JSONGrapher Manual

By Aditya Savara

The concept:

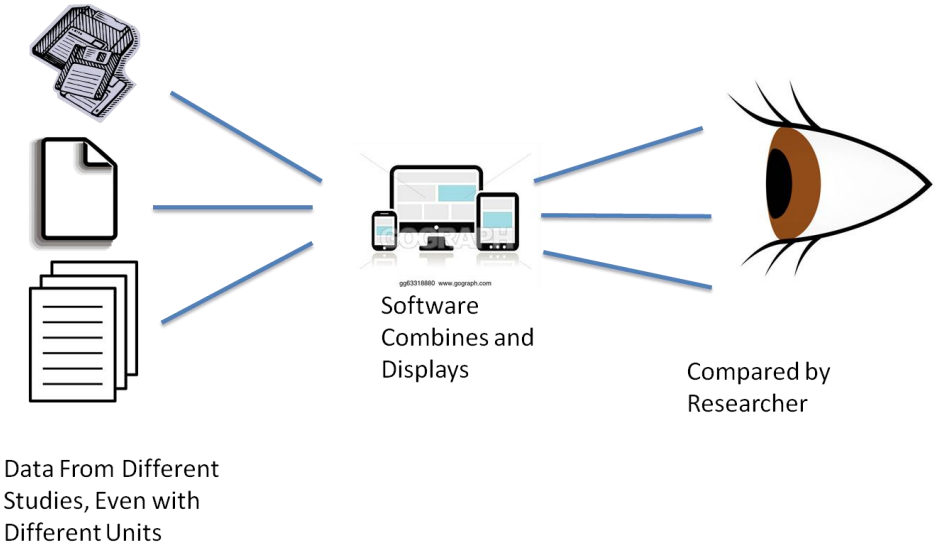


Table of Contents

[Manual Continued on Next Page]

0a. Introduction:

Imagine a world where a person can simply drag a datafile into a graphing utility and a plot will be made, including scientific units, and more plots can be dragged in from data from other sources (and other units) to compare all of the data together, in an interactive graph. Imagine that the units of all of these datasets will be converted automatically, as needed, for the comparison.

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 files are simple format and can even be made "by hand" as .csv files or .json files.

When two datasets have different units (such as kg/min and g/hour), JSON Grapher will *automatically* convert 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.

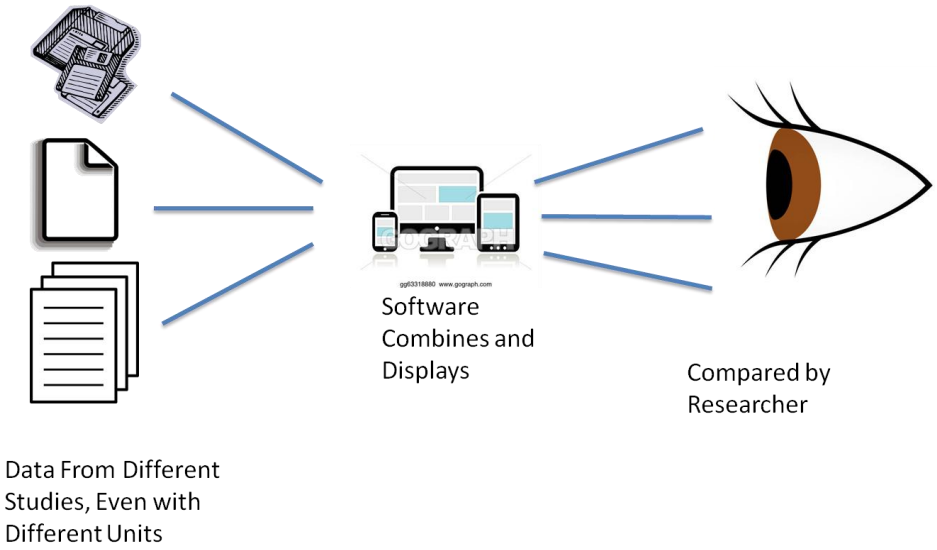


Figure 1: JSONGrapher can take data from different studies, even with different units, then combine them into one graph for direct comparison by researchers. JSON Grapher can also run models and plot the outputs alongside measured or existing data.

0b. Quick Start (how to plot)

Download the 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.**

Now, directly try the "First Usage" instructions below, or watch the demonstration video.

First Usage: Drag in an Example File.
Open www.jsongrapher.com and drag in the following file to see many series at once:

UAN_DTA Consolidated_descending.json

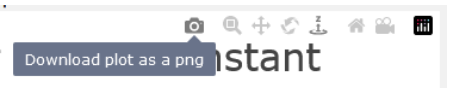
Then click "Clear Data", and drag in in the below file to see a fancy graph.

SrTiO3_rainbow.json

Then click "Clear Data", and drag in in the below file to see a 3D graph.

Rate_Constant_mesh3d.json

To save an image file, hover over the top right of the graph and click on the camera icon:



Second Usage: Drag in Multiple Example Files, one at a time.
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!
Then click "Clear Data" to be able to try more examples.

Third Usage: Drag in an advanced example file.
For a fancy example file, try dragging in:

O_OH_Scaling.json

Then try moving your mouse over individual data points.

Fourth Usage: From URL

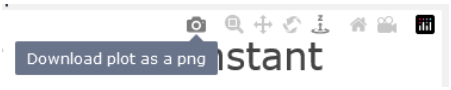
Just click on the below link to see a 3D JSONGrapher plot! You can rotate the plot etc.!
http://www.jsongrapher.com?fromUrl=https%3A%2F%2Fraw.githubusercontent.com%2FAdityaSavara%2FJSONGrapherExamples%2Fmain%2FExampleDataRecords%2F9-3D-Arrhenius%2FRate_Constant_scatter3d.json

Now you know how to plot files in JSON Grapher in a basic way!
The best way to create records is using the python JSONGrapher, but you can also use the browser record creator to create to create your own simple files. The rest of this manual is made for people who wish to make JSONGrapher files.

1. Plot Types & Adding Color Gradients / Colorscales

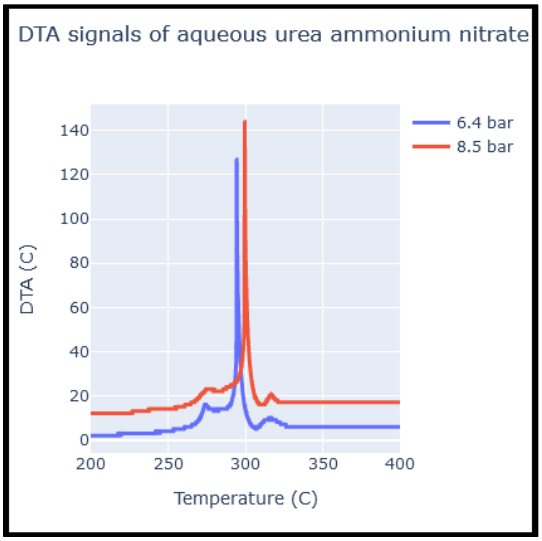
This section gives an overview of the plot_types available. All of the plot types can have color gradients added to them through a feature called colorscales. For more information about how to use the colorscales feature, see further below.

