MATLAB Production Server Interface for Power BI® software

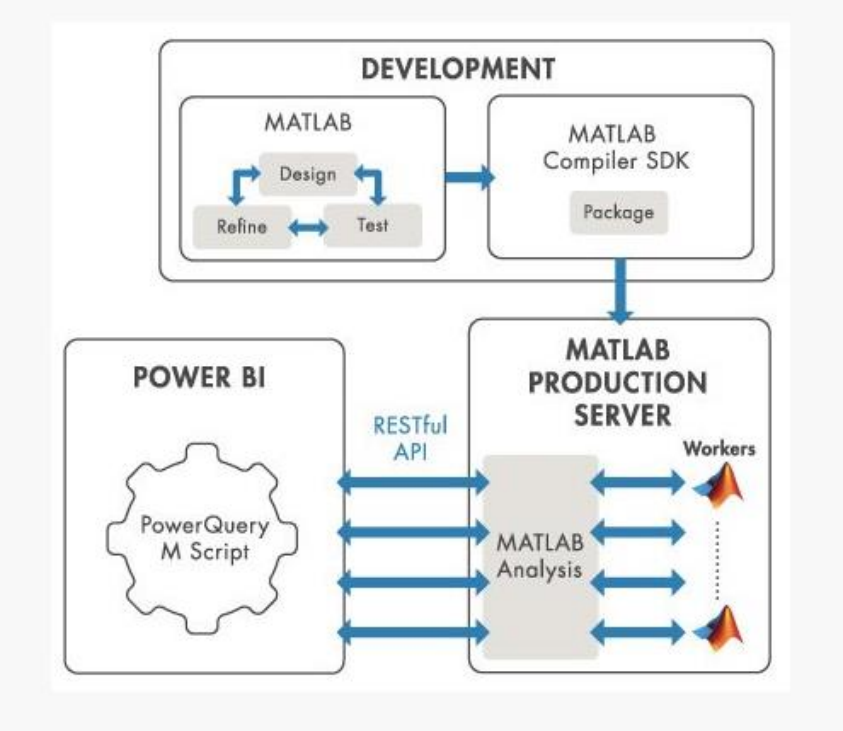
Reference Architecture

Contents

Introduction

This document provides an overview of the configuration and use of the MATLAB Production Server Interface for Power BI software. The interface is a lightweight Power BI custom data connector that enables the integration of MATLAB algorithms with Power BI visualizations. The custom data connector is authored in M Query language.

Architecture Diagram



The MATLAB Production Server Interface for Power BI enables Power BI users to directly access MATLAB algorithms using the custom data connector feature of Power BI (https://powerbi.microsoft.com/en-us/blog/data-connectors-developer-preview/).

The data connector communicates with MATLAB functions using a RESTful API, via HTTP requests. It can send data that is loaded in tables from the Power BI environment to MATLAB and can receive the results from MATLAB computations in Power BI tables. There is additional flexibility to drill down into the result set, apply transformations, and perform any other modification as required using the query editor in Power BI.

MATLAB algorithms can run either

(a) in a centralized scalable environment using MATLAB Production Server. This allows for multiple Power BI users to call the MATLAB algorithms concurrently.
(b) on a desktop machine with MATLAB Compiler SDK installed using the 'Test Client' feature. This allows the Power BI desktop application to make calls to MATLAB function on the same machine.

Description

The example used for this demo will allow Power BI users to call a MATLAB application and analyze cyclical data using a fast Fourier transform algorithm. Fourier transformations allow users to analyze variations in data, such as an event in nature over a period of time. The data retrieved here represents the number and size of sunspots for the last 300 years, using the Zurich sunspot relative number. The data retrieved from MATLAB can be plotted in Power BI to answer questions such as the frequency of peak sunspot activity, the power variation over the years etc.

The function used in this MATLAB application to perform Fourier transformation is 'fft', which has a lower computational cost when compared to other direct implementations. By integrating the MATLAB analysis with Power BI, it is possible to provide Power BI users direct access to powerful analyzing capabilities in MATLAB.

More information about this example is documented below:
https://www.mathworks.com/examples/matlab/mw/matlab_featured-ex37594814-analyzing-cyclical-data-with-fft

The Power BI file for this example is under 'Examples\Sunspots\PowerBI' path in the package provided. The MATLAB application and project files are in 'Examples\Sunspots\MATLAB'.

Installation and Configuration

System Requirements:

MathWorks Products

1. MATLAB (R2016b or later)
2. MATLAB Compiler SDK (R2016b or later)
3. MATLAB Production Server (R2016b or later) – *required if deploying MATLAB applications to an enterprise environment*

Microsoft Products

1. Microsoft Power BI Desktop (version 2.47 or later)

Getting Started

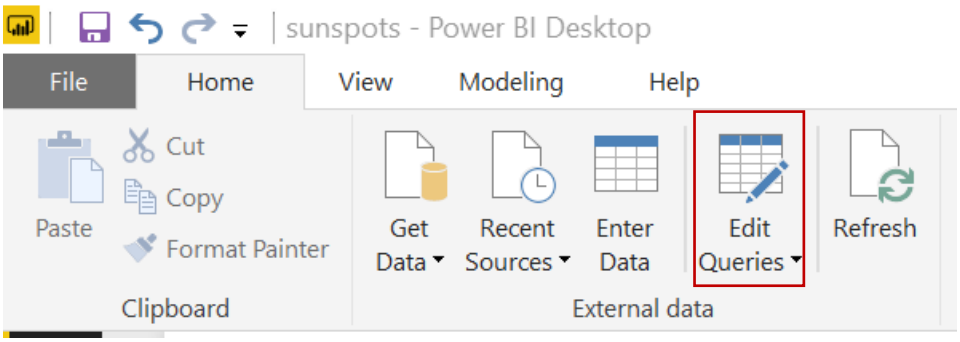
Power BI environment setup

In Power BI Desktop software, click on File->Options and Settings->Options and ensure that the below settings are checked:

(a) Under 'Privacy' select the 'Always ignore Privacy Level settings'.
(b) Under 'Preview Features' check and enable the 'Custom data connectors'.

