JSON Grapher Manual

The concept:

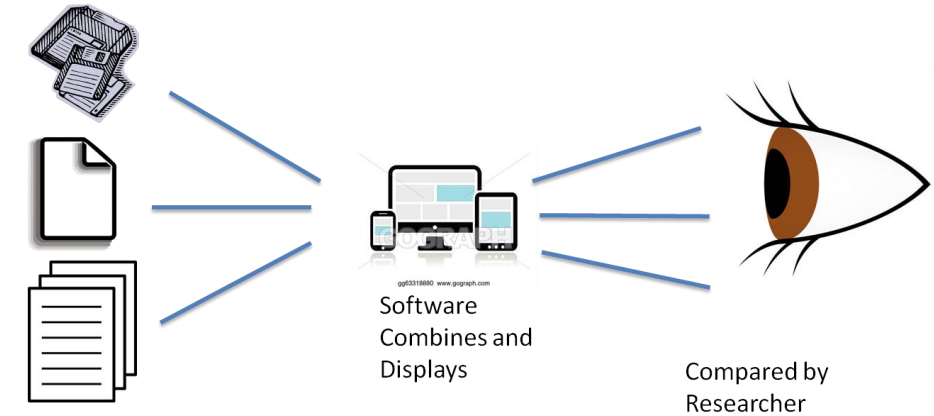

Table of Contents

Introduction:

JSON Grapher is a webapp for plotting X,Y data or model outputs from multiple sources (multiple files) onto a single graph. It is a step towards an "experiential economy" for comparing data from various sources. The required file type is a simple format that can even be made "by hand" as a .csv file.

When two datasets have different units (such as kg/min and g/hour), JSON Grapher will *automatically* converts the data between units and plot the data together! This can save researchers lots of time when trying to compare data from different studies. This concept is depicted in the below image.

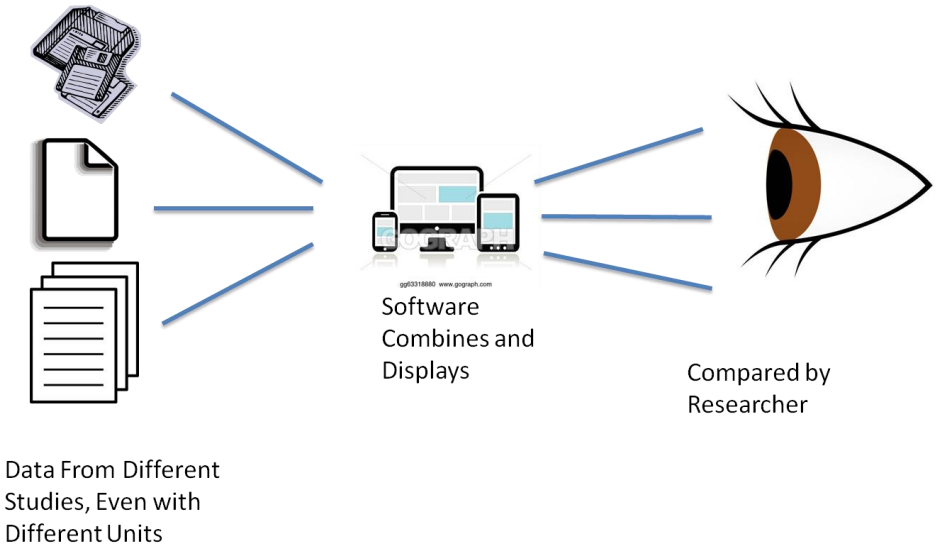

Figure 1: JSONGrapher can take data from different studies, even with different units, then combine them into one graph for direct comparison by researchers. JSON Grapher can also run models and plot the outputs alongside measured or existing data.

Example Demonstration
To see how JSON Grapher works, watch the demonstration video.

Now you can try it!

Download DemonstrationFiles.zip (click to use link)

Unzip/Copy the files to folder on your computer. The demonstration will not work if you leave those files inside the zip file.