To save an image file, hover over the top right of the graph and click on the camera icon:



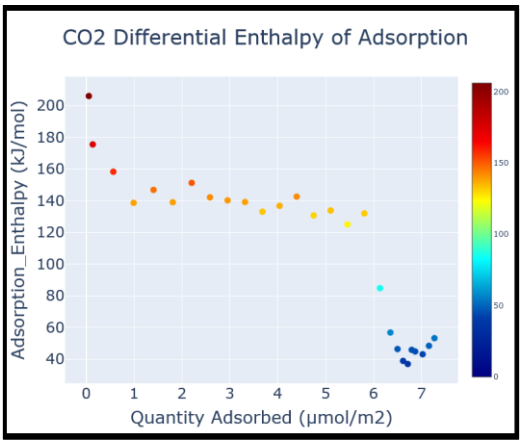
a. Trace_style : spline

To create a spline plot, simply create a regular record with "trace_style": "spline" for each data series.



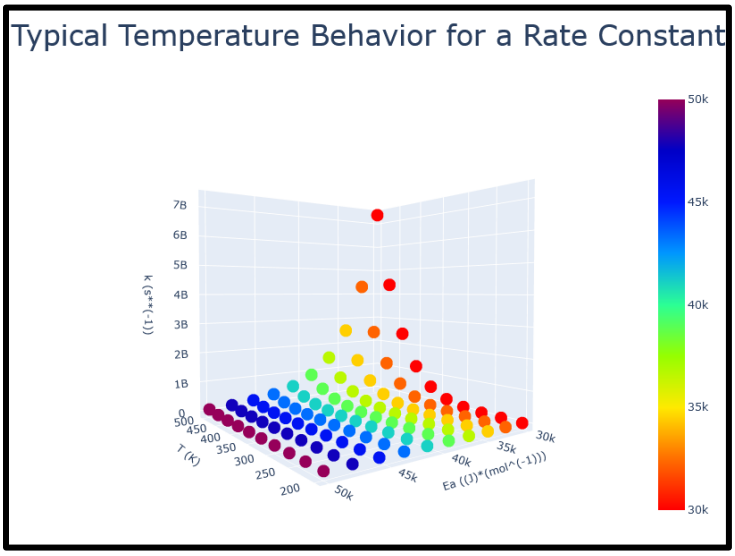
b. trace_style : scatter__colorscale and trace_style: spline__colorscale

To create a scatter__colorscale plot, simply make the trace_style as 'scatter' followed by two underscores and a colorscale, like "trace_style":"scatter__jet". Here is an example file (click) and here is the list of colorscales for online JSONGrapher. The python JSONGrapher supports a longer list of colorscales for python JSONGrapher.



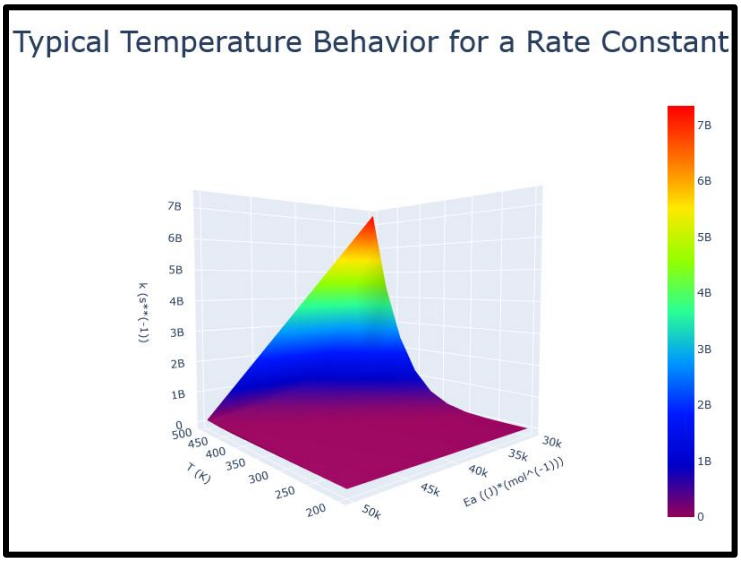
c. trace_style : scatter3d

Examples of scatter3d and how to make them are here, and the colorscales can also be changed using the “scatter3d__colorscale” like “scatter3d__RdBu”



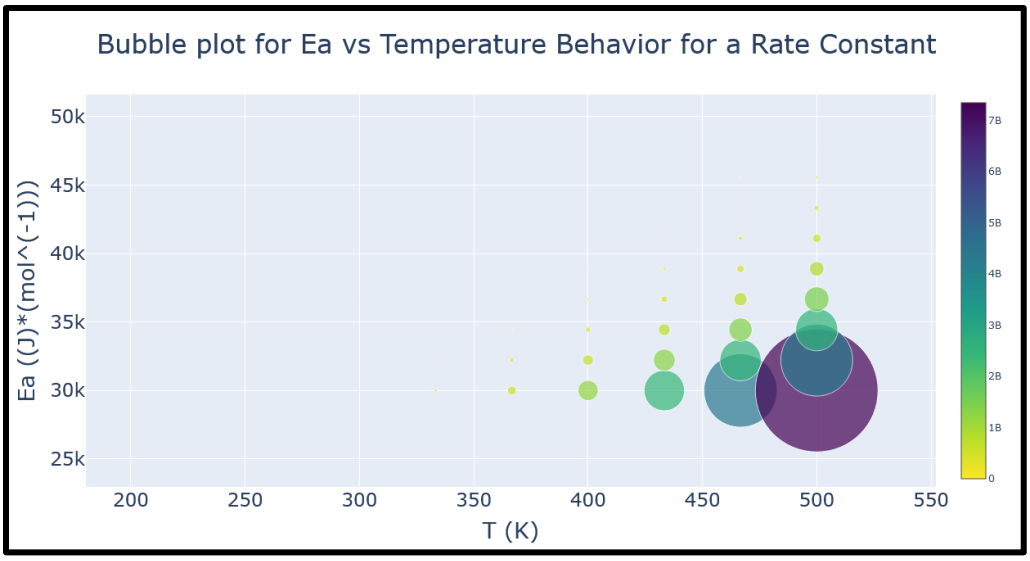
d. trace_style : mesh3d

Examples of mesh3d and how to make them are here, and the colorscales can also be changed using the “mesh3d__colorscale” like “mesh3d__RdBu”



