Automated Report Creation: From Data Import to Publication

JMP Discovery Conference – Copenhagen

Brian Corcoran – Software Development Director, SAS

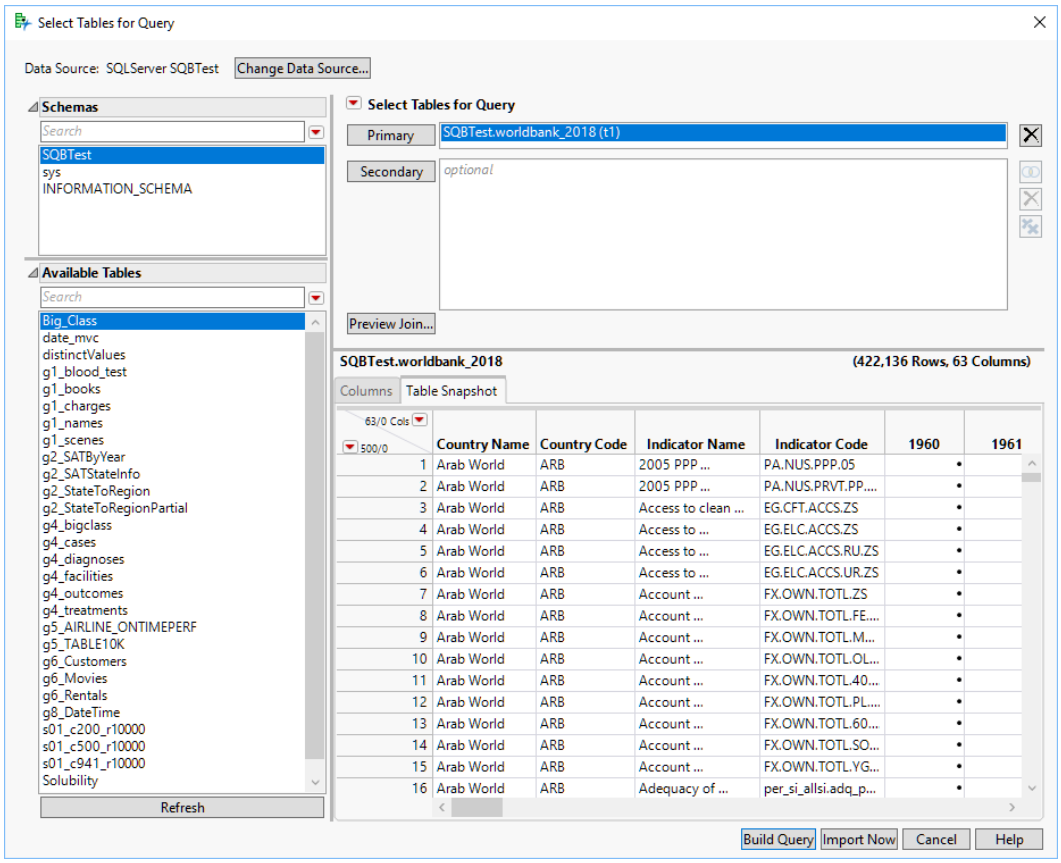
JMP provides a variety of methods to get data into the product. It has a lot of facilities to export finished reports for viewing. However, it is often difficult to envision the end-to-end workflow. A question that often comes up is: How do I take my complete report workflow and automate it so I can come to work in the morning and already have my finished report waiting for me? This tutorial will give an example of how to accomplish this. We will import data from several sources, hopefully giving a brief survey of some 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 is aggregated on one site. Any of the cities involved can update the data on a given day, so the data can change frequently. We might also have data that is updated less frequently, but shows information as to how energy is produced. It would be interesting to see if the sources used to produce energy affect air quality.

Let's take a look at the infrequently updated data first. This is data that comes out of a database. In this case, I am using data from the World Bank. I've used this data before looking at other topics. We want to get the World Development Indicator data, which is publicly available.[1] We can open JMP and select File->Database->Query Builder, open a database connection and point to our desired table.

If we look at the data in the Query Builder preview window, using **Table Snapshot**, we see a country name, a country code, an indicator name and then columns worth of numbers. It is the indicator name that we are really interest in, because this is the topic being measured. The country name contains phrases that are unfamiliar, like "Arab World". So, it is likely we are looking at data that has groups of countries as well as individual countries. If we actually press "**Import Now"** and open the table, we can see that there are 400,000+ observations all organized by the country name. Making a subset of this data and organizing it so that the indicator is the primary focus is going to be the first problem that we face.

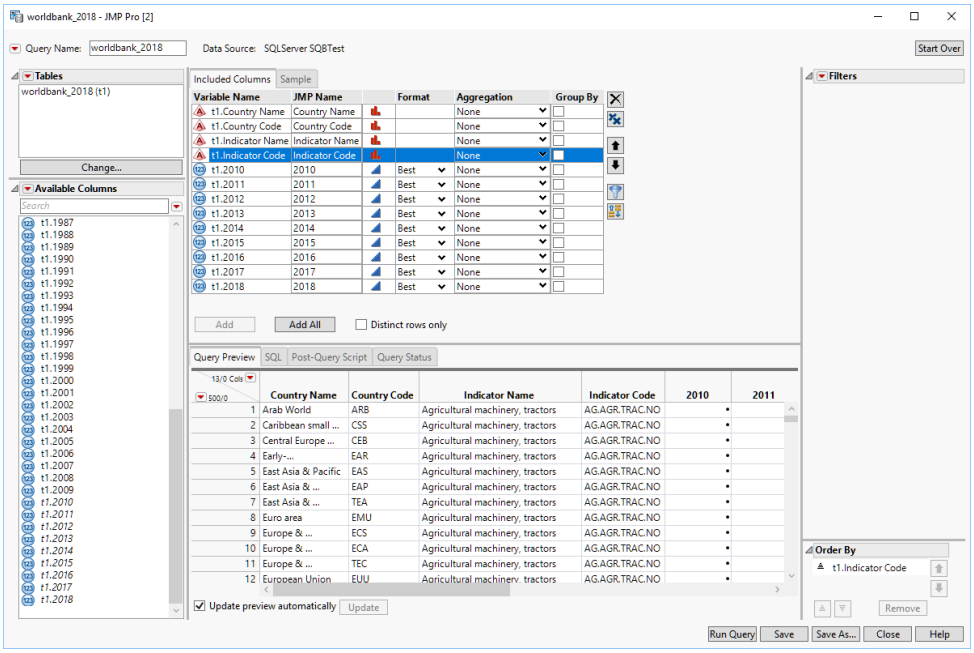
[1] World Bank Group. *World Development Indicators*, World Bank Group, n.d. Web. February 5, 2019. https://datacatalog.worldbank.org/dataset/world-development-indicators . Obtained under Create Commons CC BY 4.0 license.



Query Builder Table Snapshot

Rather than just import the data into JMP, it is worth going forward with Query Builder using **"Build Query"**. We can select the country and indicator information, as well as the data columns for 2010 – 2018. This should be adequate for our purposes.

The first thing we can do to is take the "*Indicator Code*" field and use that to order our data. Each topic of study has a unique code, and it this that will allow us to find data regarding electricity generation.