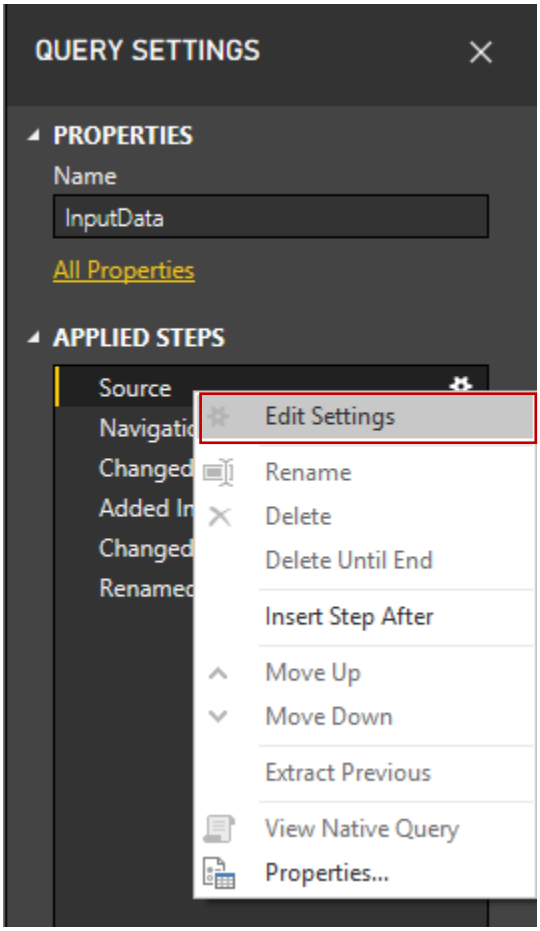
Note: Some versions of Power BI may require permissions to be set by the user to allow data extensions to load without validation or warning under the 'Security' option.

The Power BI file 'Sunspots.pbix' included in this package contains a table named 'InputData' This table has two columns – Years and Index. Open the 'Sunspots.pbix' file in Power BI and click on Edit Queries.

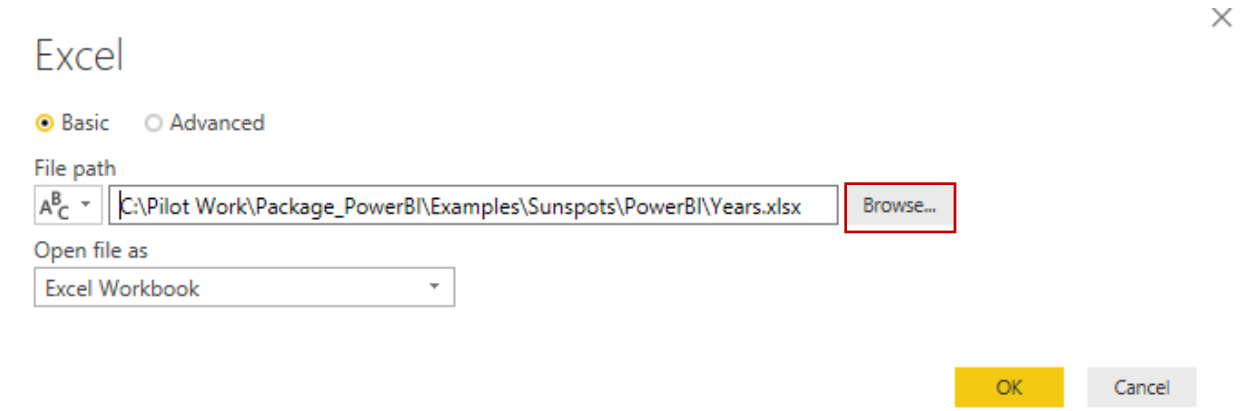


Please note that the initial data set is loaded into Power BI from an Excel workbook – Years.xlsx which is included in the package provided. The path to this Excel file may need to be set after the package is downloaded.

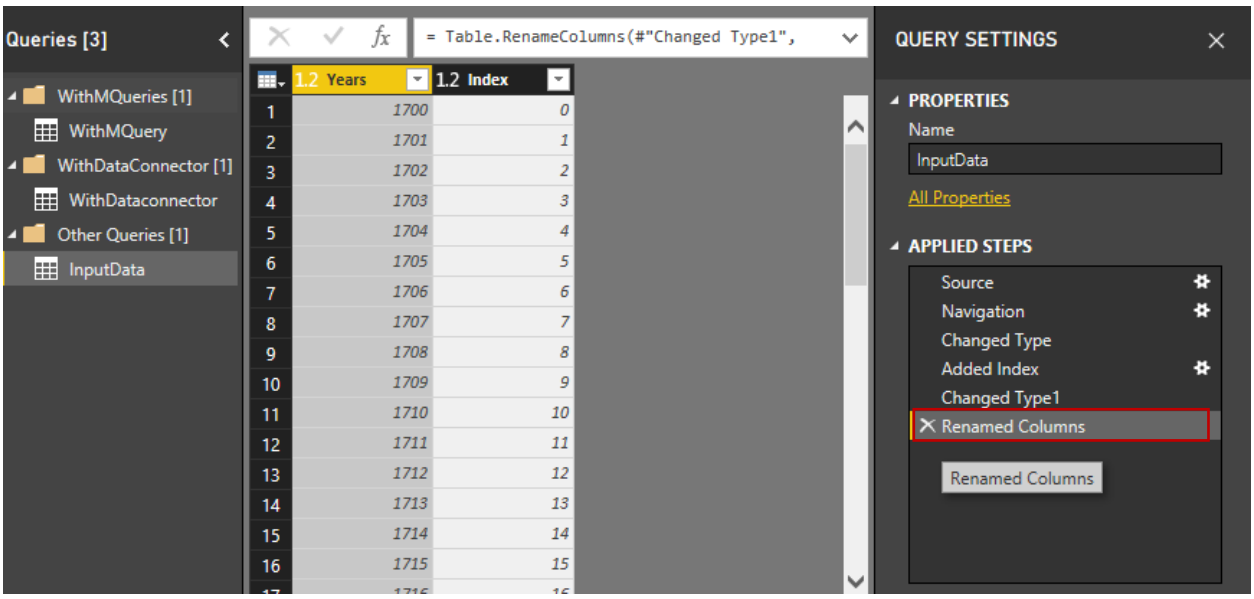
To do this, find the 'InputData' table listed on the left-hand pane under 'Other Queries'. The 'Query Settings' pane appears on the right. The transformations applied to the input data appears as a set of steps under 'Applied Steps'. Locate and right-click on 'Source', then click on 'Edit Settings'.



This will bring up the window to choose the path to the data. Click on 'Browse' and select the file Years.xlsx available in the package downloaded.



Now click on the 'Renamed Columns' step in the 'Query Settings' pane to view the data as below:



The Years column is provided as input to the MATLAB application. If the user needs to send only a subset of the values in this table, then Power BI functionality to filter the data or remove rows can be applied to create the subset. Once the user has the required data, click on 'Close and Apply' button on the top left of the screen and return to the main Power BI window.