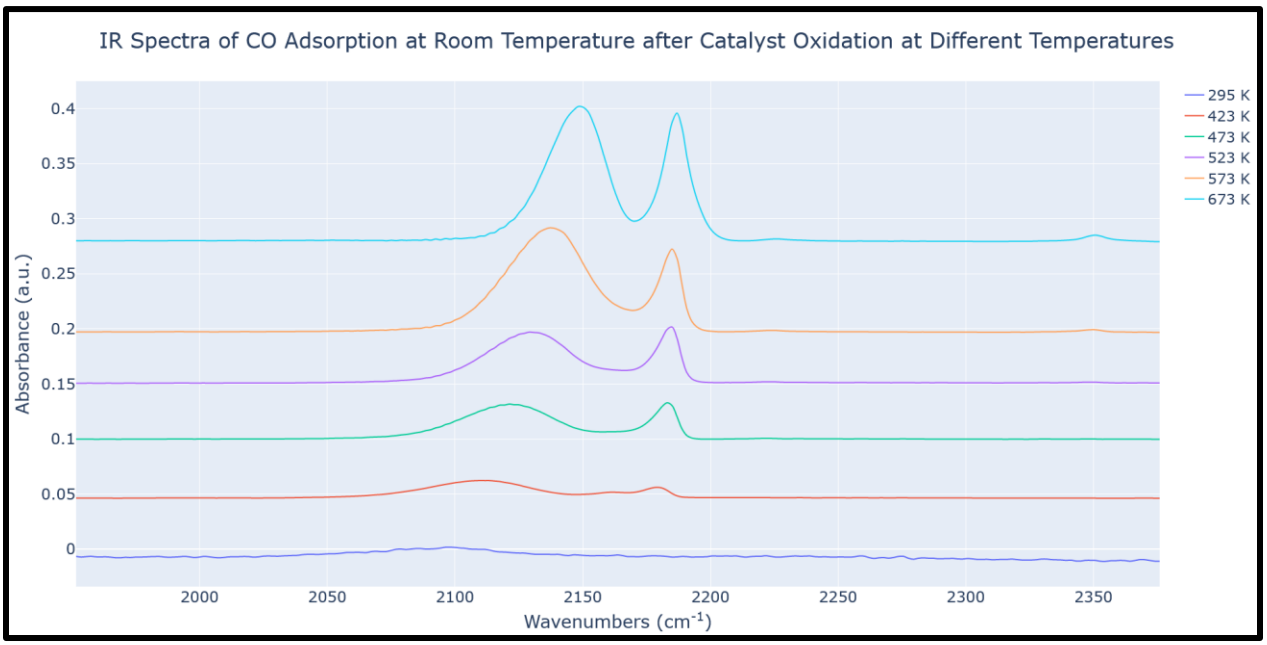
e. trace_style : bubble2d

Examples of bubble plot and how to make them are here. The bubble size can be changed with the “"max_bubble_size" field. As with the other plot types, the colorscales can be changed with the underscore delimeter with syntax of “bubble2d__colorscale” like “bubble2d__RdBu”



e. layout_style : offset2d

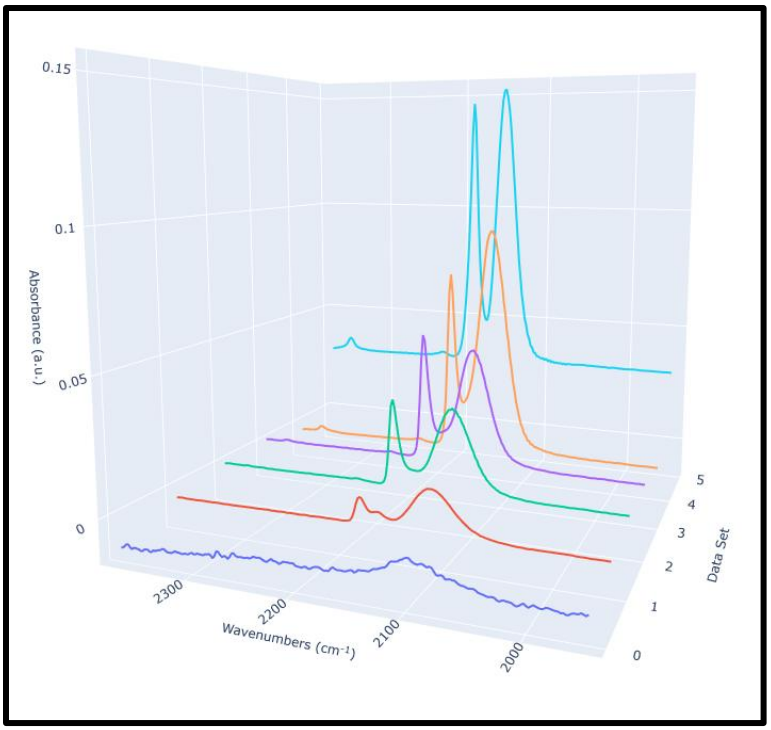
This is not actually a trace_style, but a layout_style which can be made with either spline or scatter plots. Examples of offset2d and how to make them are here. The offset size can be changed with the "offset" field inside layout.



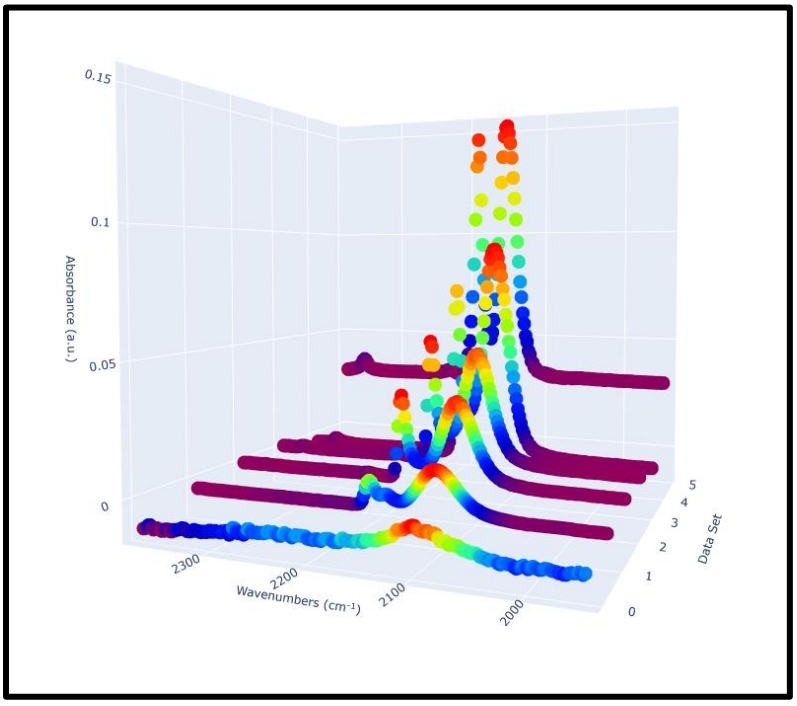
e. layout_style : arranged2dTo3d

This is not actually a trace_style, but a layout_style which can be made with either spline or scatter plots. Examples of arrange2dTo3d and how to make them are here.

