Use of the Julia Programming Language for the Analysis of TESS Data

John Kielkopf[1,2]

U. of Louisville[1] – U. of Southern Queensland[2]

Overview

The high performance Julia Programming Language is an open source project offering pre-built binaries that run on most commonly used platforms. With similarities to Python and Matlab that are often used for astrophysical data analysis, it is easy for experienced students and programmers to learn, and to adapt existing code, or to import code from other languages using its native tools. Although comprehensive libraries similar to astropy are not available, essential ones such as FITS file procedures, are supported. We have experimented with Julia for the analysis of TESS data in cases where the speed of Python is limiting: transit modeling and fitting, and processing of time series stacks of TESS Full Frame Images. This poster highlights the features of Julia programming, contrasts those against the familiar Python and Matlab languages, and compares the outcomes for test cases. These results demonstrate a significant decrease in processing time with Julia for large data arrays, especially when optimized for multiprocessing.

Julia Language

Julia is a young and developing open source computing language designed for high performance at its foundation. [1 – 3] While not yet common in astronomy and astrophysics research, it has a growing community of users and developers, especially in engineering, finance and pharmaceuticals where data science, simulations, and machine learning press the limitations of available computing resources [4]. In use it has the ease of development we find in Python, with the speed of execution that comes from multitasking and GPU processing associated with the high performance computing in C and Fortran. Highlights of its features are

- It's very fast, running native code through LLVM
- It uses a *just in time* compiler and may be used interactively
- Pre-built binaries are available for free for Windows, MacOS, and Linux
- Other packages, installed from within Julia, provide a consistent environment across operating system and hardware landscapes
- It is easily learned interactively with effective documentation and online forums for support

Julia's core includes components essential for applications in physics and astronomy. For example, the structures we use from Numpy in Python are inherent in the arrays that are fundamental to Julia. Broadcast operations on arrays are performed quickly, and may be programmed more simply in the style of Julia that resembles Matlab than with the wrappers that are required by Python. However Julia may run as fast or faster with loops because its compiler creates optimized code, and for loops may be dispatched with threading across multiple cores of a single machine with the introduction of a simple macro into the code. Julia may also incorporate code from C, Fortran, and Python so that it can provide a nearly universal platform that almost "does it all," from prototyping to production. There is a foundation of key packages too, including

- JuliaAstro has astronomical coordinate systems and time keeping, FITSIO wrapping libcfitsio, WCS wrapping libwcs, ERFA wrapping liberfa, Earth's orientation from IERS tables, JPL ephemerides, translations of the astronomical utilities used in IDL, photometry, and Lomb-Scargle periodograms
- Plots enables complex dynamic visualization simply, and providing a choice of backends including Plotly that will be familiar to Python programmers

While the format of Julia code will be familiar to users of Matlab, Mathematica, or Python, it has its unique features to learn, stumble over, and re-learn (see Avik Sengupta, Julia High Performance Computing [5])

