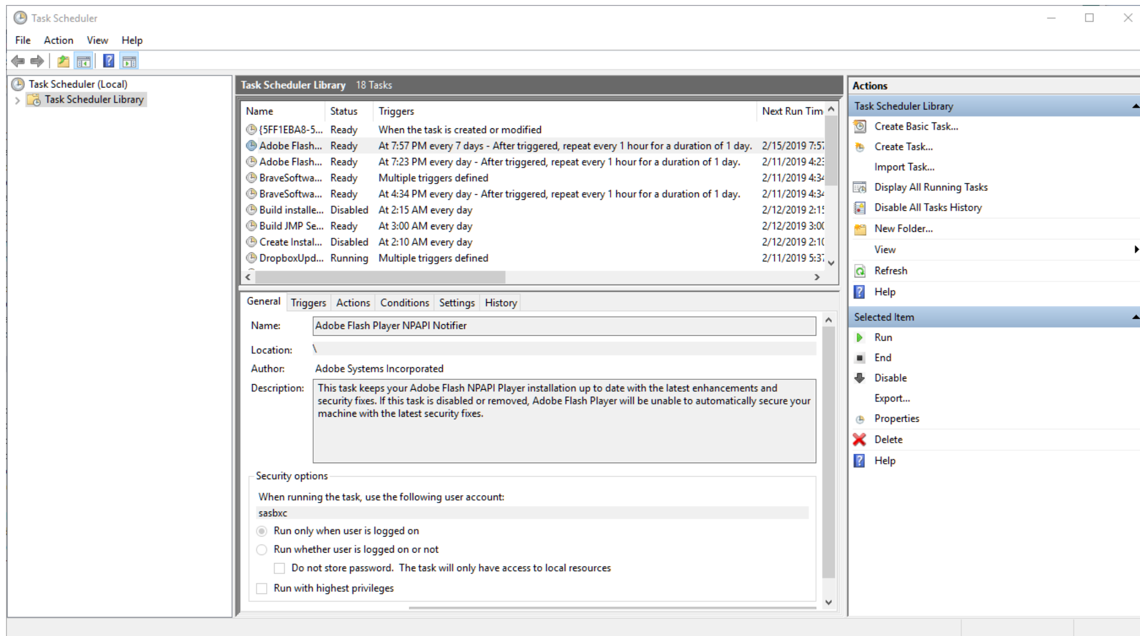
Once we are satisfied with the output to JMP Public, it is worth running the complete WorkingScript.jsl file from start to finish. Any instances where input is requested from a user must be fixed or our automated process will not work. We need to also issue a **Quit** command in JSL to shut everything down. For an automated job, we don't want to leave JMP running because the next job will start yet another JMP. The easiest command is:

```
Quit("No Save");
```

We don't really need to save because our report is being retained by JMP Public, and our queries obtain fresh data each day. Now we can work on repeating the task.

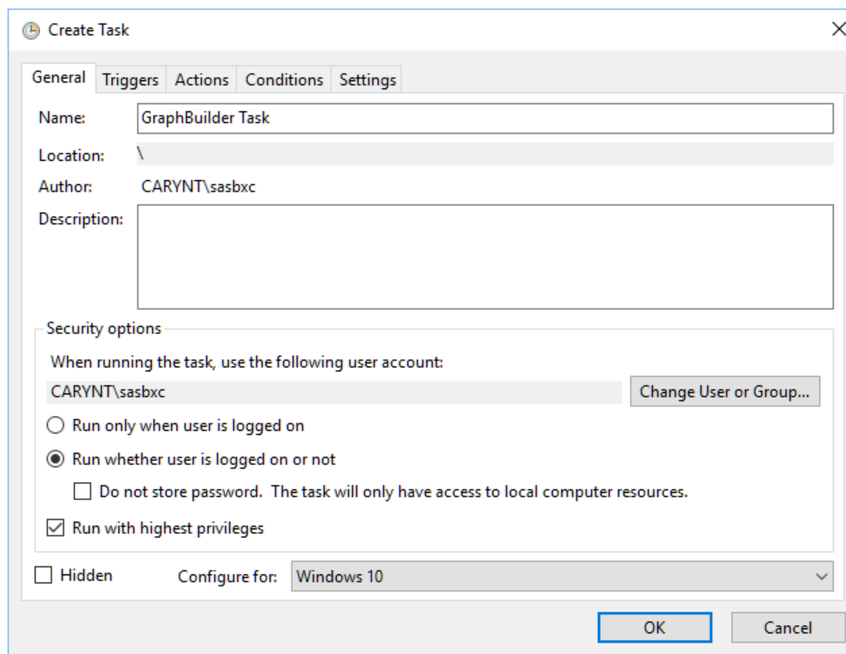
For Windows, the easiest choice is the Windows Task Scheduler. It will provide all the functionality that we need to run this task daily. Automator is probably your choice on the Mac, although you can use a **cron** job if you want to be old fashioned. Showing Automator would take too long for this talk, but there are good resources on the web to help you out.