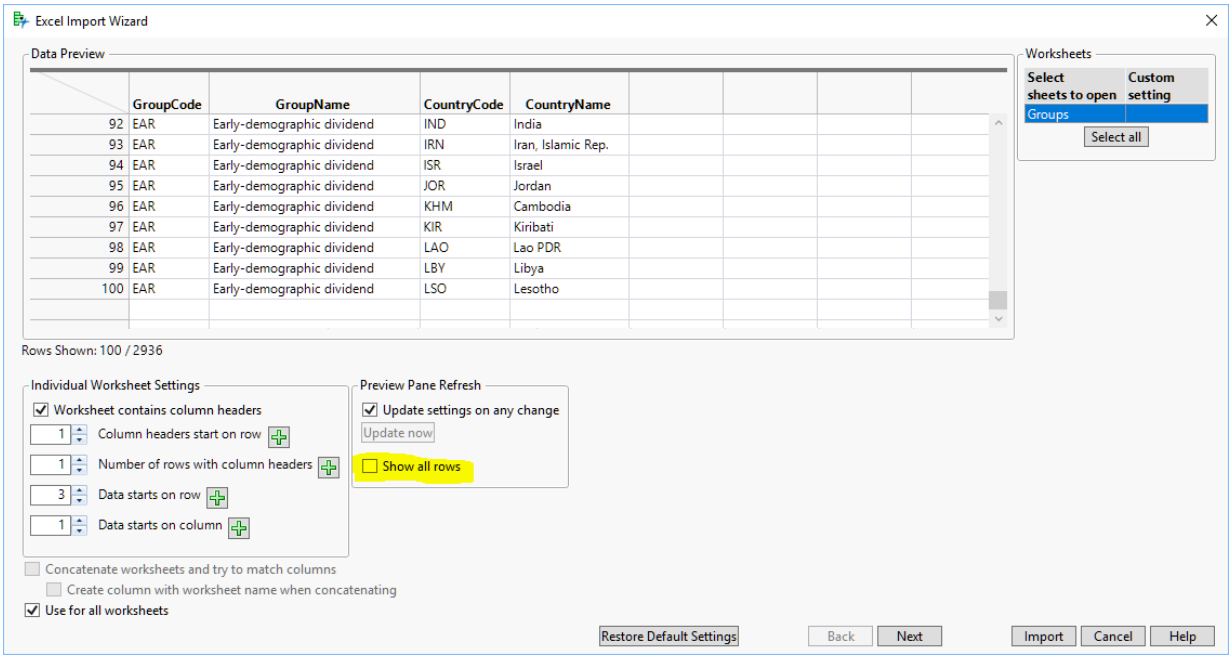
Order by Indicator

Now we can concentrate on looking at a select group of countries. Since we are in Copenhagen for a conference, it would be worth looking at some of the Scandinavian countries, and then some other countries in Europe that would serve as useful comparisons.

It would be nice to reduce the countries to a more manageable list. There are really many ways to do this. One way is to first get the data down to the European countries. The World Bank dataset clearly has groupings of countries, but it is unclear looking at the data what the groups contain. In fact, I think it takes quite a bit of effort to find out. The countries contained in groups like "*Arab World*" can be obtained on a webpage called "World Bank Country and Lending Groups" [2] It is (currently) obtained via a hyperlink entitled "current classification by income in XLS format". You can download this Excel file named CLASS.XLS and open it in JMP with the Excel Wizard.

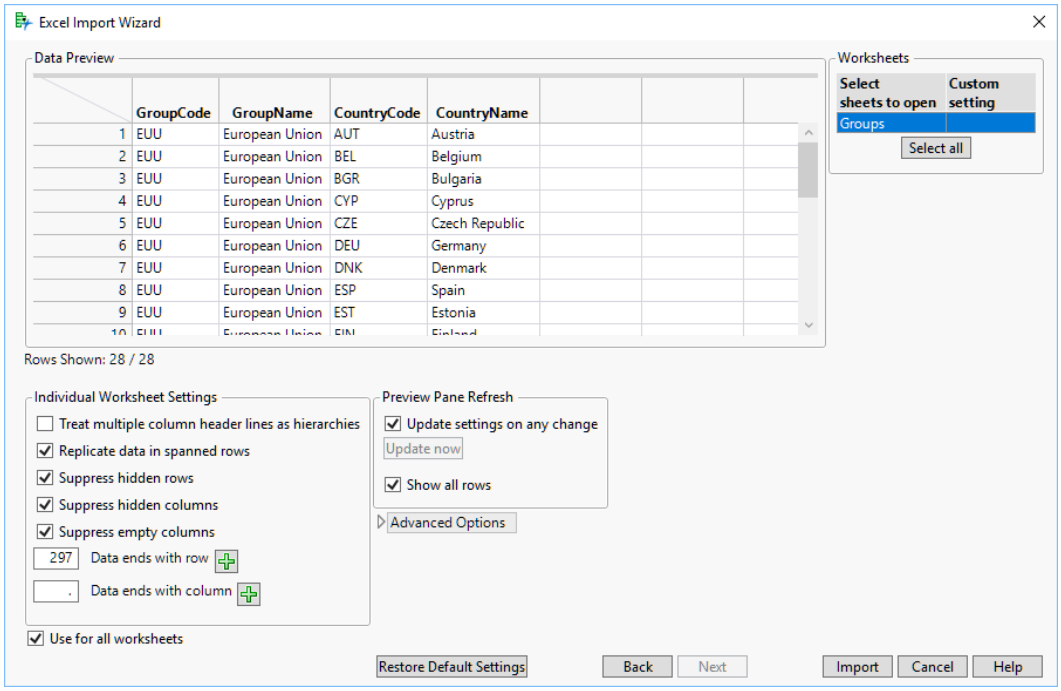
The Excel Import Wizard preview window only shows the first hundred rows of data by default to provide good performance. Scrolling down shows groups like "*Early-demographic dividend"* but there is nothing that starts with Europe.

[2] World Bank Group. *World Bank Country and Lending Groups*, World Bank Group, n.d. Web. February 5, 2019. https://datahelpdesk.worldbank.org/knowledgebase/articles/906519 . Obtained under Create Commons CC BY 4.0 license.



Only 100 Rows Show by Default

If you select the "**Show All Rows"** checkbox you can see the complete table. It turns out there are a variety of European groupings. For simplicity I'm going to select the European Union using the **"Data starts on row"** and **"Data ends of row buttons"** to select the 28 rows of countries.