Open www.jsongrapher.com and upload/drag the following three files one at a time:
amino_silane_silica_exp_343.csv
amino_silane_silica_exp_383.csv
CO2AdsorptionNaX2.json

Note how JSON Grapher allows comparing all three data sets. Importantly, the third data set actually has different units of Pa rather than kPa! JSON Grapher automatically converts the units to match those of the first uploaded dataset and plots all of the data together!

Now you know how to use JSON Grapher in a basic way!

1. Further Explanation of Data and Model Files and their usage with JSONGrapher
JSON Grapher can plot several types of data:

Single Series X,Y data (from .json or .csv)
Multiple series XYYY data (from .json or .csv)
Multiple series XY XY XY data (only from .json)
X,Y Data from Analytical / Numerical Simulation Models (only from .json)

Note: Model files do not have data in them. Model files only contain parameters and point to an external simulation function. JSON Grapher will call the external simulation function and then plot the outputs.

All of these types of examples are inside the JSONGrapherExamples github repository , inside of the zipfiles ExampleDataRecords.zip and ExampleModelRecords.zip (a list of which files correspond to the above data types is included further below).

Currently, JSONGrapher supports three formats: ".JSON", ".csv", and ".tsv". The file format will be recognized as long as the period and the letters are anywhere in the filename. The csv and tsv files are comma separated and tab separated, respectively. Tab separated files are advantageous as an option because they allow the inclusion of commas in series names, and some molecule names have commas.

For all file formats, the units must be provided within the x label and for the y label using parentheses (see example files).

All of the example files (JSON, CSV, TSV) can be used with www.JSONGrapher.com by uploading/dragging. Files must be added into JSONgrapher one file at a time. By adding data/models which are of the same type, they will be plotted together. Data which was collected with different units will have automatic unit conversion in order to plot the data together. For example, multiple CO_2 adsorption isotherms can be plotted together (whether from data records or from a model). Different data types (such as adsorption isotherms and infrared spectra) will not be plotted together, and instead an error message will be displayed.

Example files for the various types from zipfiles ExampleDataRecords.zip and ExampleModelRecords.zip:

- Single Series X,Y data (from .json or .csv)
  - 1-..\amino_silane_silica_exp_343.csv
  - 1-..\CO2AdsorptionNaX2.json
  - 1-..\CO2AdsorptionNaX2.tsv.txt
- Multiple series XYYY data (from .json or .csv)
  - 1-..\CO2Adsorption_NaX_and_CaX_two_series.json
  - 1-..\CO2Adsorption_NaX_and_CaX_two_series_csv.csv
  - 2-..\La_Perovskites_Combined.json
  - 2-..\Sr_Perovskites_Combined.json
  - 4-..\PtAtomPMFOnTiO2Rutile110.csv
- Multiple series XY XY XY data (only from .json)
  - 1-..\CO2Adsorption_NaX_and_CaX_two_series.json
  - 3-..\DRIFTS_CO_Adsorption_onAu22.csv
- X,Y Data from Analytical / Numerical Simulation Models (only from .json)
  - Files in ExampleModelRecords\1_CO2_Adsorption_Isotherms\
  - Examples are provided for single as well as multiple series in the same file.

It is easiest to use an existing example file for understanding how to use the file format, but an explanation of the fields of the file formats is provided below.

2. Explanation of Fields in CSV Data Records Format (and for TSV)

While JSON Data Files are the "authoritative" format for JSONGrapher, the .csv file format is more accessible for many users and has fewer fields. Accordingly, an explanation of the CSV Data Records format is provided first.

A CSV file has comma separation in the data series and seriesnames, while a TSV file has tab separation in the data series and seriesnames.

Currently, the CSV file has the following fields at the top which **must** be in this row order.

comments:	Any string (including symbols) may be put in this field, except line breaks.
DataType:	Typically a datatype name that would be shared among all data files that can be compared. Alphanumeric characters and underscores are allowed. This string is used to define the data's type, and is one of the checks for which data types are compatible. For advanced use of this feature see "Hierarchical Classification" section of manual to understand how to use that feature. This string is also used to see if there is any schema for the datatype, and in fact the user can choose to provide a URL to a schema in this field, rather than a datatype name.