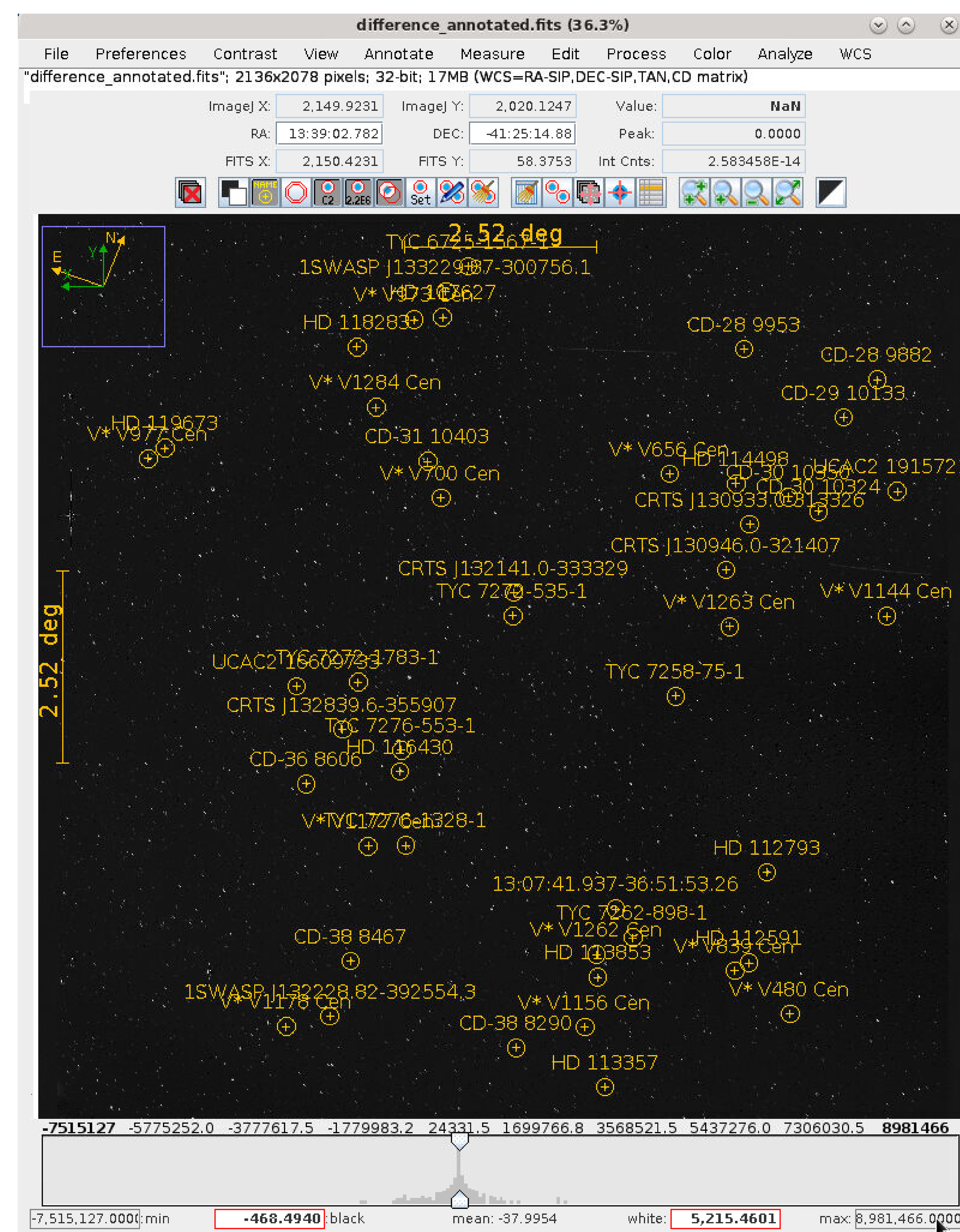
- Indexing is 1-based, not 0-based
- Typing is dynamic
- Julia favors many small functions which the compiler will optimize and insert in code in real time
- Scoping rules discourage use of global variables when speed is important

Our use of Julia developed from an interest in exploring the TESS full frame images (FFIs) as they became available. The TESS Science Processing Operating Center (SPOC) provides FFI data through MAST within a month of download from the satellite, and they extract short period exoplanet candidates for the followup community to study. The FFIs have other data of course, especially on stellar variability due to pulsations and rotations, stellar activity associated with flares and space weather, eclipsing binary stars of all kinds, solar system objects, and transient events in our galaxy and beyond. They are also ultimately a powerful resource for training students in the analysis of large astronomical data. Given the size of the FFI data sets, we found that Python routines were slow to complete and limited the science that may be possible. One solution was to move processing to a small cluster system in our department which provides 192 GB of memory in a node, up to 720 cores across all nodes, 2 NVIDIA Titan X GPUs, and 100 TB of primary storage that is shared by several users. Our group also maintains a server specifically for processing TESS data that has 128 GB of memory, 32 TB of online storage, 20 physical cores, and a Tesla K20 GPU. Its compute capacity is underutilized with Python and could be fully exploited with Julia.

Thanks to the recent introduction of TICA FFIs through MAST (see [6]) and the use of parallel processing downloads on the local cluster system, we have access to the FFIs for a sector within days of their transmission from the satellite. There is potential for rapid review of them for targets specifically of interest to our students that would inform ground based followup for photometry and spectroscopy while the targets are still visible for most of the night.

We are making progress in learning how to use Julia toward this end, supplemented with existing Python code, and greatly enabled by AstroImageJ for stack visualization, target identification, photometry, and light curve modeling [7]. A few examples are illustrated here, drawn from Sector 39 data, and analyzed with code during development.