Example using spline for each trace (creates rainbow of curves):



Example using scatter for each trace (creates heated gradient with points):



2. “From URL” Plots, Embedding, and use in Go/Wails Apps

(A) Creating a “From URL” Plot Link

To create a “From URL” plot, like this example Scatter3d.json link, there are three steps.

1. Add a JSONGrapher record to a github repository (other web locations may work, but will depend on their server settings and are not likely to work)
2. Click “From URL” on www.jsongrapher.com and paste the URL of the json record.
3. After the graph appears, click “Copy URL” to the right of the screen. Now you have a link that you can share for people to look at your graph even without drag and drop!

To combine multiple records in a URL, you will need to plot multiple records, then click the download json button (which will give you a combined json) and then upload that .json file for step 1 above.
More examples: UAN_DTA Consolidated.json Model_equilibrium.json Rate_Constant_mesh3d.json

(B) Embedding a Plot Into your Webpage

To embed a plot into your webpage, upload a JSONGrapher .json record to github or to dropbox, make sure it is publicly accessible, and follow the instructions at the below link. Unfortunately, google drive and onedrive do not allow public file access from APIs, so you cannot use google drive or onedrive for storing a JSONGrapher record that you want to have plotted on your webpage.
http://jsongrapher.com/other_html/EmbeddingExample/EmbeddingExample.html

(C) Including JSONGrapher Graphs in Go/Wails Apps

The package Wails enables graphical user interfaces for Go programs that are essentially browser based Javascript GUIs. JSONGrapher can thus be used with Go, and a simple example is provided here:
https://github.com/AdityaSavara/JSONGrapher-Go-Wails-Example

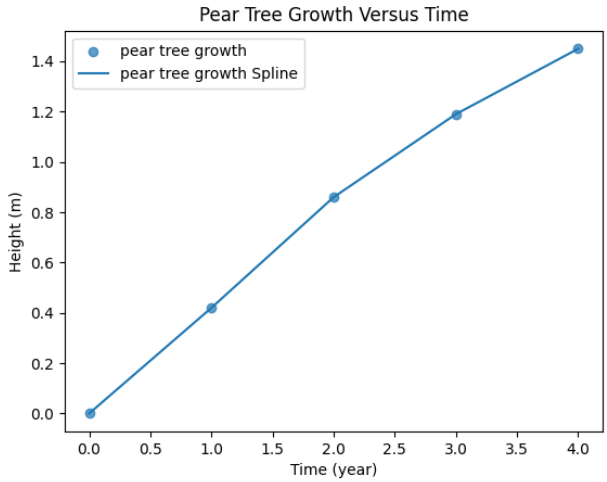
3. Creating JSON Files (python JSONGrapher and browser creator)

JSON files are the 'standard' way to use JSONGrapher. They enable using all of the functionalities of JSONGrapher. Additionally, evaluated JSONGrapher files are compatible with the plotly JSON format. Accordingly, this open format is not likely to ever become deprecated or unsupported and will be convertible to other open graph formats, if ever needed.

The best way to create JSON files is using the python JSONGrapher, see the website for that at https://github.com/AdityaSavara/JSONGrapher-py. However, simple records can also be created manually using the Browser based record creator.

The python JSONGrapher package not only enables creating JSONGrapher records, it also enables plotting interactive Plotly graphs, and also matplolib graphs for inspection and saving. It also includes "styles" so that graphs could be formatted for specific journals (or other uses) with a single command.

```
{
   "comments": "Tree Growth Data collected from the US National Arboretum",
   "datatype": "Tree_Growth_Curve",
   "data": [
      {
         "name": "pear tree growth",
         "x": [0, 1, 2, 3, 4],
         "y": [0, 0.42, 0.86, 1.19, 1.45],
         "type": "scatter",
         "line": { "shape": "spline" }
      }
   ],
"layout": {
   "title": { "text": "Pear Tree Growth Versus Time" },
   "xaxis": { "title": { "text": "Time (year)" } },
   "yaxis": { "title": { "text": "Height (m)" } }
}
```