Chart_label:	This field becomes the title of the plot.
x_label:	This becomes the chart x label and *must* include the x-units in parentheses. Units can be multiple, such as kg/s. SI units are expected. Custom units must be inside < > and at the beginning. For example, (<frogs>*kg/s) would be permissible.
y_label:	This becomes the chart y label and *must* include the y-units in parentheses. Units can be multiple, such as mol/s. SI units are expected. Custom units must be inside < > and at the beginning. For example, (<frogs>*kg/s) would be permissible.
series_names:	This must be a list of comma separated (for CSV) or tab separated (for TSV) . For XYYY data, this list must be the same length as the number of Y series.
custom_variables:	This row enables inclusion of JSON within the csv and is only for advanced users. A better name for this row would be "connected_variables", but the row is named custom_variables in the csv so that users can more easily recognize it as an optional field.
x_values,y_values	This row is ignored, and is included for readability of the input file.

The way the csv file parsing works is that the string of the field that precedes the ":" is ignored, the csv parsing uses the row number to know which field is being parsed. Thus, if a person were to use "x-label:" rather than "x_label:", JSONGrapher would still parse the CSV file correctly (provided that the fields are on the correct row).

Below the field headings

3. Explanation of Fields in JSON Data Records

In the JSONGrapherExamples github repository the directory BasicExample has a commented JSON file that helps to explain the fields, but looking at an example single series json record and example multiseries json record may be more useful.

The below table describes the top-level fields in the JSON, recognizing that the JSON is a nested object type. The below table and the example records should be sufficient for even advanced users, but a schema exists for the most advanced users who wish to see additional details.

"comments":	Any string (including symbols) may be put in this field, except line breaks.
"title":	The same as "DataType" from the csv fields. Typically a datatype name that would be shared among all data files that can be compared. A string. Alphanumeric characters and underscores are allowed. This string is used to define the data's type, and is one of the checks for which data types are compatible. For advanced use of this feature see "Hierarchical Classification" section of manual to understand how to use that feature. This string is also used to see if there is any schema for the datatype, and in fact the user can choose to provide a URL to a schema in this field, rather than a datatype name.
"data":	This field has the structure of [{series1},{series2}] within each series object, any optional field is allowed, but the required fields are: "name" : "series_name_string", "x": [x_value1,x_value2] , "y": [y_value1,y_value2].

	The “x” and “y” lists can have data as strings or as decimal numbers. Decimal numbers less than zero must have the zero (0.004 is okay, while .004 would give an error). within each series object, the “uid” field is an optional field for include a unique ID for the data series (such as a doi, or even simply a number used internally within a research lab).
“layout”:	This top level field has elements which include the information for the chart labeling, as well as the x axis units and the y axis units. “comments”: “string” “title”: “A string that will become the chart title.” "xaxis": {“title”:”x_label_string”} "yaxis": {“title”:”y_label_string”} the “title” field inside “xaxis” must include the x axis units in parenthesis. the “title” field inside “yaxis” must include the y axis units in parenthesis. For both the x axis and the y axis. The dimensions of units can be multiple, such as mol/s. SI units are expected. Custom units must be inside < > and at the beginning. For example, (<frogs>*kg/s) would be permissible. Note that while multiple series are allowed, there is a single xaxis title and a single yaxis title. This means that a single JSON file must have the same x units and same y units for all series in that file. To make several series with differing (but compatible) units would require making several json files, as is the case for the csv files.
(custom)	Custom fields are allowed in the JSON records, and users may use custom fields to store meta data.

4. Explanation of Fields in Model Records and how to Create / Use External Simulators

A .json model record file can create one or more series from an external simulator function. We will first look at how a single series is made using a model file and external simulator.