If you have Windows 7, Task Scheduler is available in System Tools. For Windows 10, I find the easiest thing to do is just type Task Scheduler in the Windows Search bar. On startup the application will look like:



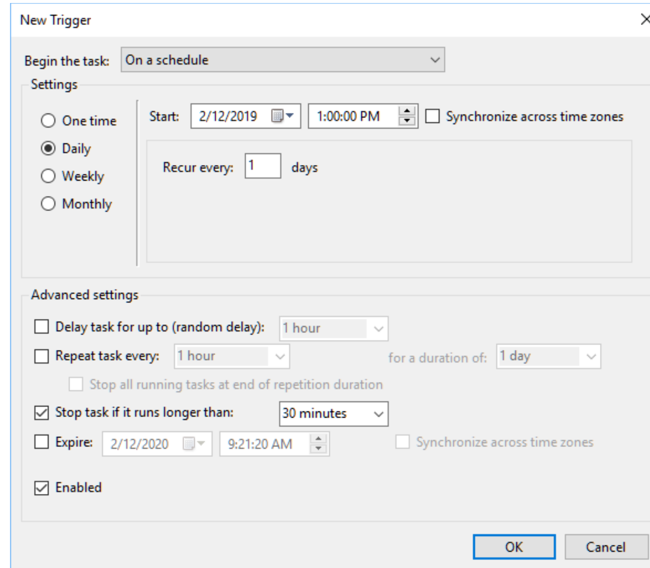
Task Scheduler on Startup

In the right **Actions** pane select "**Create Task…"** Assign a name to the task so you can easily remember it. There is usually a surprising amount of system tasks that fill the task library. Now look at the **Security options** panel. If you have Windows 7/10 Enterprise and are part of a domain, it is important to enter your user account. If the task has to log in to complete its task, it will fail if you do not provide this information. I usually select "**Run whether user is logged in or not"** and "**Run with highest privileges".** I would suggest in **Configure for:** that you select the highest level that your organization can support.



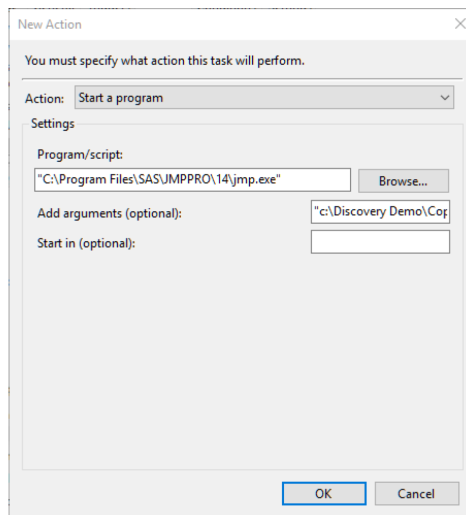
General Pane Completed

Now we can proceed to the Triggers pane and select **New Trigger…** This is pretty self-explanatory. You can select if you want to run the task once, daily, weekly or monthly and you can supply the initial start date and time. For **Advanced settings** I usually specify "**Stop task if it runs longer than…"** as 30 minutes to avoid stalled tasks. Make sure the **Enabled** checkbox is selected.