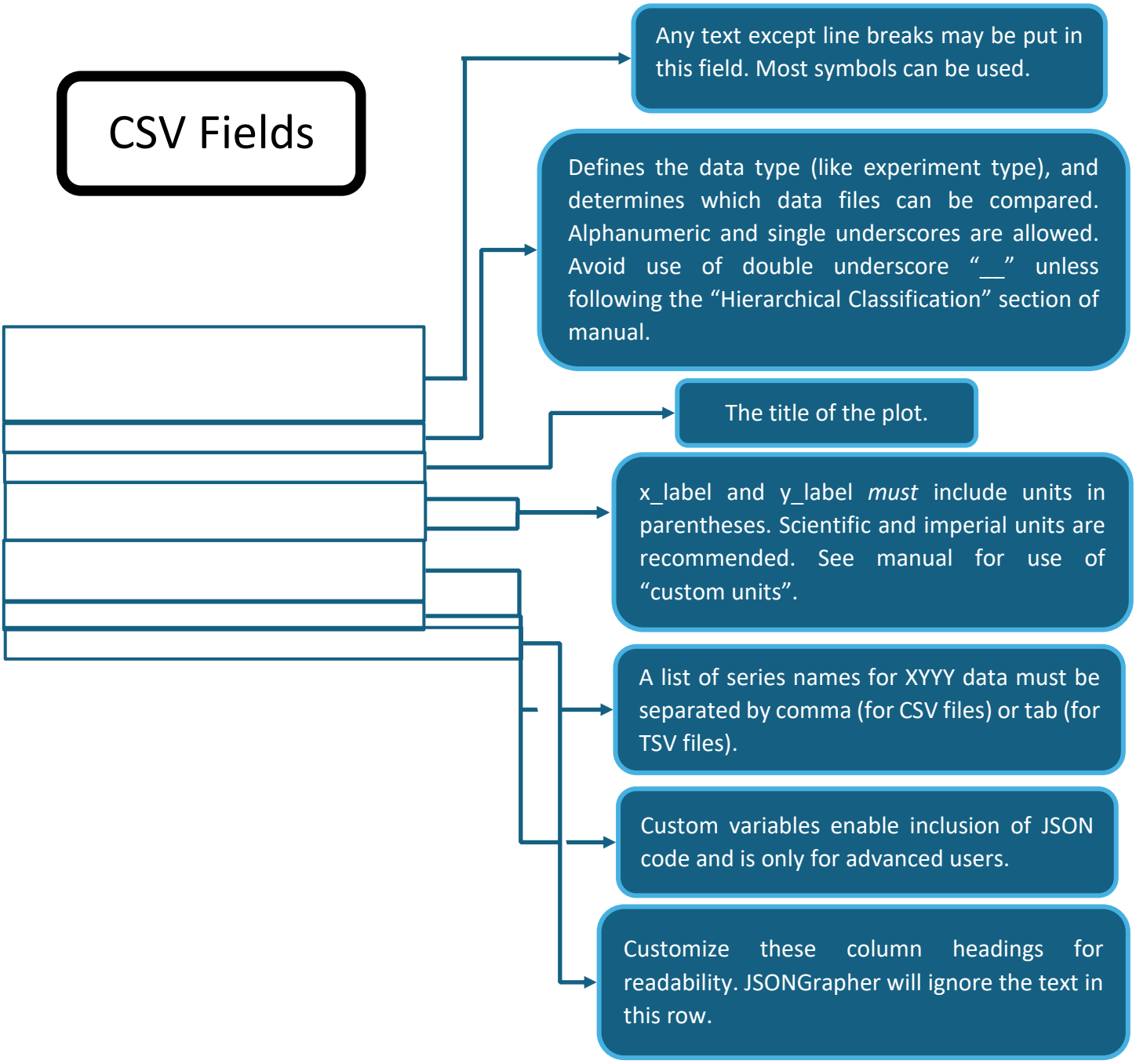
Exporting Images to File from Python JSONGrapher:
In python JSONGrapher, the standard way to export images to file is
Record.export_to_png()

This is expected to work without problems if you have python Plotly version 6.2.0
For python Plotly versions earlier than 6.0, use 'pip install kaleido==0.1.0'

[Manual Continued on Next Page]

4.Creating CSV Files: CSV Fields, Labeled

CSV files can be used to make very simple 2D JSONGrapher records. Most features are only available by using .json files. The colons are important.



5. Explanation of Fields , list of Supported Data Series Types (XY, XYYY)

a. Explanation of Fields in JSON Data Records

It is easiest to add fields using the python JSONGrapher without trying to understand all of the fields. However, in the JSONGrapherExamples github repository there are various records such as single series json record and example multiseries json record.

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.
"datatype":	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 dataype name.
"data":	This field has the structure of [{series1},{series2}] Within each series object is a data series dictionary. Each data series dictionary will contain either the data to plot ("x" and "y" fields with lists of values), *or* will contain an "equation" field with an equation to plot, *or* a "simulation" field to call a simulation. More information is found in the "Data Series Dictionaries / subJSON" section of this manual. The "trace_style" field will allow setting how a data_series will be plotted, such as "spline", "scatter_spline", "scatter3d" etc. See the "styles" section of the manual for more information.
"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. The text is contained in subfields "title": { "text": "A string that will become the chart title." } "xaxis": { "title": { "text": "x_label_string" } } "yaxis": { "title": { "text": "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. Units should be non-plural (kg instead of kgs) and should be abbreviated (m not meter). 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 fields)	Custom fields are allowed in the JSON records at the highest level (these can be dictionaries), and users may use custom fields to store meta data. JSONGrapher allows custom fields even within the default fields, but that is not strictly compatible with plotly. To add custom fields to deeper levels within default fields, any field names are allowed, but it is recommended to make those fields strings named "comments" or dictionary/strings named "extra_information", as the JSONGrapherRC python package will remove those fields if a plotly json object is requested.

b. 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. Can be used to put in a general description or metadata related to the entire record. Can include citations and links. Goes into the record's top level comments field.
DataType:	The datatype is the experiment type or similar, would be shared among data files that can be compared. It is used to assess which records can be compared and which (if any) schema to compare to. Alphanumeric characters and underscores are allowed. Use of single underscores between words is recommended. This ends up being the datatype field of the full JSONGrapher file. Avoid using double underscores '__' in this field unless you have read the manual about hierarchical datatypes. 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. Use "^" for exponents. It is recommended to have no numbers in the units other than exponents, and to thus use (bar)^(-1) rather than 1/bar.
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. Use "^" for exponents. It is recommended to have no numbers in the units other than exponents, and to thus use (bar)^(-1) rather than 1/bar.
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. By default, JSONGrapher will only include a trendline for series with more than 20 points, and will have points and trendline for less.

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).

c. Explanation of Data Series Types, Model Files, and their usage with JSONGrapher

JSON Grapher can plot several types of data, and different file extensions:

Single Series XY data (from .json or .csv)
Multiple series XYYY data (from .json or .csv)
Multiple series XY XY XY data (only from .json)

Series of XY Data from Analytical / Numerical Simulation Models (only from .json)
Note: JSONGrapher model files will not have the series data in them. They JSONGrapher Model files will only contain parameters and call an external simulation function which then returns the XY data as needed. The results of the simulation function are then plotted and can also be downloaded. A JSONGrapher model file can call for simulation of multiple series.

Each these possibilities are shown in examples inside the JSONGrapherExamples github repository , can be downloaded inside of the zipfiles ExampleDataRecords.zip and ExampleModelRecords.zip (a list of which files correspond to the above data types is included further below).

JSONGrapher can also plot series as scatter plots, line plots, and other ways (though at the time of writing this sentence, we do not have a section on how to do so in the manual).

Currently, JSONGrapher actually supports three formats: ".JSON", ".csv", and ".tsv". The file format will be recognized if the file extension is there. The csv and tsv files are comma separated and tab separated, respectively. Tab separated files are treated the same way as csv files by JSONGrapher, but are

sometimes 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 XY 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
3-..\DRIFTS_CO_Adsorption_onAu22.csv
4-..\PtAtomPMFOnTiO2Rutile110.csv

Multiple series XY XY XY data (only from .json)

1-..\CO2Adsorption_NaX_and_CaX_two_series.json
8-..\O_OH_Scaling.json (includes linear fit as well)

XY 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.

For dataseries with under 10 points, JSONGrapher will plot discrete points by default. For dataseries with more than 10 points, JSONGrapher will omit discrete points by default. This behavior can be seen with 2-..\La_Perovskites_Combined.json and 2-..\Sr_Perovskites_Combined.json

d. 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.