Inside the zipfile ExampleModelRecords , we see the following file in the subdirectory:
amino_silane_silica_LangmuirIsothermModel_343_kinetic.json

A model json record requires all of the same fields required as a regular json data record, but has the “x” and “y” fields as empty lists/arrays. Instead, for each data series to be modeled, the record has a “simulate” field, which has a “model” field within it, which links to the javascript model. The simulate json object must also include inside of it all of the parameters required for the javascript simulation function. Note that the simulate field is *per series*, so one can include multiple simulated series in a single file, and also that each simulation must return a single x,y data series since the feature is per series.

There are several requirements for the simulation function:

(1) The .js file must be hosted on github, and the direct link to the .js file must be provided.
(2) The .js file must have a function named "Simulate"
    a. The simulate function must take a single input object and a single output object.
    b. The input object received will be the entire dataseries object from the original json data record (such that, for example, input.simulate.K_eq is the contents of the K_eq field inside "simulate" of the original json record).
    c. The output object returned by the simulate function will be a nested array with fields of "x", "y" , "x_label" and "y_label". Thus, the simulation function (named "Simulate") must return only a single x/y data set, and the x_label and y_label strings must include the units in parentheses. As before, custom units may be included using <> at the beginning of the string. For example, (<frogs>*kg/s) would be permissible.

Examples of single series json model records (simulator called one time):

amino_silane_silica_LangmuirIsothermModel_343_equilibrium.json
amino_silane_silica_LangmuirIsothermModel_343_kinetic.json

Examples of multiple series json model records (simulator called multiple times):

amino_silane_silica_LangmuirIsothermModel_343_kinetic_two_models.json
amino_silane_silica_LangmuirIsothermModel_343_equilibrium_two_models.json

It is even possible to have a json record with raw data and a model in the same file, because the model is defined at the series level.

5. Hierarchical Classification of Data Types / Hierarchical Schema

JSONGrapher is designed to plot data that is compatible to plot together, and to disallow plotting of incompatible data together.

One way that JSON tells if data is compatible is by units. But, there may be other reasons to not plot data together.

The current solution use double-underscore separators in a hierarchical way in the DataType field. For example, adsorption_isotherm has no double underscore, so the string adsorption_isotherm in the data_type field would be considered a top-level classification. Then, subsets classification types can be added in front. A datatype of CO2__adsorption_isotherm is a subset of the data type adsorption_isotherm and DRIFTS__IR__vibrational_spectrum is a subset of IR__vibrational_spectrum and also a subset of vibrational_spectrum. By following the hierarchical classification system when naming DataTypes, JSONGrapher can see if different datatypes have any overlapping parent classifications for plotting the data together.

6. Usability Considerations for how JSONGrapher was Designed

It is important for software to be easily usable, particularly if an experience economy is the goal. Accordingly, we included several considerations (but not exhaustive) to avoid the unpleasant experience of a user getting "stuck" when attempting to use JSONGrapher.

- For any file added, the software checks that the file (or data after conversion) is valid json.
- If two data sets of incompatible types are attempted to be plotted together, the software produces an error message notifying the user.
- If a data set is missing required fields (such as units) the software will notify the user
- We provide a way for users to download the most recent data as JSON or CSV.

7. Technical Considerations for how JSONGrapher was designed: File Formats and Schema

There are two primary technical decisions to make when it comes to a single implementation of the experiential results. (a) Which computer language to use? (b) Which data format to use?

On the question of which computer language to use, a decision was made to use JavaScript for this example on the basis that this would allow the simplest ease of use: the user simply needs to have a modern browser. The infrastructure is thus independent of operating system and does not require any familiarity with command lines, compilation, etc. In the present version, an internet connection is also required. Although an internet connection is required in the current implementation, an advantage of the current implementation is that the software is open source and online-hosted such that any users can make improvements, and when these are accepted into the master branch they will be immediately reflected to all users. The JavaScript is presently intentionally written in such a way that the computing power is provided by the user's computer (not by the server), though it is possible to use cloud computing for simulations in the future. A decision was then made to use the software plotly (plotly.com) as this enables versatile and interacting plotting of graphs with an open source framework.