Custom Data Connector setup

The steps involved in installation and configuration of the custom data connector are:

A) Deploy the custom data connector file (.mez file)

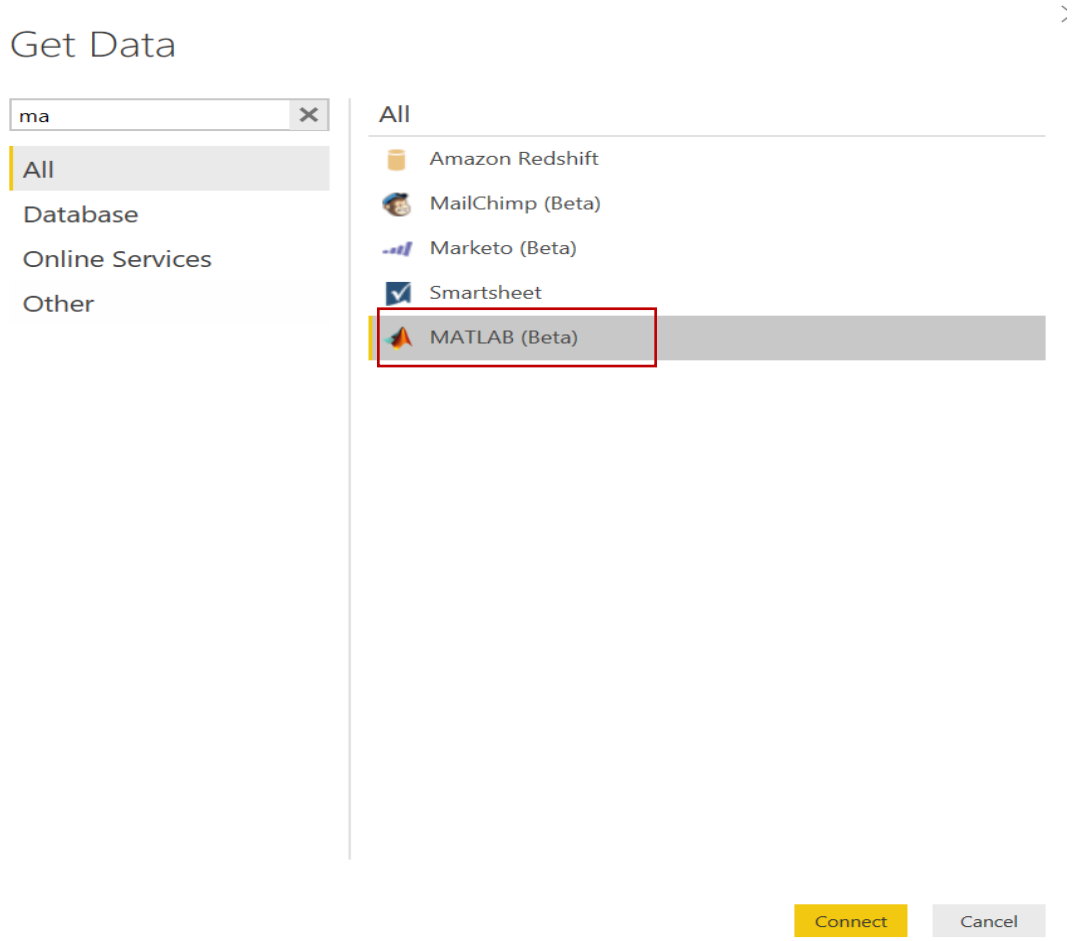
Custom data connectors in Power BI are extensions that aid in fetching data from external sources and services. They are created with the M query language and packaged into files with a '.mez' extension. To load the connector into Power BI, the '.mez' file needs to be placed in the folder 'Documents\Microsoft Power BI Desktop\Custom Connectors'. Please note that this path may not be available by default in most

Windows machines. If not available, the user needs to create the folder structure as shown in the above path.

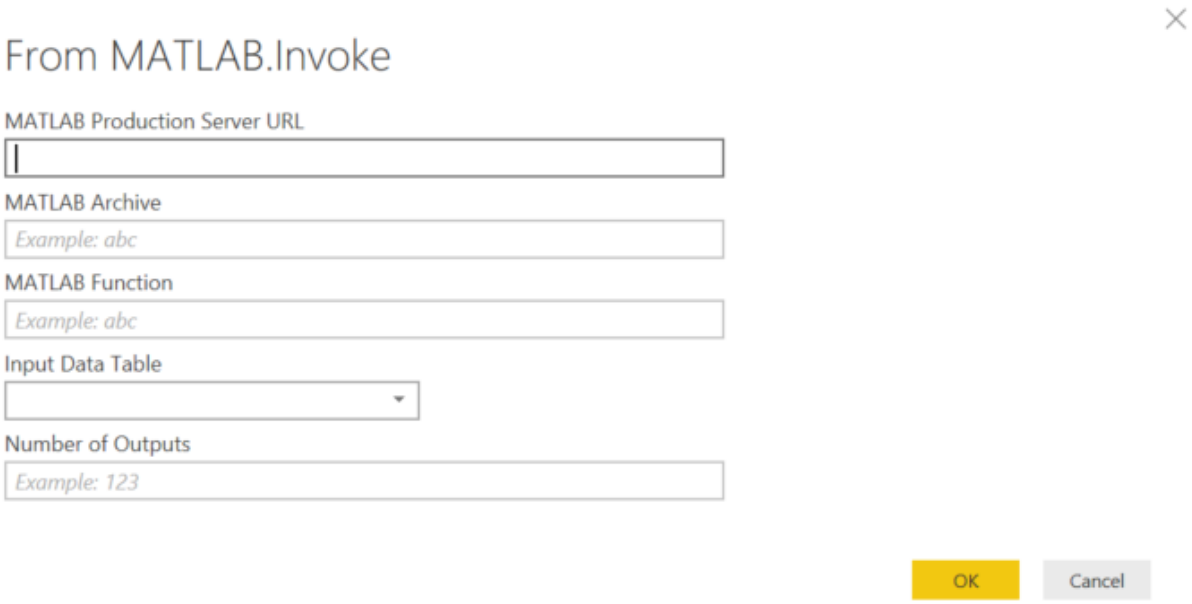
The data connector required for integrating MATLAB with Power BI has been created by MathWorks and is available in the package provided as 'MATLAB.mez'. Place this file in the '*Documents**Microsoft Power BI Desktop**Custom Connectors*' folder. Restart Power BI.

B) Setup the data connector

To set up the custom data connector so it can call MATLAB algorithms, click on the Get Data button in Power BI. This will bring up the option to connect to different data sources. You can scroll down the list to find 'MATLAB(Beta)' or type 'MATLAB' in the search area.