6. Data Series Dictionaries / subJSON

Each data series in a JSONGrapher record has a data_series dictionary (which is the same thing as a subJSON). Below are several examples.

Example 1: Data Series Dictionary, Basic Example with x,y data:

```
{
   "uid": "45c0a4",
   "name": "SeriesName",
   "trace_style": "scatter",
   "x": [ 7,  8,  9,  10 ],
   "y": [  0, 4, 9,  2, ]
}
```

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 good, while .004 would give an error).

A “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).

Example 2: Data Series Dictionary, Equation Example For x,y,z data:

This example is for a 3D Arrhenius equation, and is found in the example here.
This example includes optional fields. For more explanation about how to use the “equation” field, see the section on “Equation” and “Simulation”.


```
{
    "uid": "",
    "name": "Arrhenius Example 3D plot",
    "trace_style": "mesh3d",
    "x": [],
    "y": [],
    "equation": {
      "equation_string": "k = A*(e**((-Ea)/(R*T)))",
      "graphical_dimensionality": 3,
      "x_variable": "T (K)",
      "y_variable": "Ea (J*mol^(-1))",
      "z_variable": "k (s**(-1))",
      "constants": {  "R": "8.314 (J*mol^(-1)*K^(-1))",  "A": "1*10^13 (s^-1)",  "e": "2.71828"  },
      "num_of_points": 10,
      "x_range_default": [ 200,  500 ],
      "x_range_limits": [],
      "y_range_default": [ 30000, 50000 ],
      "y_range_limits": [],
      "x_points_specified": [],
      "points_spacing": "Linear",
      "reverse_scaling": false,
      "verbose": true
    }
  }
```

Example 3: Data Series Dictionary, Simulation Example for a particular model:
Simulations can be called in several ways.

a) A javascript simulation can be called from either online JSONGrapher or from python JSONGrapher. See the section called “Javascript Simulation Calls”.
b) Python functions can be called directly when using the python version of JSONGrapher (see section called “Python Simulation Calls section”).
c) Python functions can be called by an https call by either the online or python version of JSONGrapher (see section called “Python Simulation Calls section”).

The below example is a dataseries dictionary for with a “simulate” field to simulate using javascript, from here. To choose between different sources of simulation, one alters what is in in the “model” field. For a javascript simulator, one puts the link to the .js file in the model field like here.
For a local_python simulator to be called directly (using python JSONGrapher), the model field will have the value “local_python” like the example here.
For a local python simulator to be called using https, one must follow the instructions in the later section of this manual (one cannot just drop the example file), with the example here.


```
{
  "line": {
    "shape": "spline",
    "width": 3
  },
  "name": "CO2 Adsorption, K_eq = 99.6 (1/bar)",
  "type": "scatter",
  "x": [],
  "y": [],
  "simulate": {
    "model":
"https://github.com/AdityaSavara/JSONGrapherExamples/blob/main/ExampleSimulators/Langmuir_Isotherm.js",
    "K_eq": "99.6 (1/bar)",
    "sigma_max": "1.0267670459667 (mol/kg)",
    "k_ads": null,
    "k_des": null
  }
}
```

7. Trace Styles & Layout Styles

Both the online version and the python version of JSONGrapher support Trace Styles and Layout Styles.

- A **layout style** defines the formatting for the graph (like the title, the background color, the axes).
- A **trace_style** defines the formatting for a dataseries (like “scatter”)
- A **trace_styles_collection** changes *how* trace styles are represented on a graph. The trace_styles_collection is thus a little bit like a theme that gets applied to all of the data series. For example, it could force all of the data series to have connecting lines, or to appear as triangles, etc.

Online JSONGrapher styles files: https://github.com/AdityaSavara/JSONGrapher/tree/main/styles
Python JSONGrapher styles files: https://github.com/AdityaSavara/jsongrapher-py/tree/main/JSONGrapher/styles

There is a sortof “tutorial” for styles in Python in two of the example files:
example_2_creating_records_and_using_styles
example_7_custom_formatting_and_style_extraction

Note that instead of having a string for the “trace_style”, an entire trace_style dictionary can be passed in for the trace_style. The same is true for the layout_style and the trace_styles_collection, so a user is not confined to the built in styles.

One can also set the “trace_style” to “none” for a single series, or the “trace_styles_collection” to “none” for all series, and then manually edit the formatting of the series.

For example, for each dataset, the type of graph can be changed using the “mode” field, with a value of “markers” being a simple scatter, “line” being straight connecting lines, "line+markers” is also an option. See for Example O_OH_Scaling.json for an example of the use of the “mode” field.

8. 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.

9. Equation Field and Simulate Field

JSONGrapher supports two kinds of implicit definitions of data series.

The **"equation"** field in data series dictionaries supports symbolic expressions. It is not restricted to algebra. One can use exponential forms and other standard mathematical forms. Units must be provided for each variable and constant. Constants such as "e" or the "ideal gas constant" should be defined within the equation dictionary. Units *cannot* be provided in the equation string: if the equation has a term with units (like a y-intercept of 5 m), you must define a constant for that term. For 2D graphs, the independent variable must be "x" and the dependent variable must be "y". For 3D graphs, the independent variables must be "x,y" and the dependent variable must be "z". The user must define a default range for the independent variables axes (x or x/y) because the graph must be bounded when first plotted. Other optional settings exist. See the example directories: Example_5 and Example_6 and Example_9. To understand the other optional settings, see equation_creator.py .

The **"simulate"** field in data series dictionaries supports arbitrary functions. The key requirement is that the function should accept the simulate dictionary and then must return back the same simulate dictionary populated with the simulation outputs.

Simulations can be called in several ways.

a) A javascript simulation can be called from either online JSONGrapher or from python JSONGrapher. See the section called "Javascript Simulation Calls".
b) Python functions can be called directly when using the python version of JSONGrapher (see section called "Python Simulation Calls section").
c) Python functions can be called by an https call by either the online or python version of JSONGrapher (see section called "Python Simulation Calls section").

The Data Series Dictionaries section of this document has an example dataseries dictionary for with a "simulate" field to simulate using javascript, from here. To choose between different sources of simulation, one alters what is in in the "model" field.
For a javascript simulator, one puts the link to the .js file in the model field like here.
For a local_python simulator to be called directly (using python JSONGrapher), the model field will have the value "local_python" like the example here.
For a local python simulator to be called using https, one must follow the instructions in the later section of this manual (one cannot just drop the example file), with the example here.

