The Journal of Open Source Software

widgyts: Custom Jupyter Widgets for Interactive Data Exploration with yt

Madicken Munk[1] **and Matthew J. Turk**[1]

1 National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. 1205 W Clark St, Urbana, IL USA 61801

DOI: 10.21105/joss.01774

Software

- Review
- Repository
- Archive

Editor: Lorena Pantano

Reviewers:

- @harpolea
- @KayleighRutherford

Submitted: 18 September 2019
Published: 29 January 2020

Summary

`widgyts` is a custom Jupyter widget library to assist in interactive data visualization and exploration with `yt`. `yt` (Turk et al., 2011) is a python package designed to read, process, and visualize multidimensional scientific data. `yt` allows users to ingest and visualize data from a variety of scientific domains with a nearly identical set of commands. Often, these datasets are large, sparse, complex, and located remotely. Creating a publication-quality figure of an area of interest for this data may take numerous exploratory visualizations and subsequent parameter-tuning events. The `widgyts` package allows for interactive exploratory visualization with `yt`, enabling users to more readily determine which parameters and selections they need to best display their data.

The `widgyts` package is built on the `ipywidgets` (Grout, Frederic, Corlay, & Ragan-Kelley, 2019) framework, which allows `yt` users to browse their data using a Jupyter notebook or a Jupyterlab instance. `widgyts` is developed on GitHub in the Data Exploration Lab organization. Issues, questions, new feature requests, and any other relevant discussion can be found at the source code repository (Munk & Turk, 2019).

Motivation

Data visualization and manipulation are integral to scientific discovery. A scientist may slice and pan through various regions of a dataset before finding a region they wish to share with colleagues. These events may also require shifting colormap settings, like the scale, type, or bounds, before the features are highlighted to effectively convey a message. Each of these interactions will require a new image to be calculated and displayed.

A number of packages in the python ecosystem use interactivity to help users parameter-tune their visualizations. Matplotlib (Hunter, 2007) and ITK (Ibanez et al., 2019) have custom widgets build on the ipywidgets framework (Corlay, Silvester, et al., 2019; McCormick et al., 2020) that act as supplements to their plots. This is the principle that widgyts follows as well. Other libraries like Bokeh (Bokeh Development Team, 2019) distribute interactive javascript-backed widgets. Other frameworks like bqplot (Corlay, Sunkara, et al., 2019) have every plot returned with interactive features. The packages named here are by no means comprehensive; the python ecosystem is rich with interactive tools for visualization. However, it is illustrative of the need and investment in interactivity for the visualization community.

A common user case for visualization is to have data stored remotely on a server and some interface with which to interact with the data over the web. Because every plot interaction requires a new image calculation, this may result in significant data transfer needs. For this case, a request is sent to the server, which calculates the images, with every new plot

interaction. When the request is sent the server calculates a new image, serializes it, and the image is sent back to the client. The total time to generate one image can generally be expressed as T_{server}, where

$$t_{\text{server}} = t_{\text{request}} + t_{\text{image calc, server}} + t_{\text{pull, image}} + t_{\text{display}}.$$

The total compute time spent on image generation is $T_{\text{server}} = n * t_{\text{server}}$, where n is the number of interactions with the figure.

`widgyts` modifies this process by shifting image calculation to occur client-side in the browser. Rather than image serialization and calculation happening on a remote server, a portion of the original data is uploaded into the `WebAssembly` backend of `widgyts`. The time to calculate image client-side can be expressed as:

$$t_{\text{client}} = t_{\text{request}} + t_{\text{pull,data}} + t_{\text{image calc, client}} + t_{\text{display}}.$$

Subsequent interactions (n) with the image only affect the final two terms of the equation, so

$$T_{\text{client}} = t_{\text{request}} + t_{\text{pull,data}} + n * [t_{\text{image calc, client}} + t_{\text{display}}].$$

Thus, this becomes advantageous as

$$T_{\text{client}} < T_{\text{server}}$$

or

$$n * t_{\text{image calc, client}} + t_{\text{pull, data}} < n * [t_{\text{image calc, server}} + t_{\text{pull, image}}].$$

The time to pull an image or data is dependent on the data size and the transfer rate. T_{client} will be lower than T_{server} as the number of interactions n grows, as the size of the image ($\text{data}_{\text{image}}$) grows, and as the time to calculate the image on the client $t_{\text{image calc, client}}$ decreases.