Select the connector and click on Connect. This will bring up a dialog box to enter information to connect to MATLAB as below:



- The MATLAB Production Server URL is the server address and port number and is of the format http://<Server Address>:<Port Number>.
  For example, if MATLAB Production Server is running locally, or if using the 'Test Environment' in MATLAB Compiler SDK, the default port number is 9910. The URL should then be http://localhost:9910.

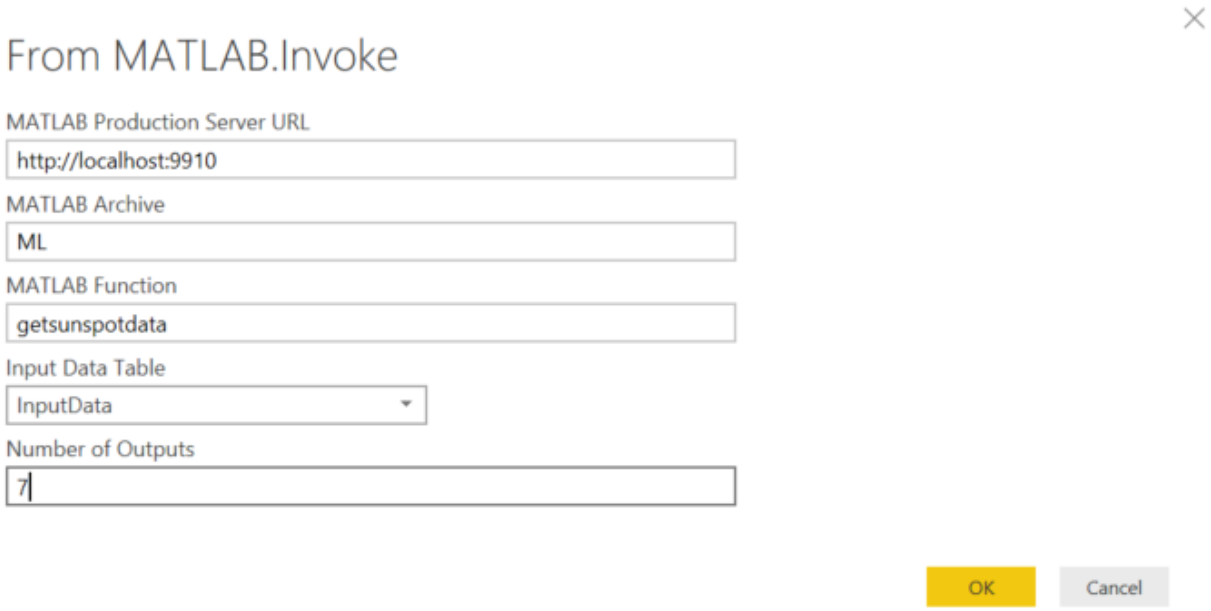
- MATLAB Archive is the name of the packaged MATLAB analytics created using the MATLAB Compiler SDK. The package has an extension '.ctf'. Enter the name of the archive file without the extension. The archive name for this example is 'ML'.

- MATLAB Function is the name of the MATLAB function packaged in the MATLAB archive. The function name for this example is 'getsunspotdata'.

- Input Data Table is the Power BI table containing the input arguments for the MATLAB function call. This is a drop-down list containing a list of all available tables in the Power BI environment. The table to select for this example is 'InputData'.

- Number of Outputs is the number of output parameters expected in the result set from the MATLAB function. The number of outputs expected by this example is 7.

The data connector configuration screen with the correct values should look like the screenshot shown below:



Click the 'OK' button to save the configuration. The MATLAB data connector is now setup to call MATLAB analytics.

Note that once the OK button is clicked, Power BI will open the Query Editor and execute a query named 'Query1' that will contain the code to invoke the data connector. This call will not be successful until the MATLAB environment is also set up.

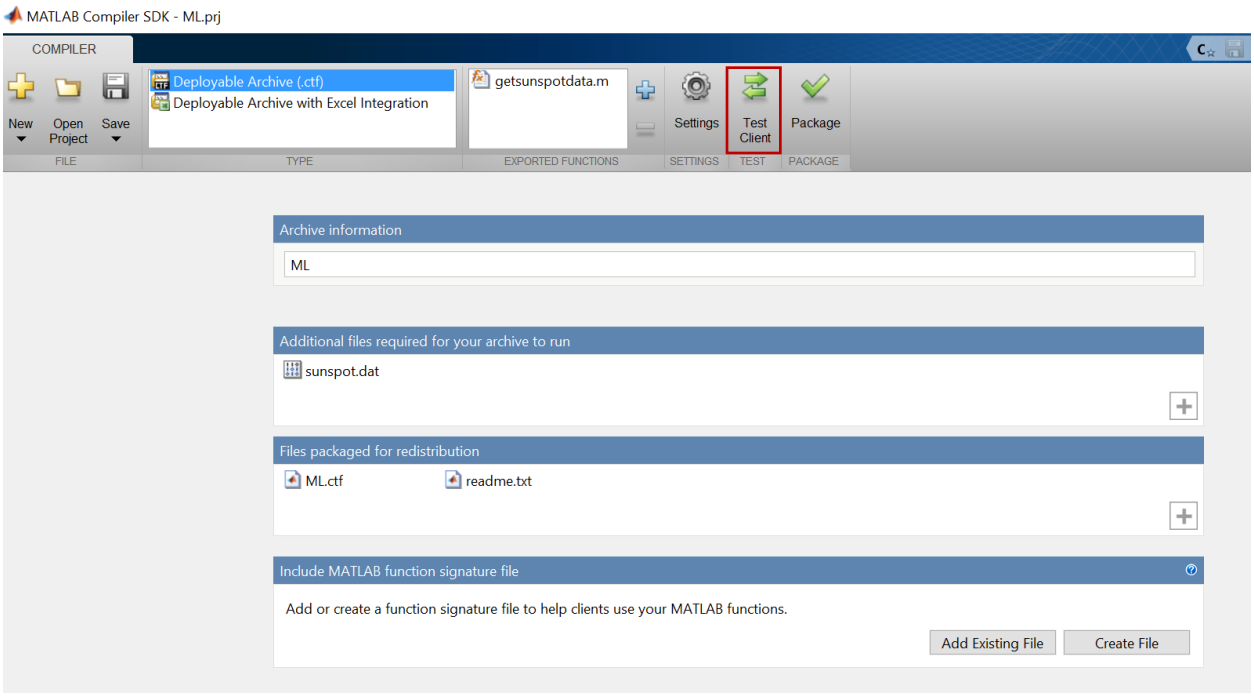
To proceed, it is safe to right click on the newly created query and delete it. The complete M query required to call the MATLAB application using the custom data connector is provided in Appendix A and is also included in the 'Sunspots.pbix' file. The execution of this script to retrieve results from MATLAB is described in the Usage section.