10. Javascript Simulation Calls

JSONGrapher files can be made with entries containing simulation parameters rather than data, whereupon JSONGrapher will call to execute external simulations to obtain XY data to be plotted.

Such calls can be made to external Javascript simulation functions. The requirements are that the javascript must be hosted on github with a function named “simulate”. The function name must be lower case. That function must receive a single argument: a JSON object which has the a field named “simulate”, and which then contains parameters needed parameters in subfields. The javascript function must return a JSON object with the field “data” and with the original subfields within that of “x”, “y”, “x_label”, “y_label”, now populated with the simulated values.

For the convenience of anyone who will be making javascript simulation functions, there is also a standalone html file for testing javascript functions. Opening the tester will be self explanatory for most developers, but we also include a README on how to use the testing file.
https://github.com/AdityaSavara/JSONGrapherExamples/tree/main/ModelSimulationTesters
You can also open the tester by clicking here:
https://adityasavara.github.io/JSONGrapher/other_html/ModelSimulationTesters/javascript_function_tester.html

Example javascript file:
https://github.com/AdityaSavara/JSONGrapherExamples/blob/main/ExampleSimulators/Langmuir_Isotherm.js

Example JSON that would be “plotted” with JSONGrapher and which woudl call that javascript file:
https://github.com/AdityaSavara/JSONGrapherExamples/blob/main/ExampleModelRecords/1-CO2__Adsorption_Isotherms/amino_silane_silica_LangmuirIsothermModel_343_equilibrium.json

Example input that javascript function could receive:


```
{
  "comments": "// The curly bracket starts a data series. A file can have more than one data series. The uid is an optional unique ID and can even be a doi, for example.  The name field is the name of the series and will appear in the legend.",
  "line": {
    "shape": "spline",
    "width": 3
  },
  "name": "CO2 Adsorption, K_eq = 99.6 (1/bar)",
  "type": "scatter",
  "x": [],
  "y": [],
  "simulate": {
    "comments": "// The model field allows description of whether it is an elementary step model or some other kind of model. In this case, the model is a Langmuir_Isotherm model. This model requires
```

having *either* K_E or k_ads and k_des. The fields of k_f and k_r will only be checked if the K_E has null as its value. The units of pressure must be expressed with a division symbol like 1/bar.",
   "model":
"https://github.com/AdityaSavara/JSONGrapherExamples/blob/main/ExampleSimulators/Langmuir_Isotherm.js",
   "K_eq": "99.6 (1/bar)",
   "sigma_max": "1.0267670459667 (mol/kg)",
   "k_ads": null,
   "k_des": null
  }
}

Example output that javascript function would send back to JSONGrapher:


```
{
  "success": true,
  "message": "Simulation initialized successfully",
  "data": {
    "comments": "// The curly bracket starts a data series. A file can have more than one data series. The uid is an optional unique ID and can even be a doi, for example.  The name field is the name of the series and will appear in the legend.",
    "line": {
      "shape": "spline",
      "width": 3
    },
    "name": "CO2 Adsorption, K_eq = 99.6 (1/bar)",
    "type": "scatter",
    "x": [
      0.001145434009333668,
      0.0025772265210007527,
      0.0044181026074298635,
      0.006872604056002009,
      0.010308906084003013,
      0.015463359126004517,
      0.024054114196007025,
      0.04123562433601206,
      0.09278015475602713
    ],
    "y": [
      0.1,
      0.2,
      0.3,
      0.4,
      0.5,
      0.6,
```

```
    0.7,
    0.8,
    0.9
   ],
   "simulate": {
    "comments": "// The model field allows description of whether it is an elementary step model or some other kind of model. In this case, the model is a Langmuir_Isotherm model. This model requires having *either* K_E or k_ads and k_des.  The fields of k_f and k_r will only be checked if the K_E  has null as its value. The units of pressure must be expressed with a division symbol like 1/bar.",
    "model": "https://github.com/AdityaSavara/JSONGrapherExamples/blob/main/ExampleSimulators/Langmuir_Isotherm.js",
    "K_eq": "99.6 (1/bar)",
    "sigma_max": "1.0267670459667 (mol/kg)",
    "k_ads": null,
    "k_des": null
   },
   "x_label": "Pressure (1/(1/bar))",
   "y_label": "Amount Adsorbed (mol/kg)"
  }
}
```


11. Https Calls for Simulations by other Languages)

JSONGrapher files can be made with entries containing simulation parameters rather than data, whereupon JSONGrapher will call to execute external simulations to obtain XY data to be plotted.

Such calls can also be made by http calls that enable the simulations to be executed by functions in any language (using https API POST fetch requests).

This functionality can also be used with services such as “pinggy” to run files on your own computer using https calls. Free pinggy https tunnels have a time limit of around 30 minutes or 1 hour, so each time one wants to upload a JSON file to JSONGrapher, one would need to update the link. If one does not wish to keep updating the link, one can upgrade to a paid user of pinggy.

The external function should expect to receive a JSON-like string and return a JSON-like string with the same requirements as in the previous Javascript section.

For greater details on how to use https calls for simulations, see the section on Python Simulation Calls, which gives an example.

The code that enables JSONGrapher to make https calls is actually a javascript wrapper located here: https://github.com/AdityaSavara/JSONGrapherExamples/blob/main/ExampleSimulators/https_simulator_link.js

For the convenience of anyone who will be making https calls for simulations, there is also a standalone html file for testing the https calls to functions. Opening the tester will be self explanatory for most developers, but we also include a README on how to use the testing file.
https://github.com/AdityaSavara/JSONGrapherExamples/tree/main/ModelSimulationTesters
You can also open tester by clicking here:
https://adityasavara.github.io/JSONGrapher/other_html/ModelSimulationTesters/https_call_tester.html

12. Python Simulation Calls (run on your own computer)

If using the python version of JSONGrapher, python simulation calls can be done easily by following this example. That method involves providing the function into python JSONGrapher: for security reasons a JSONGrapher record will never execute python source code directly.

The rest of this section will be for the case of python simulation calls using https calls, which will work with both the online JSONGrapher and the python JSONGrapher.

JSONGrapher records can be setup to generate a plot with a data series made by simulation using on-the-fly by Python simulations that are run on your own computer (or server).

To enable JSONGrapher to safely make a call to run a python simulation on your own computer for on-the-fly data, we utilize python flask in conjunction with the service pinggy. Pinggy enables an ssh based http call, and services similar to pinggy can also work, but it's easiest to use pinggy.

Below, we'll give the basic conceptual steps, then go through running an example, then describe the key details to making your own.

Basic Conceptual Steps

The basic conceptual steps are as follows.

(1) Start python flask
(2) Make an https call ink (we will use a service named pinggy)
(3) Drop the appropriate file into JSONGrapher to initialize
(4) Drop the appropriate file into JSONGrapher again to plot.