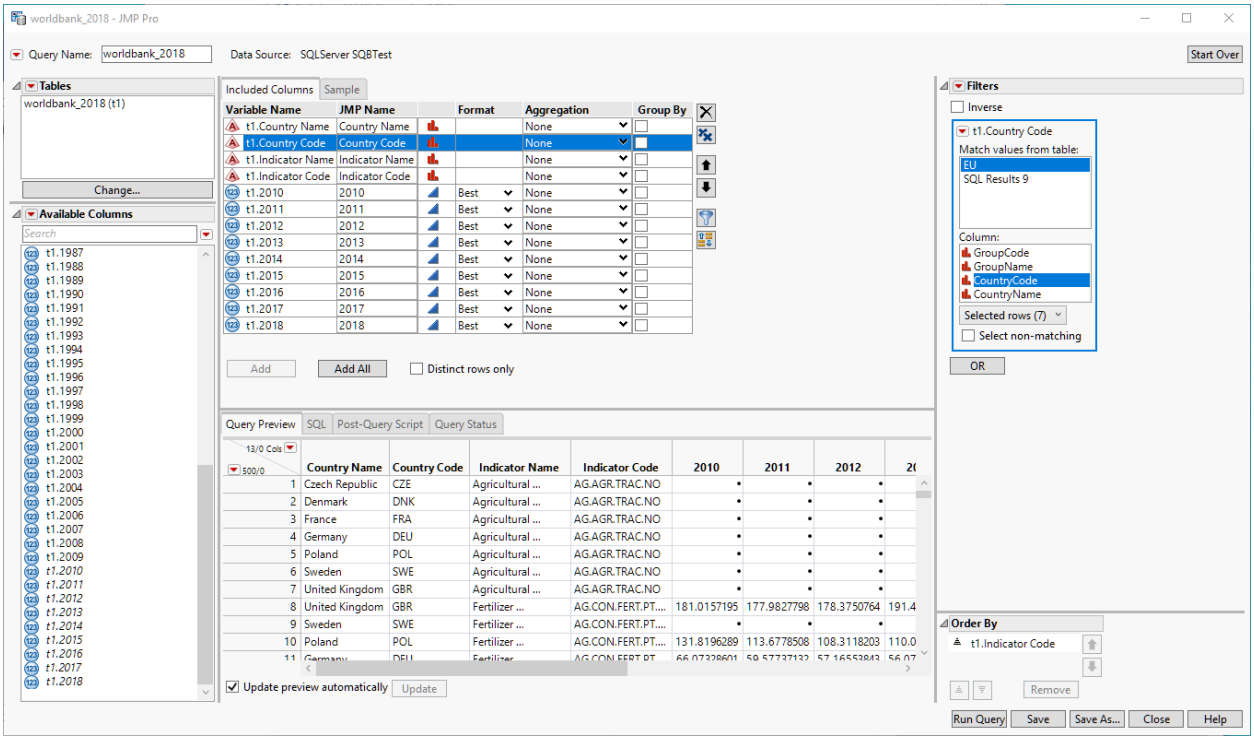


EU Selected

Now I can just press the **"Import"** button and bring the table into JMP. I'll go ahead and save that out as EU.jmp. Now I can return to my query. I can drag *"t1.Country Code"* over from **Included**

Columns into the **Filters** pane. A list of all of the countries will show up. What I want to do is only include the list of EU countries. I can go to the red triangle menu next to *"t1.Country Code"* in the filter and select Filter Type->Match Column Values. A list of open tables comes up, with our EU table in it. I select this, and then specify that I want to match the column *"CountryCode"*. For now, I can select the **All rows (28)** in the dropdown list. Now I see just the EU country values.

Now I'm going to return to my EU table. I'm going to select the countries I'm interested in for my report. I'll go with Czech Republic, Germany, Denmark, France, United Kingdom, Poland and Sweden. Now I'm going to press File->Save on the EU table. This is to save the selection state. I now return to my query and change the dropdown for our filter to **Selected rows (7)**.

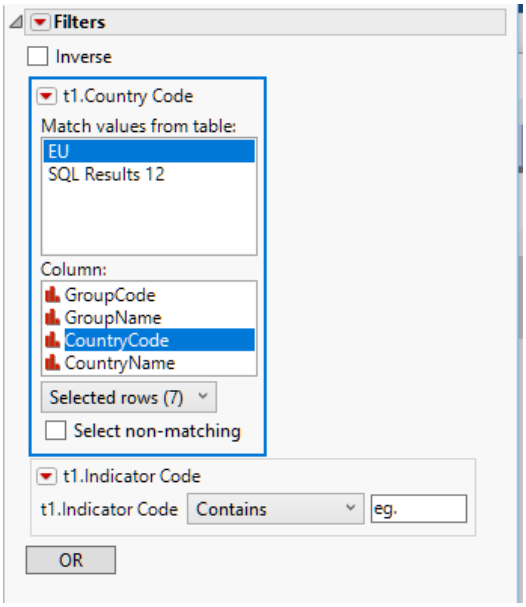


Matching Columns Filter

Now I have an even smaller subset of the data. If I were to import the data now, I'd see around 11,000 rows. We are down to 7 countries, but 11,000 rows means we have a lot of different indicators, or subjects that were measured. It would be useful to only look at the indicators that are related to energy usage and power production. The *"Indicator Codes"* field is described in the World Bank site if you hunt for it. [3] There is a description of the methodology, and an Excel file containing the indicator descriptions and codes. Suffice it to say that the topics that we are interested in all start with "eg." In the indicator code. We can use this to further refine our filter. I'll drag the "*t1.Indicator Code"* variable name from the **Included Columns** pane into the **Filters** area. This will present a list of all the indicator

[3] World Bank Group. *How does the World Bank code its indicators*, World Bank Group, n.d. Web. February 5, 2019. https://datahelpdesk.worldbank.org/knowledgebase/articles/201175-how-does-the-world-bank-code-its-indicators. Obtained under Create Commons CC BY 4.0 license.

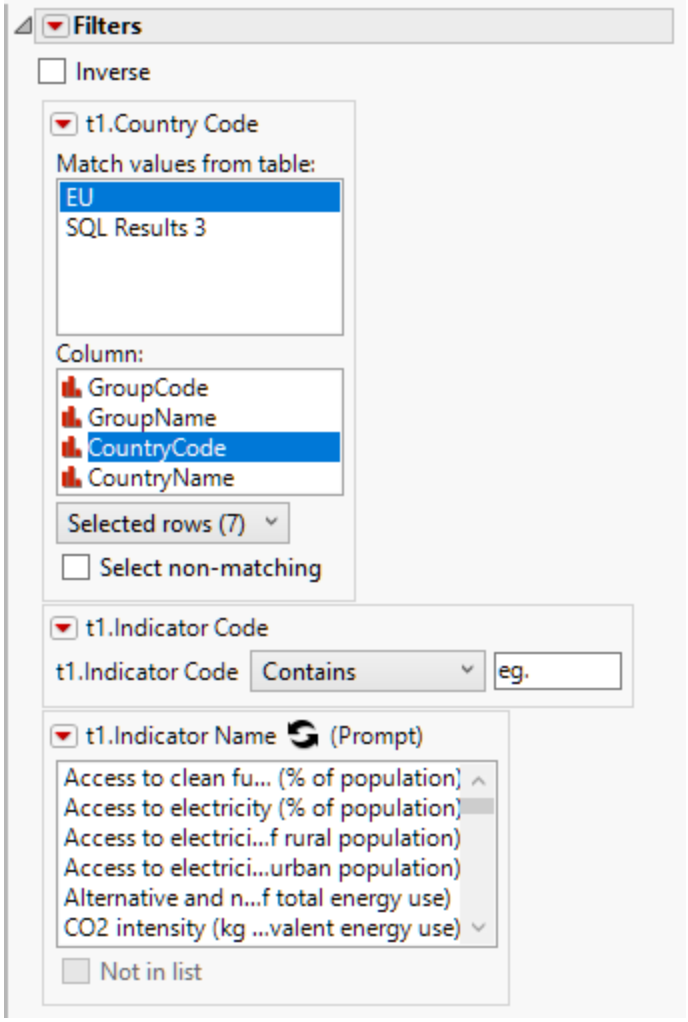
codes. Now I can select Filter Type->Contains from the red triangle menu and enter “eg.” Into the edit box next to the **Contains** dropdown.