MATLAB environment setup

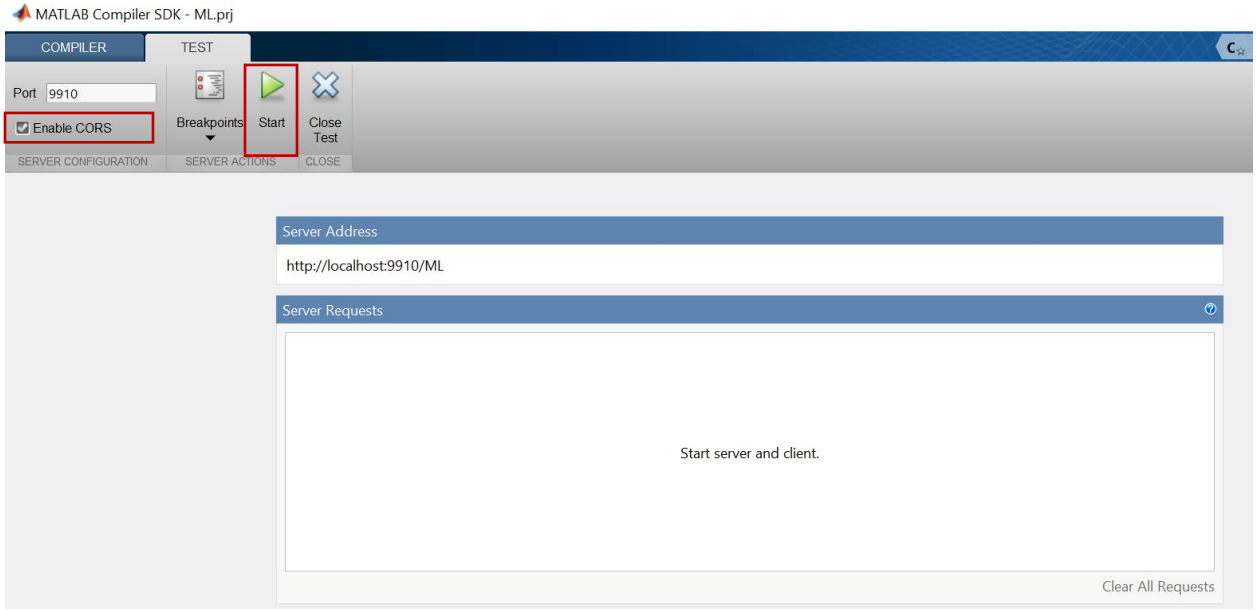
Setting up the MATLAB environment involves changing the MATLAB code so that it is ready to handle the input parameters from Power BI, as well as quickly starting a test server locally with MATLAB Compiler SDK.

To load the MATLAB application for this example, locate the MATLAB folder in this package which contains the 'getsunspotdata.m' file and 'ML.prj' project file. Follow below steps to quickly set up a test environment using MATLAB Compiler SDK:

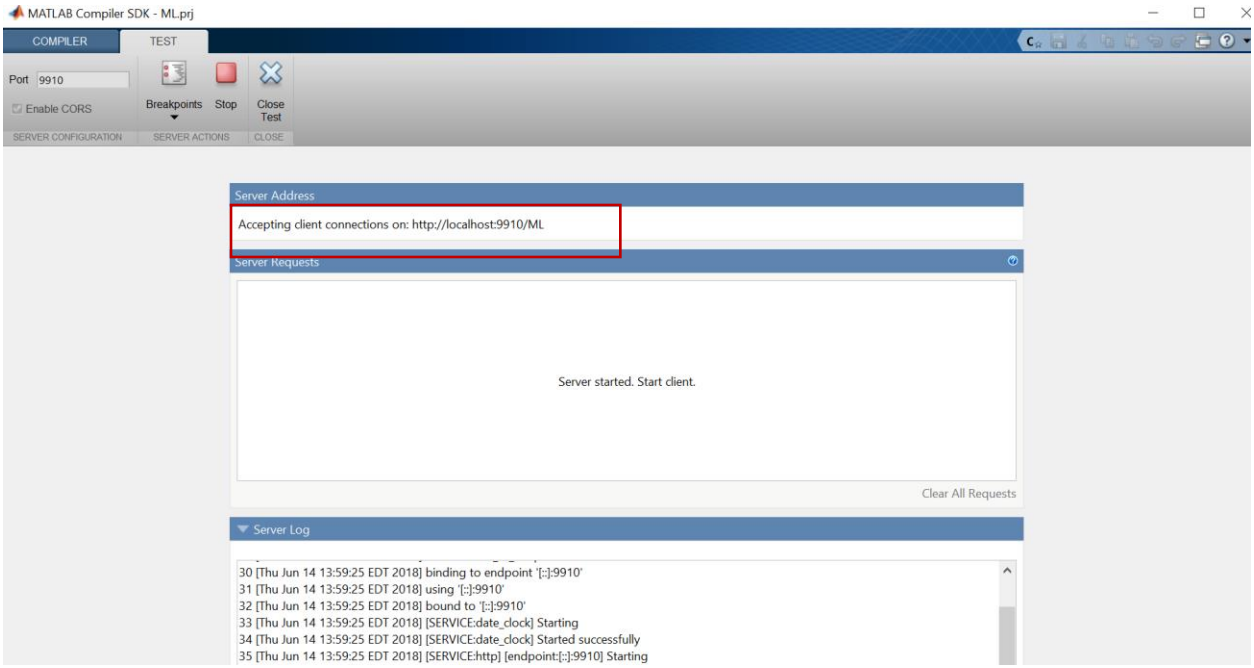
1) Open the 'Ml.prj' file to display the window shown below.



2) Click on the 'Test Client' button highlighted above.
3) This should open the 'Test Environment' window.



4) Ensure that the 'Enable CORS' checkbox highlighted above is checked and then click the start button. This will enable your MATLAB session to accept HTTP requests from client applications in the same machine.



Additional Considerations

MATLAB Function Signature Modification

This change is only required if you are testing the integration with a MATLAB application not included in this package. Since the data in Power BI is sent to MATLAB in a single table, the MATLAB application needs to be able accept a variable number of input arguments. This is done by modifying the MATLAB function signature to accept 'varargin'. More information about variable length input argument list is in the link: https://www.mathworks.com/help/matlab/ref/varargin.html

Please note that no further changes are necessary for the sunspots example included in this package as it is already modified to enable it to accept any number of input arguments from Power BI.