Moving image calculation to the client requires a large initial cost of transferring a portion of the original data to the client, which may be substantially larger than the size of a single image. However, a dataset with sparse regions will be more efficient to transfer to the client and subsequently calculate and pixelize there. Pixelizing a dataset with large, sparse regions of low resolution, such as one calculated from an adaptive mesh, with a fixed higher resolution will require recalculating and sending pixel values for a region that may only be represented by a single value. Thus, for certain data representations this methodology also becomes advantageous.

The WebAssembly Backend

To allow for efficient data loading into the browser we chose to use Rust compiled to WebAssembly. The WebAssembly backing of `widgyts` allows for binary, zero-copy storage of loaded data on the client side, and WebAssembly has been designed to interface well with JavaScript. Further, the primitive structure of WebAssembly reduces the time to calculate the image in the browser, thus reducing the time to calculate the image client-side. Finally, WebAssembly is executed in a sandboxed environment separate from other processes and is memory safe. At the time of writing, widgyts is the only webassembly-native backed visualization widget in the python ecosystem.

While yt can access data at an arbitrary location within the dataset, widgyts is structured to access any data within a 2D slice. Thus, only a slice of the data is uploaded client-side, not the entire dataset. For the large, sparse datasets that widgyts has been designed for, it would be infeasible to upload the entire dataset into the browser. A new slice in the third dimension will require an additional data upload from the server. Therefore, not all exploration of the dataset can be performed exclusively client-side.

Results

The following image is a simple timing comparison between using yt with the Jupyter widgets package ipywidgets and using the widgyts package on the same dataset. The dataset is the IsolatedGalaxy dataset; a commonly used example in the yt documentation consisting of a galaxy simulation perfomed on an adaptive mesh. The dataset has variable resolution and is sparse near the domain boundaries. A notebook is included in the widgyts repository (Munk & Turk, 2019) for one interested in replicating this analysis.