Contains Filter

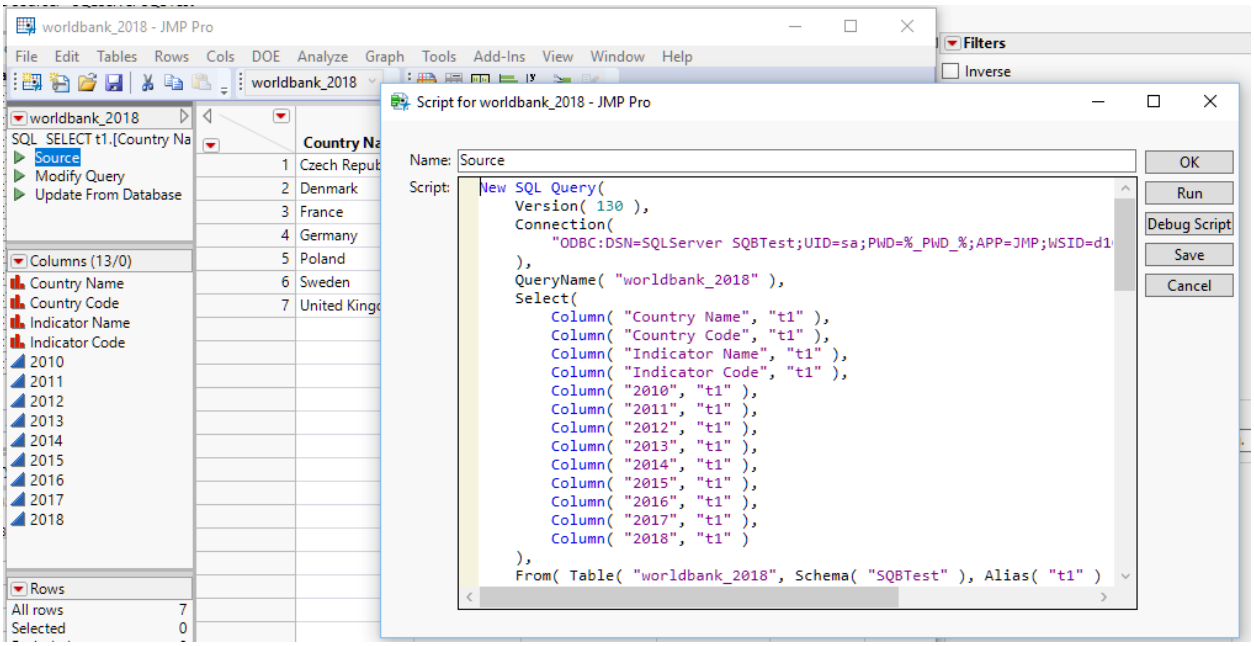
The preview will refresh, and we press **Run Query** now we will be down to less than 400 rows. At this point, it is a good idea to save the query as we are getting close to having something useful. If we look at this output, we have all the indicators that contain “.eg”. Unfortunately, a few are from unrelated embedded indicators, but we will proceed with this subset. If we want to do an analysis, we can sort on the indicators within the data table and select the group we want. However, we can do one step better in our query.

First, we can drag “*Indicator Name”* over below the filter for “*Indicator Code”.* Now, we can select the red triangle and **Conditional.** What this does is says that we want to produce a list of indicator names whose contents derive from the previous filters that we set up. In other words, we only want the indicator names that derive from fields in the countries we’ve selected and that have “.eg” in the indicator code. If we select **Prompt** as well in the filter, we are ready to go. Now, when we run the query we will get a dialog containing the indicators that interest us. If we select a topic, for instance “Electricity production from oil, gas and coal sources (% of total)”, then only the 7 rows of data for that indicator and our selected countries show up.



Filter with Conditional Indicator

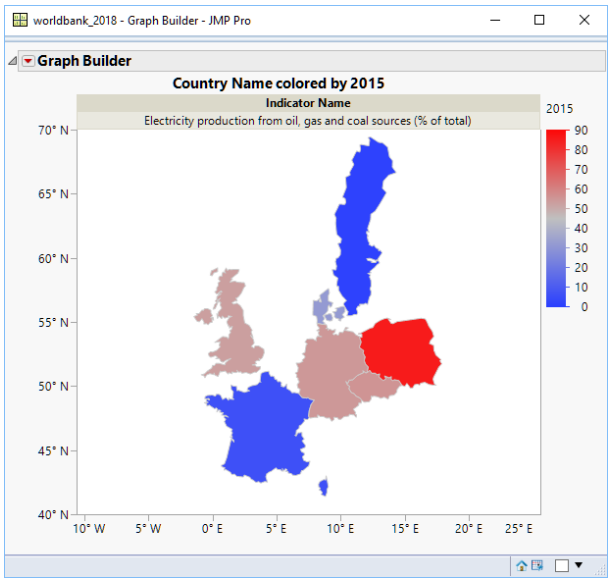
If we look at the resulting data table in JMP, we see a panel on the top left with a snippet of the SQL that generated the table. More importantly, we see red triangle menus for **Source, Modify Query, and Update From Database.** The **Source** script, which you will find in many tables that are imported, contains the JSL necessary to regenerate the table. **Modify Query** will bring up the Query Builder design panel again. **Update From Database** will fetch the current data and repopulate the existing table. It is the **Source** script that we are interested in now. If we right click on the green triangle and select Edit, JMP will bring up a dialog showing the JSL to recreate the table. This contains all of the work we have been doing up to this point, referencing the EU.jmp table that we imported. If we are providing a query to give to others to use interactively, this part is done. However, if we want to later automate this the UI that comes up will be an issue, as it will stop processing. Let' shut off the prompt and select the field we are most interested in looking at, "Electricity production from oil, gas and coal sources (% of total)". Now let's re-run the Query and open the Source script from the resulting table.



Source Script for Finished Query

We can run or debug the script right from this dialog, but for now let's just copy the contents to a new script and save that out as WorkingScript.jsl. This will be the basis for a larger task that we will be running later. It is worth modifying the script at this time to supply the database credential to avoid a prompt. You can obfuscate the script later if you want to prevent others from seeing the information.

We could go ahead and create a graphic with the data right now just to make sure we have what we need. We can open Graph->Graph Builder from the JMP menu and work on our result set data table. If we drag *"Country Name"* into the Map Shape rectangle of Graph Builder, it will draw the outlines of the countries that we are examining. Now we can drop in 2015, the last year with good data, into the center drop zone. Finally, we can drop in *"Indicator Name"* into Group X and look at the result



An example graph using our query

We can go ahead and save the script from that Graph Builder output to our WorkingScript.jsl file. Let's assign the result to an object in the JSL in case we want to use the graph later.

Now we are ready for the more frequently updated data. The air quality data comes from a website and organization called **openaq**[4]. 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 we have been looking at previously is getting migrated to a databank accessible through REST interfaces. The goal with this tutorial 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 this, 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 "London":"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. Probably the key JSL snippet for this example is the following:

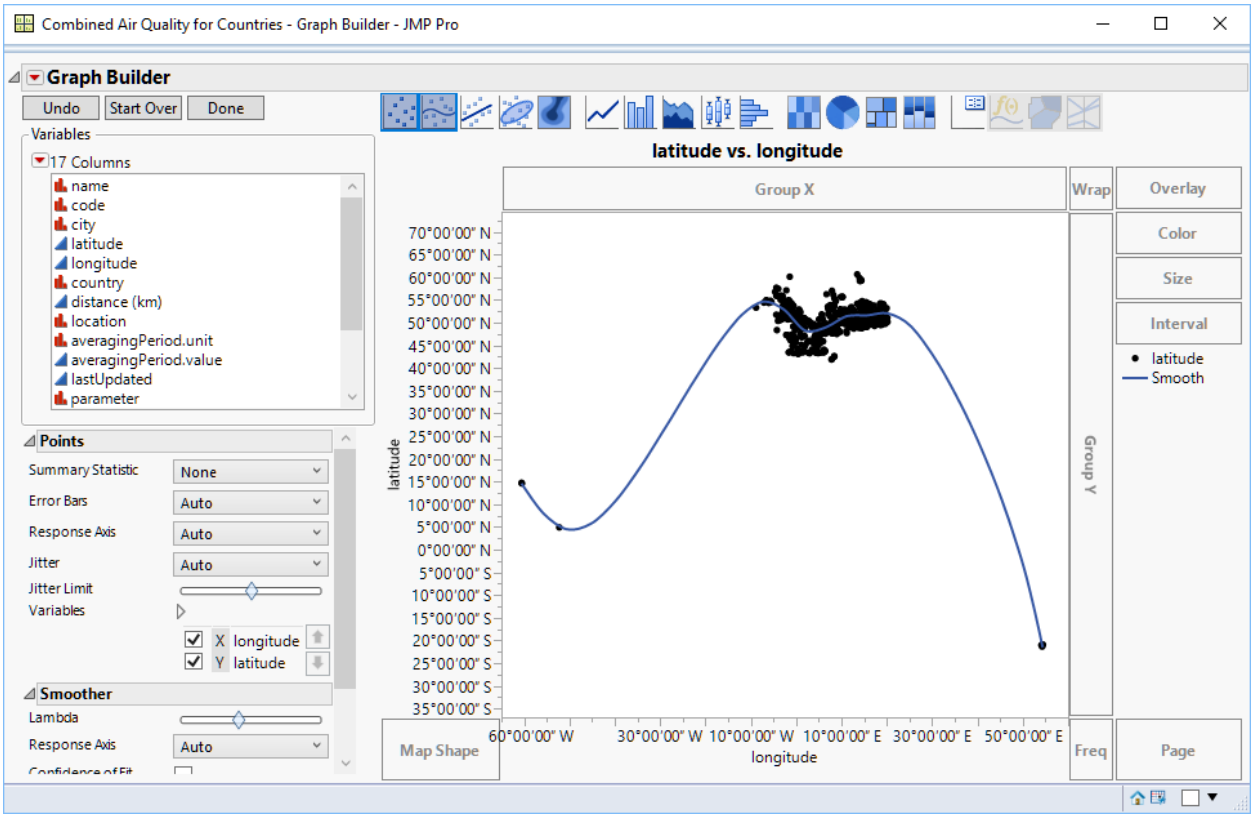
```
query = [=>];

query["country"] = code;
query["limit"] = 200;
request = New HTTP Request (
	URL("https://api.openaq.org/v1/latest"),
	Method("Get"),
	Query String(query)
);
json = request << Send();
```