An example of the change to be made is given below:
If the MATLAB function signature is

```
function output = MPSFunction(A,B,C)
```

then modify the function signature to be

```
Function output = MPSFunction(varargin)
```

You can then index into the arguments with:

```
inputParams =  struct2cell(varargin{1});
```

Here inputParams is a cell. The arguments A, B and C can be accessed using standard MATLAB operations. For example,

```
A = inputParams{1}; B = inputParams{2} ; C = inputParams{3}
```

Deploy MATLAB package to MATLAB Production Server or use MATLAB Compiler SDK

Before deploying to MATLAB Production Server, the MATLAB developer can use the 'Test client' feature to test the integration with Power BI in the local development machine. More information about using the 'Test Client' feature is documented below:
https://www.mathworks.com/help/compiler_sdk/mps_dev_test/test-in-process.html

To deploy a MATLAB application to MATLAB Production Server, the MATLAB code files and any required data files need to be compiled using MATLAB Compiler SDK. This is a necessary step and the complete instructions for compiling and creating project files are in the link below:

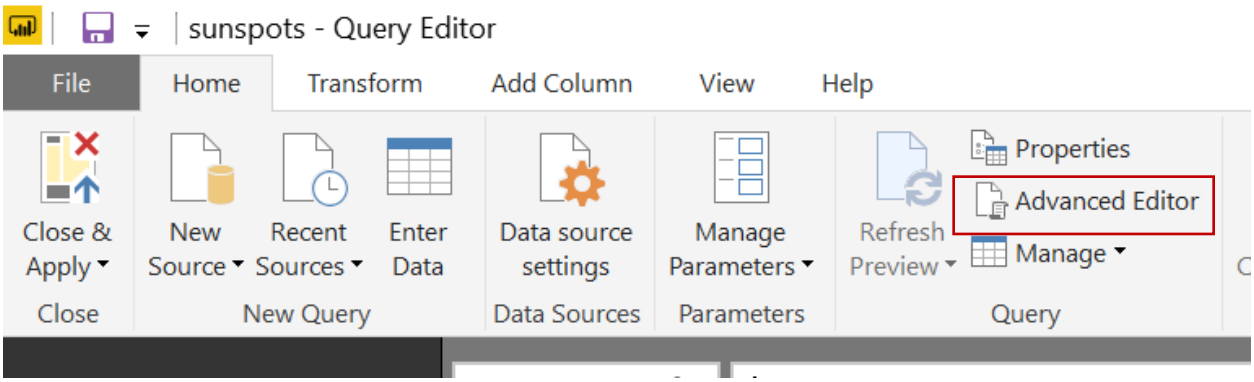
https://www.mathworks.com/help/mps/deployable-archive-creation.html

Please note that if using the sunspot example included in this package, you just need to open the 'ML.prj' file to use the test environment provided by MATLAB Compiler SDK.

Usage

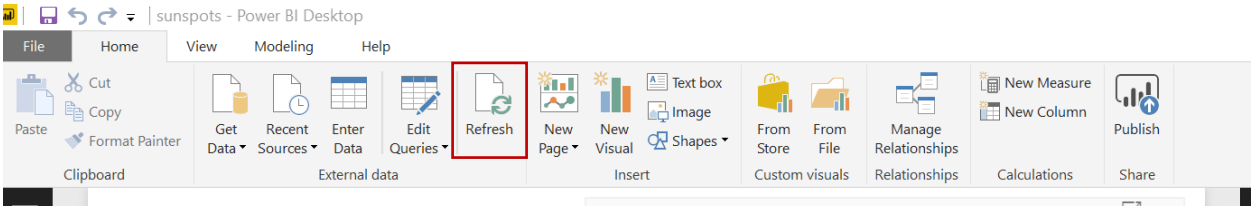
Once the packaged MATLAB application is hosted on MATLAB Production Server or on the testing environment, Power BI can use the data connector to make RESTful calls to the MATLAB functions.
To invoke the MATLAB analysis from Power BI, follow below steps:

1) In the Power BI Desktop UI, click on 'Edit Queries' to open the query window.
2) On the left-hand pane under Queries, right click and select New Query -> Blank Query. A new query named 'Query1' will be formed.
3) Click on Query1 and then 'Advanced Editor' highlighted below:



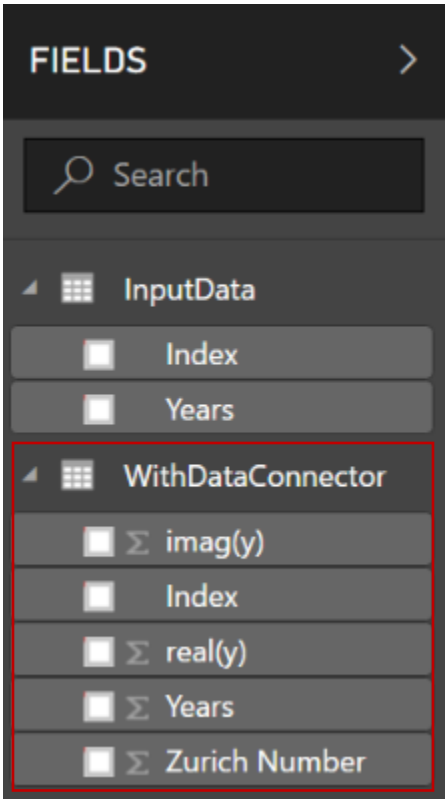
4) In the window for the Query, paste the M code in Appendix A.
5) Right click on 'Query1' and rename the query to 'WithDataConnector'.
6) Click 'Close & Apply' button on the top left of the screen and return to the main Power BI window.

7) Click on the ‘Refresh’ button in the menu:



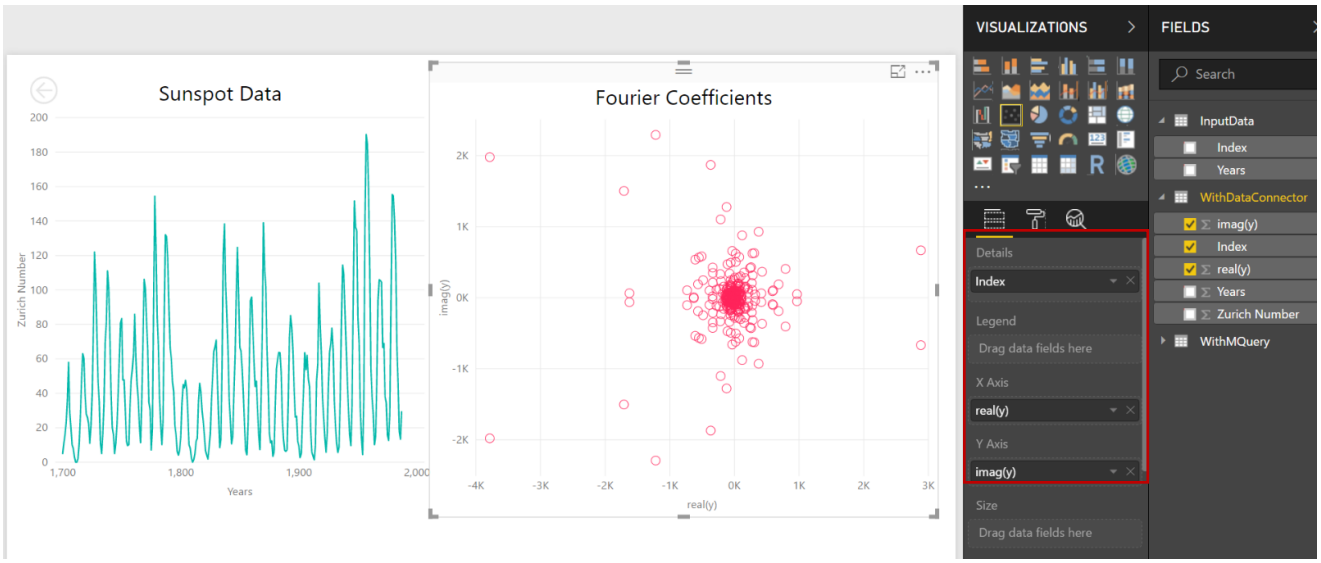
This will initiate an HTTP call to MATLAB and populate the tables as defined in the M-code in ‘WithDataConnector’.

Expand the ‘WithDataConnector’ under fields to view the columns that are populated:



Once data is available in Power BI, it is easy to create the Line chart and Scatter plot in Power BI. The ‘Sunspot.pbix’ example included in the package already has the plot configured.

The columns used in the X and Y axis, and the column for details for plotting the Fourier Coefficients scatter plot is shown below:



The columns used to plot the line chart displaying the sunspot Zurich numbers is shown below:

Making multiple calls to MATLAB

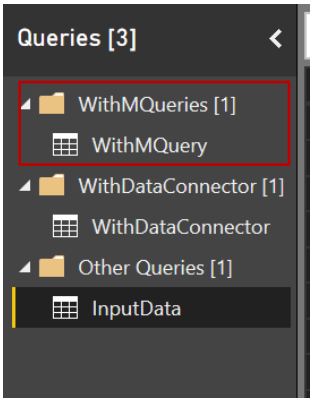
To make function calls to MATLAB with different input parameters, the Power BI user needs to filter or apply transformations on the table selected as the input table in the custom data connector configuration settings. For this example, the 'InputData' table can be modified in the query window in Power BI to select only the rows containing the data to be sent to MATLAB.

The Query Editor provides tools to filter, remove and add data rows and columns. Once the table contains the required input parameters, click 'Refresh' on the Power BI screen to call MATLAB algorithms.

Limitations

The custom data connector provided by Microsoft is still in preview mode as of July 2018. As a result, publishing Power BI sheets to use online is not a supported feature. However, the M queries that are used to connect to MATLAB are available for you to use directly within Power BI. Using the M queries directly avoids the usage of a custom data connector; therefore, you will be able to publish Power BI sheets online. An example of a call to MATLAB using just the M query is included in the example provided with this package.

To view the complete M query used to connect to MATLAB, click on 'Edit Queries'. The 'WithMQueries' group highlighted below contains the results without using custom data connector. You can modify the behavior using the 'Advanced Editor' for this query. This code is also available in Appendix B.



Summary

This guide provided the steps required to integrate MATLAB analytics with Power BI. The integration depends on the availability of the custom data connector feature which is currently in preview mode. For additional information and questions, please contact MathWorks.

Contact Information

Please contact mwlab@mathworks.com for issues/questions.

Appendix A: M-code using Custom Data Connector

```
let

//Call MATLAB using the custom data connector

Source = MATLAB.Invoke("http://localhost:9910", "ML", "getsunspotdata", InputData, 7),

//Retrieve output arguments

   lhs = Source[lhs],

   lhs1 = lhs{1},
   zurichnumbers = lhs1[mwdata],

   lhs2 = lhs{2},
   relNums = lhs2[mwdata],

   lhs3 = lhs{3},
   imagNums= lhs3[mwdata],

   lhs4 = lhs{4},
   freqNums= lhs4[mwdata],

   lhs5 = lhs{5},
   powerNums= lhs5[mwdata],

   lhs6 = lhs{6},
   periodNums= lhs6[mwdata],
//Create tables, and merge into single table
   #"zurichTable" = Table.FromList(zurichnumbers, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
   #"zurichtype" = Table.TransformColumnTypes(#"zurichTable",{{"Column1", type number}}),
   #"zurichIndex" = Table.AddIndexColumn(#"zurichtype", "index", 0, 1),

   #"relTable" = Table.FromList(relNums , Splitter.SplitByNothing(), null, null, ExtraValues.Error),
   #"relType" = Table.TransformColumnTypes(#"relTable",{{"Column1", type number}}),
   #"relIndex" = Table.AddIndexColumn(#"relType", "index", 0, 1),

   #"imagTable" = Table.FromList(imagNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
   #"imagType" = Table.TransformColumnTypes(#"imagTable",{{"Column1", type number}}),
   #"imagIndex" = Table.AddIndexColumn(#"imagType", "index", 0, 1),

   #"freqTable" = Table.FromList(freqNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
   #"freqType" = Table.TransformColumnTypes(#"freqTable",{{"Column1", type number}}),
   #"freqIndex" = Table.AddIndexColumn(#"freqType", "index", 0, 1),

   #"powerTable" = Table.FromList(powerNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
   #"powerType" = Table.TransformColumnTypes(#"powerTable",{{"Column1", type number}}),
   #"powerIndex" = Table.AddIndexColumn(#"powerType", "index", 0, 1),

   #"periodTable" = Table.FromList(periodNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
   #"periodType" = Table.TransformColumnTypes(#"periodTable",{{"Column1", type number}}),
   #"periodIndex" = Table.AddIndexColumn(#"periodType", "index", 0, 1),
```

```
    #"ZuricRelMerged" = Table.NestedJoin(#"zurichIndex",{"index"},relIndex,{"index"},"relIndex",JoinKind.Inner),
    #"ExpandedZuricRelMerged" = Table.ExpandTableColumn(#"ZuricRelMerged", "relIndex", {"Column1"}, {"real(y)"}),

    #"ImagZuricRelMerged" = Table.NestedJoin(#"ExpandedZuricRelMerged",{"index"},imagIndex,{"index"},"imagIndex",JoinKind.Inner),
    #"ExpandedImagZuricRelMerged" = Table.ExpandTableColumn(ImagZuricRelMerged, "imagIndex", {"Column1"}, {"imag(y)"}),

    #"FreqImagZuricRelMerged" =
Table.NestedJoin(#"ExpandedImagZuricRelMerged",{"index"},freqIndex,{"index"},"freqIndex",JoinKind.LeftOuter),
    #"ExpandedFreqImagZuricRelMerged" = Table.ExpandTableColumn(FreqImagZuricRelMerged, "freqIndex", {"Column1"},
{"frequency"}),

    #"PowerFreqImagZuricRelMerged" =
Table.NestedJoin(#"ExpandedFreqImagZuricRelMerged",{"index"},powerIndex,{"index"},"powerIndex",JoinKind.LeftOuter),
    #"ExpandedPowerFreqImagZuricRelMerged" = Table.ExpandTableColumn(PowerFreqImagZuricRelMerged, "powerIndex",
{"Column1"}, {"power"}),

    #"AllTablesMerged" =
Table.NestedJoin(#"ExpandedPowerFreqImagZuricRelMerged",{"index"},periodIndex,{"index"},"periodIndex",JoinKind.LeftOuter),
    #"AllTablesMergedExpanded" = Table.ExpandTableColumn(AllTablesMerged, "periodIndex", {"Column1"}, {"period"}),

    #"Merged Queries" = Table.NestedJoin(#"AllTablesMergedExpanded",{"index"},InputData,{"Index"},"Sheet1",JoinKind.LeftOuter),
    #"Expanded Sheet1" = Table.ExpandTableColumn(#"Merged Queries", "Sheet1", {"Years"}, {"Years"}),
    #"Renamed Columns" = Table.RenameColumns(#"Expanded Sheet1",{{"Column1", "zurich number"}, {"Years", "years"}})
in
    #"Renamed Columns"
```