The initialization step is only needed the first time for any given plot. Dragging further files for simulated or non-simulated data will get plotted without delay.

Demonstration (will require two separate command prompts).

(1) Create a python environment, pip install flask, pip install flask_cors.
(2) Download the JSONGrapher examples repository zip:
https://github.com/AdityaSavara/JSONGrapherExamples/archive/refs/heads/main.zip
(3) Copy the ModelSimulatorPython directory out of that zip file, and put it where you will be running your python from.

a. When you are making your own applications, you will likely make a separate copy of this folder for each place you want to run simulations from. (otherwise it will get cluttered)

(4) Start flask by opening a command prompt running:
    python flask_connector.py
(5) Create a pinggy link by opening a separate command prompt and running the below command. Keep the command as written below, including retaining "adityasavara.github.io". See further instructions below the command.
    ssh -p 443 -o StrictHostKeyChecking=no -R0:127.0.0.1:5000 a.pinggy.io x:xff x:fullurl a:origin:adityasavara.github.io x:passpreflight

    You will be asked for a password. Press "Enter" on your keyboard without entering any password. Any https link will appear in the terminal. Highlight this link and copy it. (Or, ctrl+click on the link and then copy the link from the browser address). You will need this link for the next step.
(6) Open the file of https_343_equilibrium.json inside of \JSONGrapherExamples\ModelSimulatorPython\python_models
    Paste the https link you copied into the field "https_call_link"
(7) Drag the example file of https_343_equilibrium.json into JSONGrapher

The above steps should work! If they don't work, try replacing the last step with opening https://adityasavara.github.io/JSONGrapher/other_html/ModelSimulationTesters/https_call_tester.html
You can then paste your pinggy link in there (and upload https_343_equilibrium.json) and try to do a test of the python flask simulation call directly, without JSONGrapher. It can help give some clues about the problem.

Key Details To Making Your Own Python Simulation Call

Making your own python simulation call functionality is best performed in with local testing.

(1) Create a python environment, pip install flask, pip install flask_cors.
(2) Download the JSONGrapher examples repository zip: https://github.com/AdityaSavara/JSONGrapherExamples/archive/refs/heads/main.zip
(3) Copy the ModelSimulatorPython directory out of that zip file, and put it where you will be running your python from.
    a. When you are making your own applications, you will likely make a separate copy of this folder for each place you want to run simulations from. (otherwise it will get cluttered)
(4) Next, make your own python file with your own function (it can simply be a "wrapper" that calls a function from a more sophisticated module or package).
    a. See the example files for example inputs.
    b. You can call your function anything.
    c. The function should take in a single JSON-like dictionary as an argument, and return a single JSON-like dictionary as an argument.
        i. Since there is no "x" or "y" data,

d. In your function, it is a best practice to do a back and forth conversion between JSON-like string and JSON-like dictionary at the top and the bottom to make sure your JSON will be valid for JSONGrapher. The example python model files provided do so.

(5) Optional: Test your function with hard-coded inputs See the "if.. main" statements in the example python file (click).

(6) Create a JSONGrapher file for use with your python function, and test. See the "if.. main" statements in the example python file (click).

(7) Come up with a second name for your function call, one that is suitable to put inside the JSONGrapher file. It is best to make your dictionary label different from the function's actual name to help with debugging. Do the following steps.

a. Go into Inside the ModelSImulatorPython directory, edit flask_connector.py to import your function, and to have your function linked inside the dictionary. This dictionary will pull out your function, so you can ignore the existing entries (or you can remove them along with their imports).

b. Add your function's dictionary label inside of the JSON file in the field simulation_function_label.

(8) Start flask by opening a command prompt running:

python flask_connector.py

(9) In the steps below, we'll do an https call test to your function with without JSONGrapher.

(10) Create a pinggy link by opening a separate command prompt and running the below command. Keep the command as written below, including retaining "adityasavara.github.io". See further instructions below the command.

ssh -p 443 -o StrictHostKeyChecking=no -R0:127.0.0.1:5000 a.pinggy.io x:xff x:fullurl a:origin:adityasavara.github.io x:passpreflight

You will be asked for a password. Press "Enter" on your keyboard without entering any password. Any https link will appear in the terminal. Highlight this link and copy it. (Or, ctrl+click on the link and then copy the link from the browser address). You will need this link for the next step.

(11) Optional step: Open https_call_tester.html. You can open it locally, or you can use the below link:

a. https://adityasavara.github.io/JSONGrapher/other_html/ModelSimulationTesters/https_call_tester.html

b. Paste the pinggy https link into the URL field.

c. Click the ping call button.

d. Browse to and then load your JSONGrapher file (similar to https_343_equilibrium.json)

i. Make sure you click load.

e. Click the button to make the HTTPS Simulation Call.

f. You should see the JSON output from your simulation on the screen! This means things are working!

(12) Copy your pinggy link into the https_call_link field under "simulate" within your JSONGrapher file. **This step is critical to real usage. The most common cause of problems is forgetting to copy the new pinggy URL when testing or using.**

(13) Drag your JSON file into JSON Grapher. First drag it once to initialize, then drag it a second time for the simulation.
    a. The initialization step is only needed the first time for any given plot. Dragging further files for simulated or non-simulated data will get plotted without delay.

Remember that you need to copy the pinggy link into your json file after *each* time you generate a new pinggy link. The "connection" to run python is always temporary. If you want a permanent unchanging like, read the note below.
Note: Free pinggy https tunnels have a time limit of around 30 minutes or 1 hour, so each time one wants to use this feature to call a python function on their computer with JSONGrapher, one would need to update the link. If one does not wish to keep updating the link, one can upgrade to a paid user of pinggy.

13. Running JSONGrapher-web locally during development

Simply opening index.html will not allow you to use JSONGrapher during development.
To run JSONGrapher locally, the easiest way is the following:

1. Open a terminal. Change the directory to where the JSONGrapher source code is located on your computer.
2. Run the following command (leave the terminal open):
   python -m http.server 8000
3. Go to this location in a web browser (you can click):
   http://localhost:8000/
    a. The link should open JSONGrapher automatically if you are in the correct directory. Otherwise you can navigate to the correct directory. This will allow you to test JSONGrapher locally.
    b. You can also use relative paths like: http://localhost:8000/other_html/ModelSimulationTesters/https_call_tester.html

14. 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.

15. 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.

16. License

17. Credits

JSON Grapher utilizes plotly, UUC, AJV, MathJS, 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
Aditya Savara added advanced features such as 3D plotting, color gradients (colorscales), and the symbolic expressions (equations) module.
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.