Here we create a 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. The key takeaway from this tutorial is that you can assemble data from lots of places and repeat it using standard techniques.

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 thousand 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.

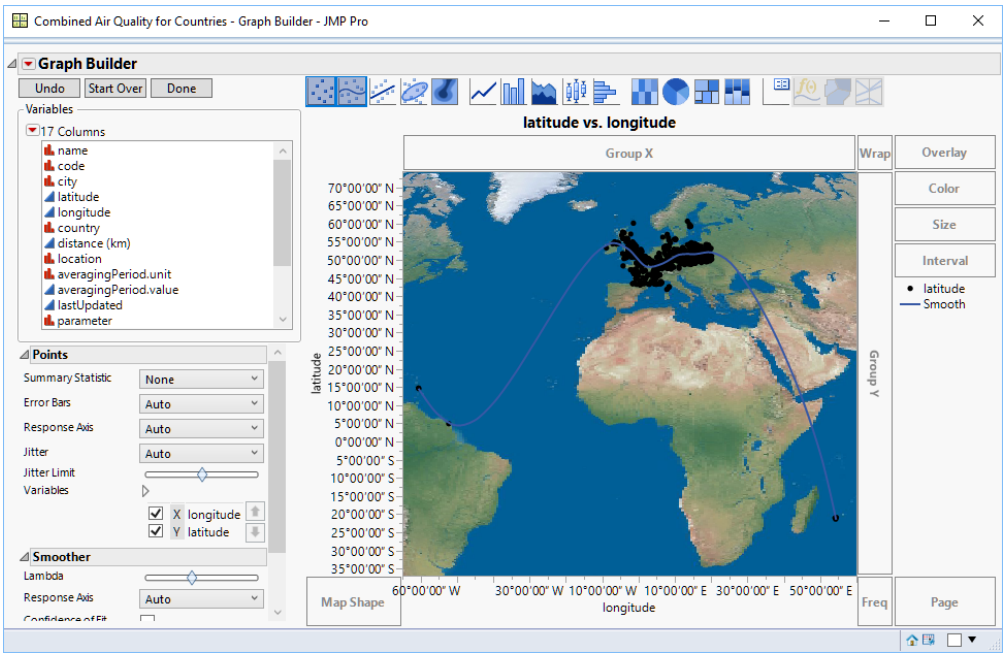
[4] openaq. *Air Quality Data*, openaq, n.d. Web. February 5, 2019. https://openaq.org . Obtained under Create Commons CC BY 4.0 license. Web API data obtained from https://api.openaq.org.

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

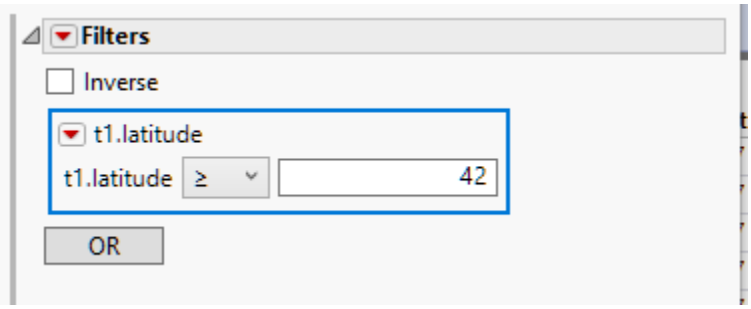


Data with Map

It looks like the points are there on purpose. Incorrectly mapped points often show up in clusters by the poles or equator. In fact, it turns out the issue is that France includes its territories in its air quality measurements. So, French Guiana and St. Martin are included. We've been restricting ourselves to European areas, so it would be nice to exclude those points. It's time for Query Builder for JMP files. Before we do this, we better put in a JSL snippet into our scripting to save the data to disk. This would look like:

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

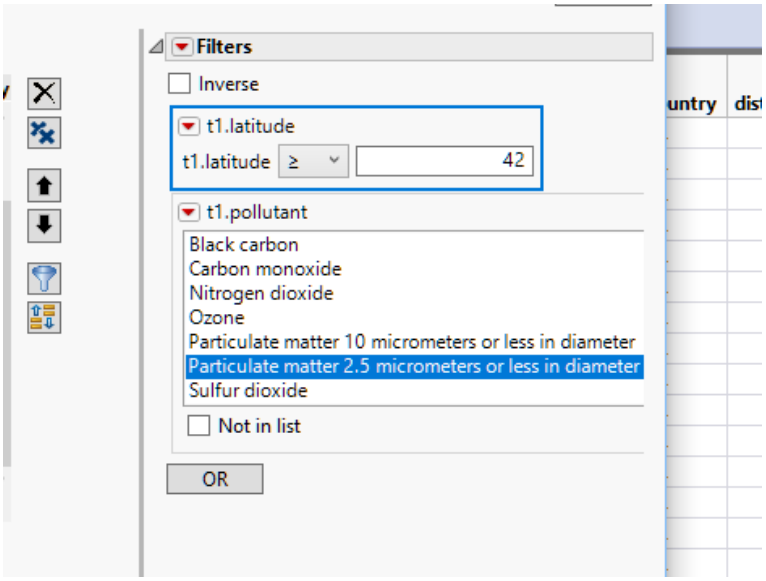
Now we can use Tables->Query Builder on the result table from our HTTP Request script. The first thing to do is to apply a filter to the latitude. Dragging *"t1.latitude"* over the filters area and providing a value of 42 reduces our observations to Europe.