On the question of which data format to use, there are several common structured formats that can be considered, and conversion can occur between file formats. JSON, YAML, CSV, HDF5. The key considerations for the data format are that it should have the ability to store metadata, hierarchical data, should have a robust schema framework, and ideally be human readable and human editable. HDF5 is not human-readable, but compatibility with HDF5 is desirable as HDF5 has been designed for managing extremely large and complex data collections. A schema is a set of 'rules' that specify a standard for a file. For example, a schema could specify that adsorption isotherm data must have units of pressure for the x axis, and units of mass or moles divided by mass or moles for the y axis. A thorough discussion of schema is beyond the scope of this work, but we note that for each data type provided to JSONGrapher, there must be an existing schema in the schema directory. A generic schema can be setting ScatterPlot or XY as the DataType.

YAML can store meta data, has a robust schema framework and is human readable, can in principle store hierarchical data, though is not commonly used to store data. https://yaml.org/

JSON can store meta data, has a robust schema framework (though the schema libraries are less robust than those for YAML), is human readable, is hierarchical, is commonly used for data, and is the most easily converted to HDF5, and the most easily accessible by JavaScript. Plotly is designed with this recognition, and a plotly JSON schema exists and is included in JSONGrapher. Technically, JSON data can

be included within YAML files, but in practice the two are often treated as separate formats. https://json-schema.org/

CSV / TSV file formats can store meta data, but do not have a robust schema framework (though schema frameworks do exist), but are not well suited to hierarchical data storage.

From the above considerations, JSON was chosen as the preferred format.

However, for users who are creating a file "by hand" with this use of spreadsheet software, the CSV / TSV format is most accessible. Thus, CSV compatibility has been included, by mapping the fields onto those of the JSON schema. That is, the CSV is internally converted to JavaScript arrays that are equal to a JSON, and then treated as a JSON. However, because of the limitations of CSV files, not all of the fields are editable through CSV (for example, the CSV file does not allow changing the plot layout). It is possible to add more complexity to the CSV format support, but not as facile, since it would require more complex parsing of the CSV file. Thus, the way we support additional fields in the CSV is to allow one line to be a string of unlimited length that can include JSON, in the field named "custom_variables". That field is currently ignored.

The Schema are also created in a hierarchical way, which we explain here. The way hierarchical schema are treated in YAML and JSON are different. YAML allows flexible 'importing' of fields from 'parent' Schema. With JSON Schema, the concept of a 'parent' schema does not exist: the analogous feature is to use the $ref keyword in such a way that requires the record to conform to *both* schema completely. In order to circumvent the lack of parent schema in JSON Schema, and to maintain facile compatibility with the CSV method of creating records, the current solution is to make child schema include all fields from the parent schema and to give the child schema filenames that include the parent schema after a double-underscore separator. For example, CO2__adsorption_isotherm is a subset of the data type adsorption_isotherm and DRIFTS__IR__vibrational_spectrum is a subset of IR__vibrational_spectrum and also a subset of vibrational_spectrum.

8. License

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to https://unlicense.org

9. Credits

JSON Grapher utilizes plotly, UUC, AJV, and custom code.
https://plotly.com/
https://github.com/Lemonexe/UUC
https://ajv.js.org/

Piotr Paszek made the core code of JSON Grapher, which relies on plotly. He has significant experience with javascript and data visualization, and he may be hired at https://www.upwork.com/freelancers/paszek

Med. Amar Filali added most of the additional features: including unit conversions (using UUC), the ability for external simulations, and CSV download of the last dataset. He has significant experience in making dynamic websites and specialized Javascript codes. He may be hired at https://www.upwork.com/freelancers/~01844d5a23ecf022cf

The idea of JSONGrapher was conceived of by Aditya Savara, and it is used as a demonstration for the concepts described in a publication which has the core authors of Aditya Savara, Sylvain Gouttebroze, Stefan Andersson, Francesca Lønstad Bleken.