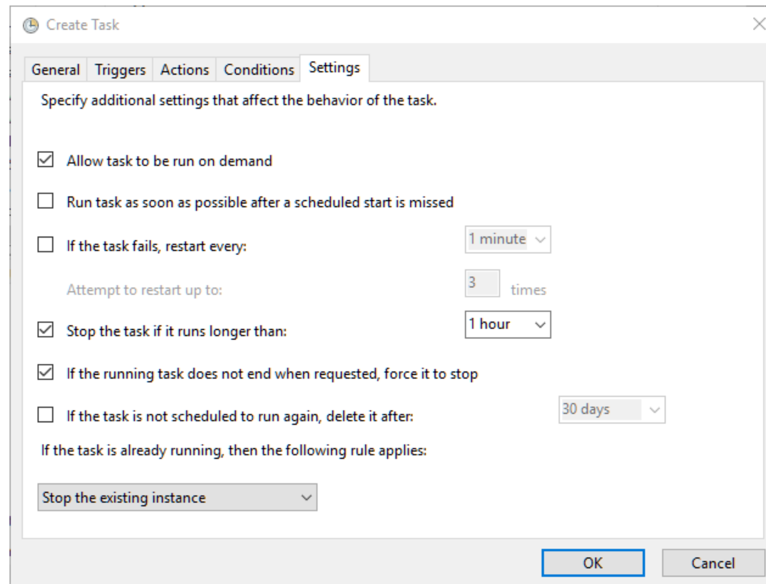
Task Trigger Completed

Once we have set up the timing of the task, we actually get to specify what task we are trying to accomplish. We can now select the **Actions** pane, making sure that "**Start a program"** is selected next to **Action:**. Select the "**Browse…"** button to navigate to the JMP executable in "c:\program files\sas\jmp\15". For the **Add arguments (optional):** setting, we need to fill in the location of our WorkingScript.jsl file. In the case of this tutorial, that is "c:\Discovery Demo\Tucson\WorkingScript.jsl". Now we can select **OK.**



Specify the Script to Run

The conditions panel doesn't really contain anything of interest for our task, so it is fine to move on to **Settings**. I usually check "**Allow task to be run on demand"** because it allows me to test the task whenever I want. I also usually change "**Stop the task if it runs longer than:"** to 1 hour because this task should be done in a few minutes. Finally, I usually specify "**Stop the existing instance"** because if there is a stalled instance from a previous run I want to kill it and start anew.



Task Customization

Now we are finished. We can press **OK**, and we will be prompted for credentials if we said we want to run even if we are logged off. This is so the Task Scheduler can supply the authentication to the OS to run the task.

Now at the specified time JMP should start and run the script that we have developed over the course of the tutorial. JMP Public should have a new report appear, and JMP should be shut down. This is all done invisibly, so the only output we will see is that JMP Public report and any files that we have explicitly saved to disk.

If you are publishing the same report day after day, you will get a new JMP Public package for every publish event. If this is what you would like, it is advisable to append some kind of identifier to the title to make it easy for users to tell the difference between the packages. There is also an option to replace the package that is already in JMP Public with the newly generated report. There are two steps to doing this. First, you must publish the package the first time using the steps we have already discussed. Now, you need to open the report in JMP Public and examine the URL. It might look something like:

https://public.jmp.com/packages/My-Web-Report/js-p/5c62d6f720e8bb0f94e4d49e

The really import part of that is the long sequence of numbers and letters at the end of the URL. That is the package ID. Now you can modify your JSL for publishing to specify the **Replace** option, which requires this package ID. I will now look something like:

```
webreport = New Web Report();
```

```
webreport << Add Report( app );
use_data = "true";
url = webreport << Publish( URL("https://public.jmp.com"),
Username("Someone.somewhere@jmp.com"),Public(1),
Password("My_SAS_Profile_Password"), Publish Data(use_data),
Replace(“5c62d6f720e8bb0f94e4d49e”) );
```

Now you’ll have only one report that will be refreshed at the interval that you’ve specified.

We’ve covered a lot of material with this tutorial, but hopefully you can see the power of the various features within JMP, and that with a little practice you can quickly produce an automated report result for others in your organization to view.

Conventions used:

Window panes, static UI elements and JSL keywords are **bolded**

Buttons / Dialog controls are “**quoted and bolded”**

Variables / Column names are “*quoted and italicized*”