Reduce Observations to Europe

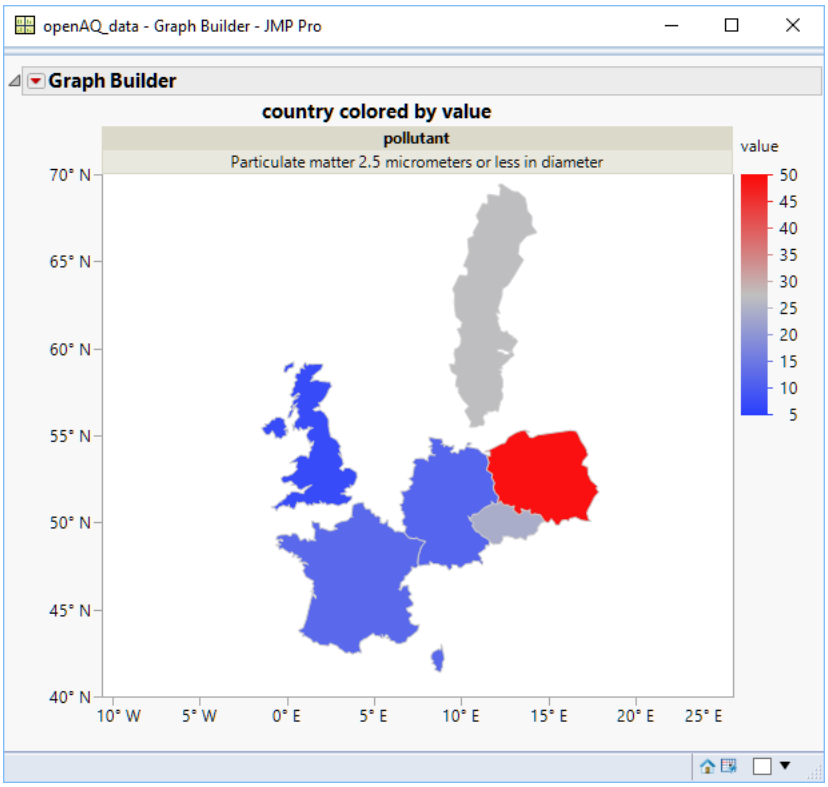
Now we can look at data for the same region that we examined with the World Bank data. We can drop in a filter for the pollutant. This will allow us to examine the country data for each topic individually. The easiest way is to make *"t1.pollutant"* one our filters. We can then generate a few query results 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 "Carbon Monoxide" and "Particulate matter less than 2.5 micrometers in diameter". Unfortunately, Denmark does not report the second value, but it is an important one and worth looking at.



Filtering on Pollutant

Now if we run this query we get just the values for "Particulate matter…" and we can produce a graph based on this. If, instead of latitude and longitude, we use "*country*" as the Map Shape and *"value"* as our data, Graph Builder will color the countries by the mean value of this pollutant.



Graph of Mean Particulate Matter

If I save the script out at this point to our WorkingScript file, you would see something like:

```
Graph Builder(
	Size( 534, 490 ),
	Show Control Panel( 0 ),
	Variables( Group X( :pollutant ), Color( :value ), Shape( :country ) ),
	Elements( Map Shapes( Legend( 6 ) ) )
);
```

This is going to operate on the current data table. It is worth showing part of the query script from the result table at this point, which we will also want to copy to the WorkingScript:

```
qdt1 = New SQL Query(
	Version( 130 ),
	Connection( "JMP" ),
	JMP Tables(
		["Combined Air Quality" =>
		"\c:\Discovery Demo\Copenhagen\Combined Air Quality.jmp"]
	),
	QueryName( "openAQ_data" ),
	Select(
		Column( "name", "t1" ),
		Column( "code", "t1" ),
		Column( "city", "t1" ),
		Column( "latitude", "t1", Numeric Format("Latitude DMS", "0", "NO", "")),

	)
) << Run
```

The last part in particular is 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. Our Graph Builder 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);
```

Then we can run our Graph Builder script that we got from doing the report through the user interface:

```
gbPart = Graph Builder(
	Size( 534, 490 ),
	Show Control Panel( 0 ),
	Variables( Group X( :pollutant ), Color( :value ), Shape( :country ) ),
	Elements( Map Shapes( Legend( 8 ) ) )
	);
```

Now, it would be nice to combine these two graphs with the graph that we created earlier about the amount of electricity generated from fossil fuels. 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 three 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 three graphs.

Now you can use the dashboard red triangle menu to do "Save Script->To Script Window". The problem with this is that App 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.

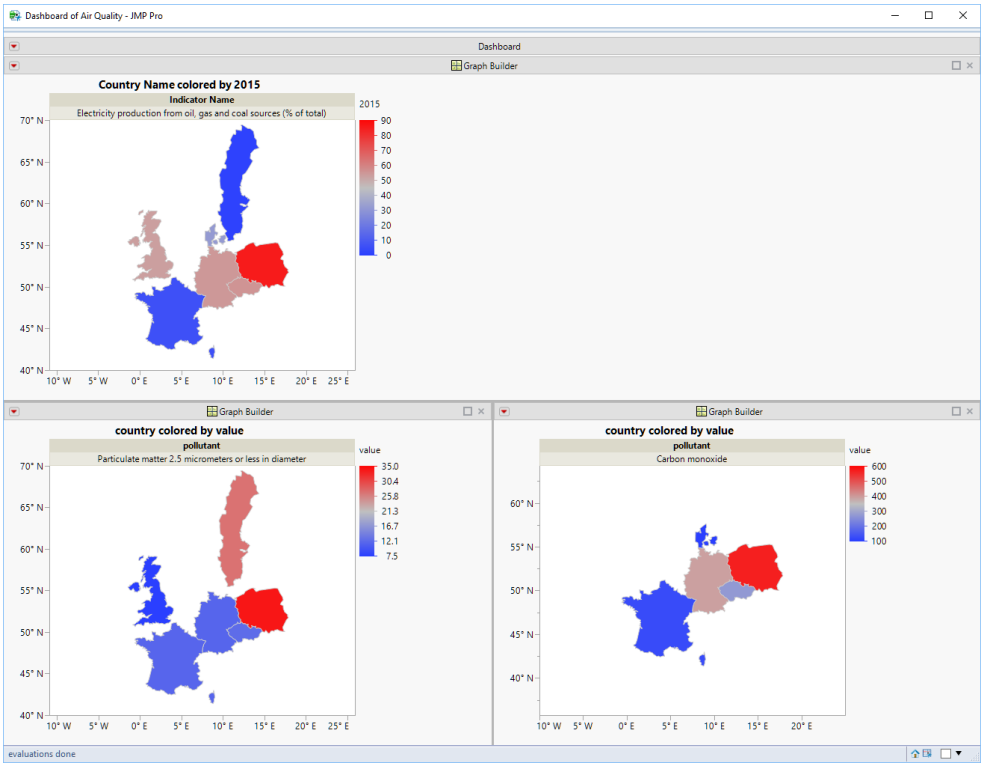
So, we can include a simple JSL snippet to do the Combine Windows. It looks like this:

```
app = JMP App();
app << Set Name( "Dashboard of Air Quality" );
app << Combine Windows( {gbElectric << Report, gbPart << Report, gbCO <<
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 Electricity and Pollutants

The nice thing with the Combine Windows script snippet is that it gives us an Application object, which is also recognizable as a Report. It is a Report that is easily saved out to other output formats. For example, if we want to save our dashboard out as a PDF file, we can again consult the JMP Scripting Index and get a snippet like this:

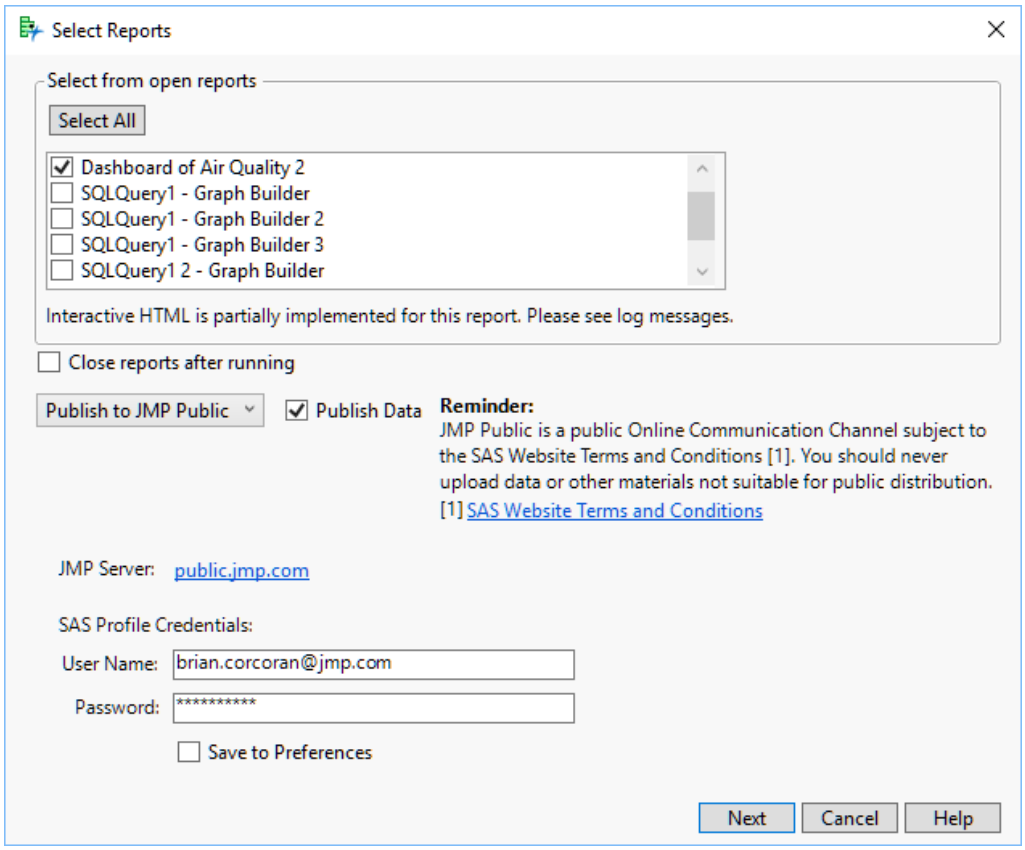
```
biv = bivariate( y( :weight ), x( :height ) );
rbiv = biv << report;
rbiv << Save PDF( "path/to/example.pdf" );
```

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\Copenhagen\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. Any JMP 14.2 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:



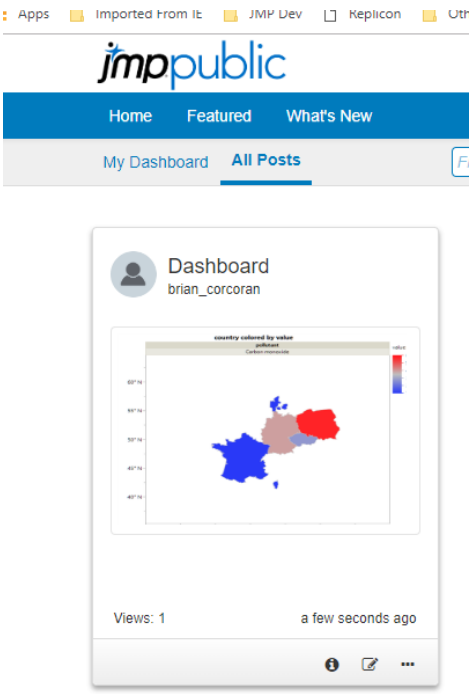
Publish UI for JMP Public

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:

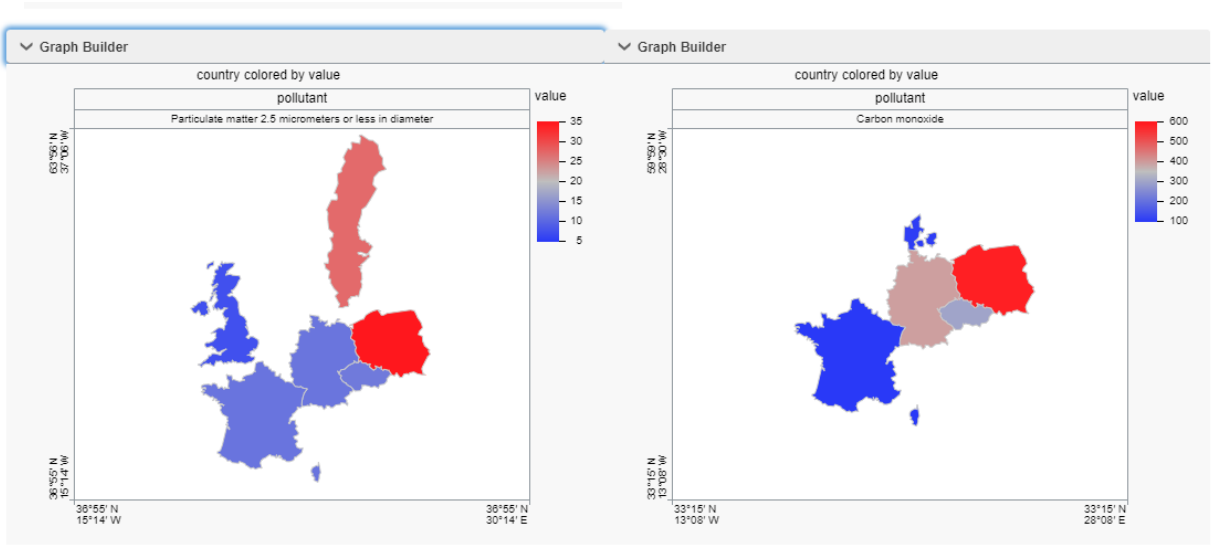
```
webreport = New Web Report();
webreport << Add Report( app );
use_data = "true";
url = webreport << Publish( URL("https://public.jmp.com"),
Username("Someone.somewhere@jmp.com"),Public(1),
Password("My_SAS_Profile_Password"), Publish Data(use_data) );
If( !Is Empty( url ),
      Web( url )
);
```

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 public. 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 countries to get tooltips with the underlying data.



Part of the JMP Public dashboard report

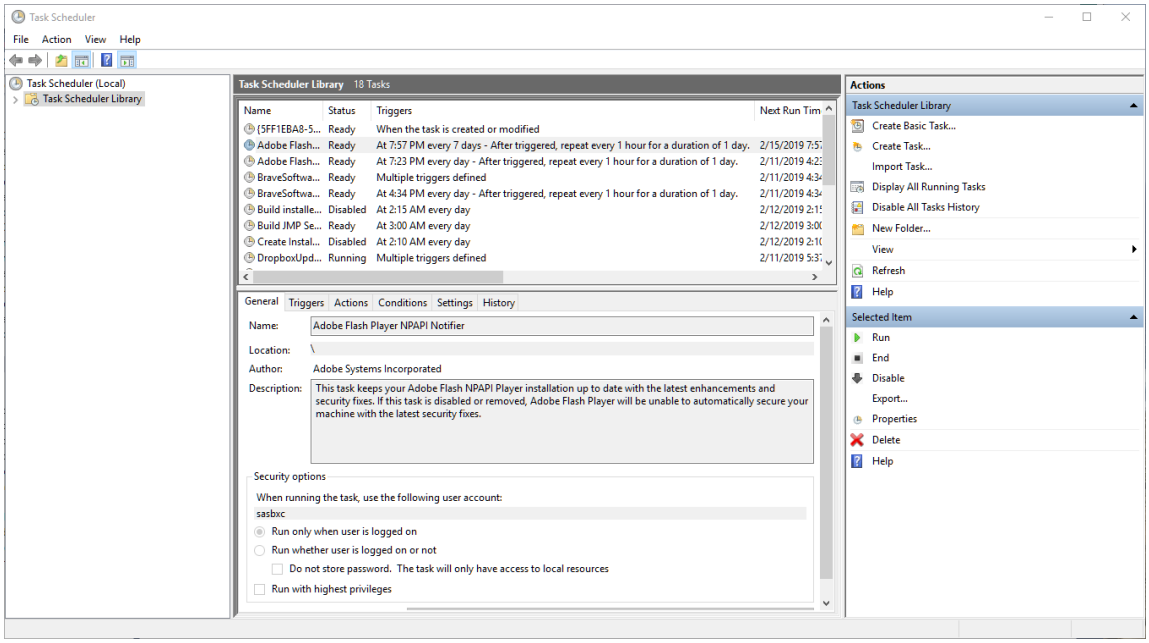
Once we are satisfied with the output to JMP Public, it is worth running the complete WorkingScript.jsl file from start to finish. Any instances where input is requested from a user must be fixed or our automated process will not work. We need to also issue a **Quit** command in JSL to shut everything 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.