Appendix B: M-code without using Custom Data Connector

```
let

    URLTOCall = "http://localhost:9910/ML/getsunspotdata",
    inputParams = Text.FromBinary(Json.FromValue(InputData)),
    body = Text.Combine({"{""nargout"":",Text.From(7),","rhs"": ",inputParams,"}"}),
    Source= Json.Document(Web.Contents(URLTOCall, [Headers=[#"Content-Type"="application/json"],Content=Text.ToBinary(body)
])),
//Retrieve output from MATLAB
    lhs = Source[lhs],

    lhs1 = lhs{1},
    zurichnumbers = lhs1[mwdata],

    lhs2 = lhs{2},
    relNums = lhs2[mwdata],

    lhs3 = lhs{3},
    imagNums= lhs3[mwdata],

    lhs4 = lhs{4},
    freqNums= lhs4[mwdata],

    lhs5 = lhs{5},
    powerNums= lhs5[mwdata],

    lhs6 = lhs{6},
    periodNums= lhs6[mwdata],

//Create tables from output and merge into single table.
    #"zurichTable" = Table.FromList(zurichnumbers, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"zurichtype" = Table.TransformColumnTypes(#"zurichTable",{{"Column1", type number}}),
    #"zurichIndex" = Table.AddIndexColumn(#"zurichtype", "index", 0, 1),

    #"relTable" = Table.FromList(relNums , Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"relType" = Table.TransformColumnTypes(#"relTable",{{"Column1", type number}}),
    #"relIndex" = Table.AddIndexColumn(#"relType", "index", 0, 1),

    #"imagTable" = Table.FromList(imagNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"imagType" = Table.TransformColumnTypes(#"imagTable",{{"Column1", type number}}),
    #"imagIndex" = Table.AddIndexColumn(#"imagType", "index", 0, 1),

    #"freqTable" = Table.FromList(freqNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"freqType" = Table.TransformColumnTypes(#"freqTable",{{"Column1", type number}}),
    #"freqIndex" = Table.AddIndexColumn(#"freqType", "index", 0, 1),

    #"powerTable" = Table.FromList(powerNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"powerType" = Table.TransformColumnTypes(#"powerTable",{{"Column1", type number}}),
    #"powerIndex" = Table.AddIndexColumn(#"powerType", "index", 0, 1),

    #"periodTable" = Table.FromList(periodNums, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    #"periodType" = Table.TransformColumnTypes(#"periodTable",{{"Column1", type number}}),
    #"periodIndex" = Table.AddIndexColumn(#"periodType", "index", 0, 1),
```

```
    #"ZuricRelMerged" = Table.NestedJoin(#"zurichIndex",{"index"},relIndex,{"index"},"relIndex",JoinKind.Inner),
    #"ExpandedZuricRelMerged" = Table.ExpandTableColumn(#"ZuricRelMerged", "relIndex", {"Column1"}, {"real(y)"}),

    #"ImagZuricRelMerged" = Table.NestedJoin(#"ExpandedZuricRelMerged",{"index"},imagIndex,{"index"},"imagIndex",JoinKind.Inner),
    #"ExpandedImagZuricRelMerged" = Table.ExpandTableColumn(ImagZuricRelMerged, "imagIndex", {"Column1"}, {"imag(y)"}),

    #"FreqImagZuricRelMerged" =
Table.NestedJoin(#"ExpandedImagZuricRelMerged",{"index"},freqIndex,{"index"},"freqIndex",JoinKind.LeftOuter),
    #"ExpandedFreqImagZuricRelMerged" = Table.ExpandTableColumn(FreqImagZuricRelMerged, "freqIndex", {"Column1"},
{"frequency"}),

    #"PowerFreqImagZuricRelMerged" =
Table.NestedJoin(#"ExpandedFreqImagZuricRelMerged",{"index"},powerIndex,{"index"},"powerIndex",JoinKind.LeftOuter),
    #"ExpandedPowerFreqImagZuricRelMerged" = Table.ExpandTableColumn(PowerFreqImagZuricRelMerged, "powerIndex",
{"Column1"}, {"power"}),

    #"AllTablesMerged" =
Table.NestedJoin(#"ExpandedPowerFreqImagZuricRelMerged",{"index"},periodIndex,{"index"},"periodIndex",JoinKind.LeftOuter),
    #"AllTablesMergedExpanded" = Table.ExpandTableColumn(AllTablesMerged, "periodIndex", {"Column1"}, {"period"}),

    #"Merged Queries" = Table.NestedJoin(#"AllTablesMergedExpanded",{"index"},InputData,{"Index"},"Sheet1",JoinKind.LeftOuter),
    #"Expanded Sheet1" = Table.ExpandTableColumn(#"Merged Queries", "Sheet1", {"Years"}, {"Years"}),
    #"Renamed Columns" = Table.RenameColumns(#"Expanded Sheet1",{{"Column1", "zurich number"}, {"Years", "years"}})
in
    #"Renamed Columns"
```