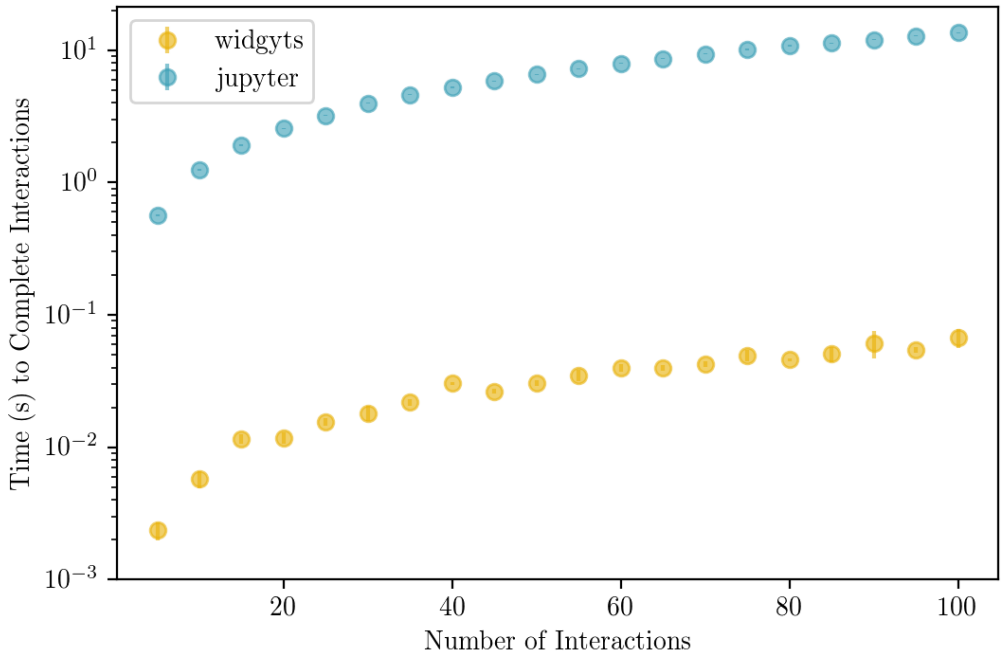

Figure 1: A timing comparison between using ipywidgets with yt and widgyts on the IsolatedGalaxy dataset distributed with yt. The image generated by each tool is 512x512 pixels. Each timing point is based on a number of panning interactions in x, averaged over 10 measurements. The data points are accompanied by a 95% confidence interval.

While widgyts outperforms the implementation of ipywidgets with yt that we wrote for this paper, there are a number of factors that may affect these timing results beyond loading the data in the browser. In the Jupyter widgets implementation we are using yt functionality to recalculate image and convert it into a .png, which is being done in WebAssembly in widgyts. This is the most extreme example of reducing the time to compute in the browser, because almost no computation is being performed browser-side in the ipywidgets case. This calculation was performed locally, so while the data is being transfered continuously to the browser with the Jupyter widgets implementation, the timing results may become more disparate with a slower data transfer time from a remote server. Other packages with custom

tools for interactivity may be faster than the naive implementation with `ipywidgets` that we've included here. However, these results remain an illustrative example that loading data into the browser and performing image recalculation in the browser is advantageous.

Conclusions

In this paper we introduced `widgyts`, a custom widget library to interactively visualize and explore data with `yt`. `widgyts` makes large, sparse, data exploration accessible by passing data to the browser with WebAssembly, allowing for image generation to occur client-side. As the number of interactions from the user increases and as datasets vary in sparsity, `widgyts`' features will allow for faster responsiveness. This will reduce the use of expensive compute resources (like those of a lab or campus cluster) and move parameter-tuning events to a local machine.

Acknowledgements

We would like to acknowledge the contributions to this project from other developers, including Nathanael Claussen, Kacper Kowalik, and Vasu Chaudhary. This work was supported by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4561 (MJT).

References

Bokeh Development Team. (2019). *Bokeh: Python library for interactive visualization*. Bokeh. Retrieved from https://bokeh.org/

Corlay, S., Silvester, S., martinRenou, Boone, K., Caswell, T. A., Nielsen, J. H., Sundell, E., et al. (2019, November). Jupyter-matplotlib. Matplotlib Developers. Retrieved from https://github.com/matplotlib/jupyter-matplotlib

Corlay, S., Sunkara, S., Madeka, D., Menegaux, R., Cherukuri, C., Grout, J., & Mabille, J. (2019, November). Bqplot: Plotting for jupyter. Bloomberg. Retrieved from https://github.com/bloomberg/bqplot

Grout, J., Frederic, J., Corlay, S., & Ragan-Kelley, M. (2019, September). Ipywidgets. Jupyter Widgets. Retrieved from https://github.com/jupyter-widgets/ipywidgets

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55

Ibanez, L., Lorensen, B., McCormick, M., King, B., Blezek, D., Johnson, H., Lowekamp, B., et al. (2019). *InsightSoftwareConsortium/itk: ITK 5.0.1*. Zenodo. doi:10.5281/zenodo.3351620

McCormick, M., Musy, M., Mader, K., Chen, D., Sullivan, B., Chandradevan, R., Fillion-Robin, J.-C., et al. (2020). *InsightSoftwareConsortium/itkwidgets: itkwidgets 0.24.2*. Zenodo. doi:10.5281/zenodo.3603359

Munk, M., & Turk, M. J. (2019, July). Widgyts. Data Exploration Lab. Retrieved from https://github.com/data-exp-lab/widgyts

Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., & Norman, M. L. (2011). yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement Series*, *192*, 9. doi:10.1088/0067-0049/192/1/9