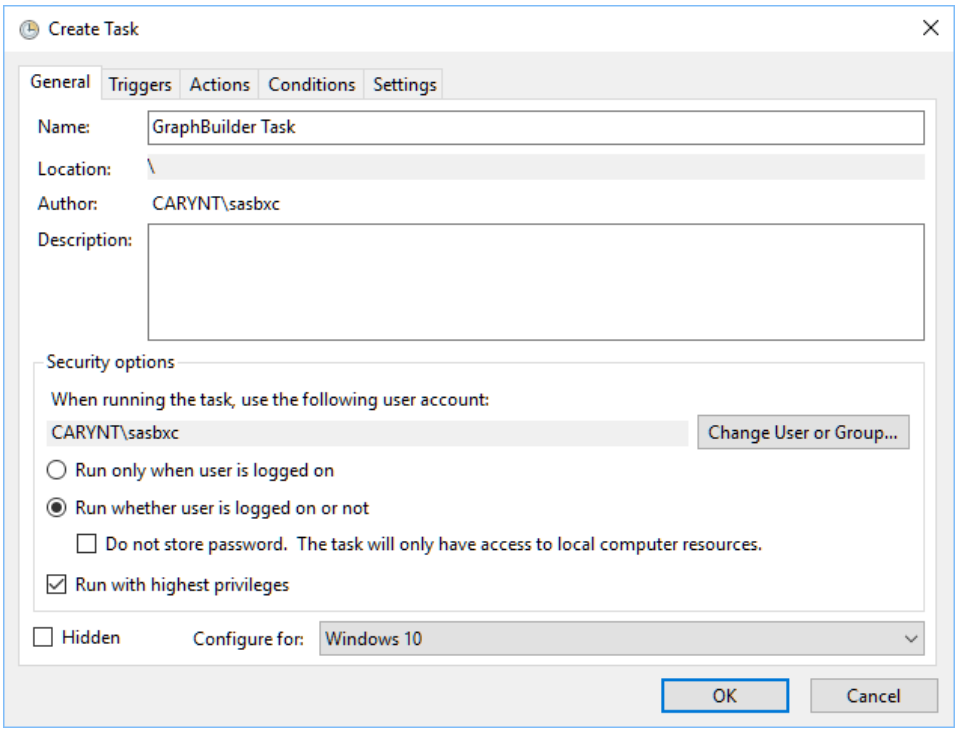
For Windows, the easiest choice is the Windows Task Scheduler. It will provide all the functionality that we need to run this daily. Automator is probably your choice on the Mac, although you can use a **cron** job if you want to be old fashioned. Showing Automator would take too long for this talk,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



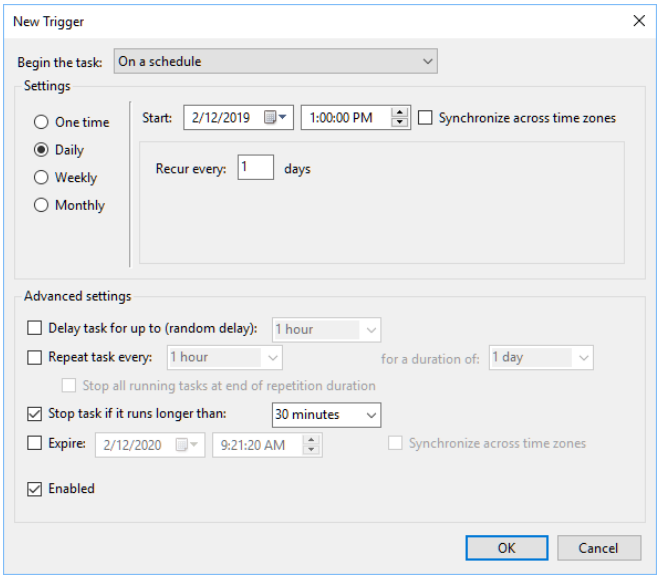
Task Scheduler on Startup

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

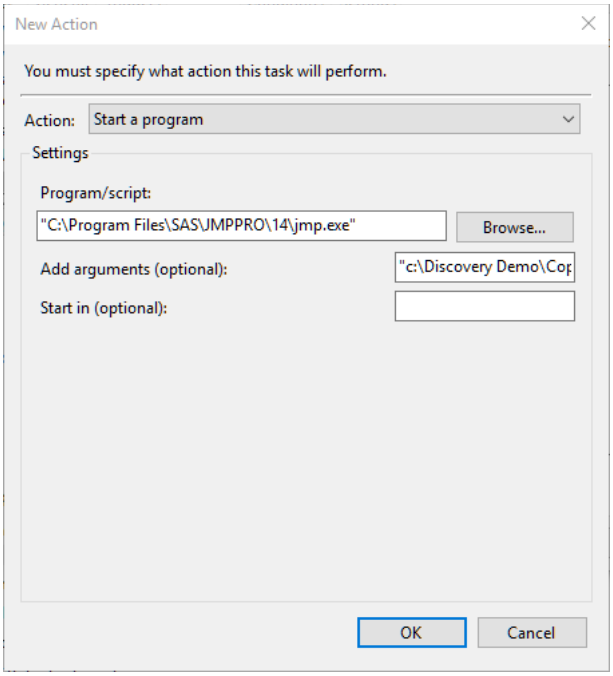
Now we can proceed to the Triggers pane and select **New Trigger…** This is pretty self-explanatory. You can select if you want to run the task once, daily, weekly or monthly and you can supply the initial start date and time. For **Advanced settings** I usually specify "**Stop task if it runs longer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



Task Trigger Completed

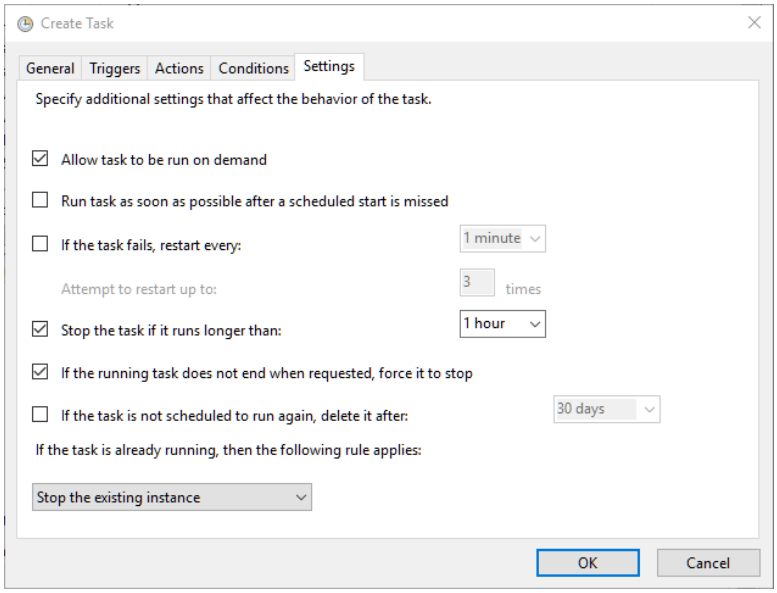
Once we have set up the timing of the task, we actually get to specify what task we are trying to accomplish. We can now select the **Actions** pane, making sure that "**Start a program"** is selected next

to **Action:**. Select the "**Browse…"** button to navigate to the JMP executable in "c:\program files\sas\jmp\14". 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\Copenhagen\WorkingScript.jsl". Now we can select **OK.**



Action Panel Completed

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.



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*"