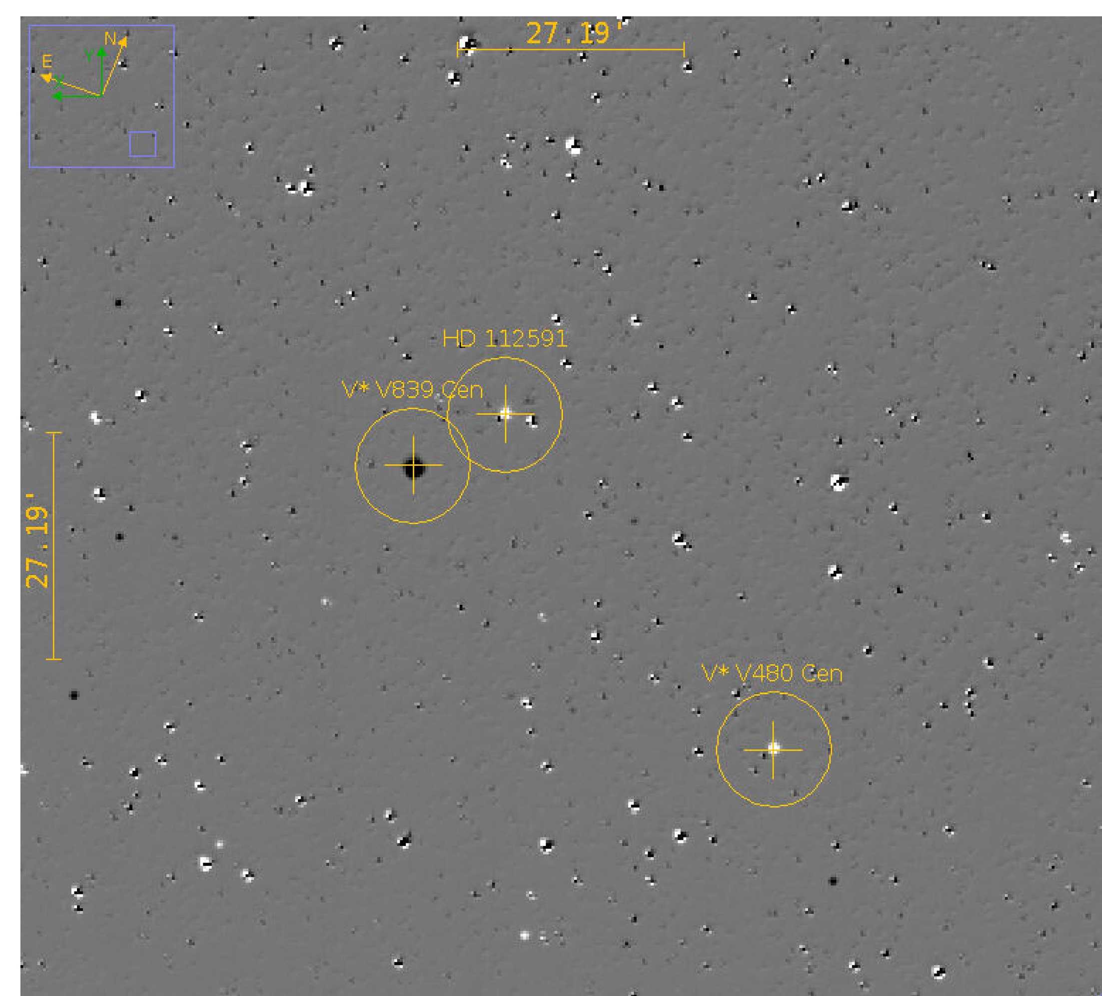
Examples of Julia Applied to TESS Data



AstroImageJ offers interactive visualization and analysis of TESS FFIs. This screen is from Sector 39, Camera 1, CCD 2. The image is the difference (mean minus median) of 1024 sequential images from orbit 1. The mean, median, and difference are computed with one run of a Julia program, in this case using 10 cores of the cluster with the image stack in memory. For each pixel (indexed by $[i,j]$) the code calculates the median over all images at once in the stack (the sequential, first, axis in storage for Julia)

$$median_image[i,j] = median(images[:,i,j])$$

The program loops over columns with threads that dispatch multiple CPUs to run the loops over rows. Most of the processing time is used to load the images into memory, and in optimal work once loaded they would be used for more than one outcome. In this case, to produce the mean, median and difference of all 1024 images took 10.1 seconds for the median and 3.2 seconds for the mean. A similar Python code using broadcasting took 4 minutes 5 seconds of CPU time. Swarp, compiled in C, took, 3 minutes and 50 seconds. Stars are identified as possibly variable in the stack difference by their appearance as either bright or dark. Smaller variations add a background which masks detection of some variable stars. Asteroids make streaks, the faster they go (likely closer to Earth) the longer the streak. Star designations come from Simbad through the AstroImageJ functions.



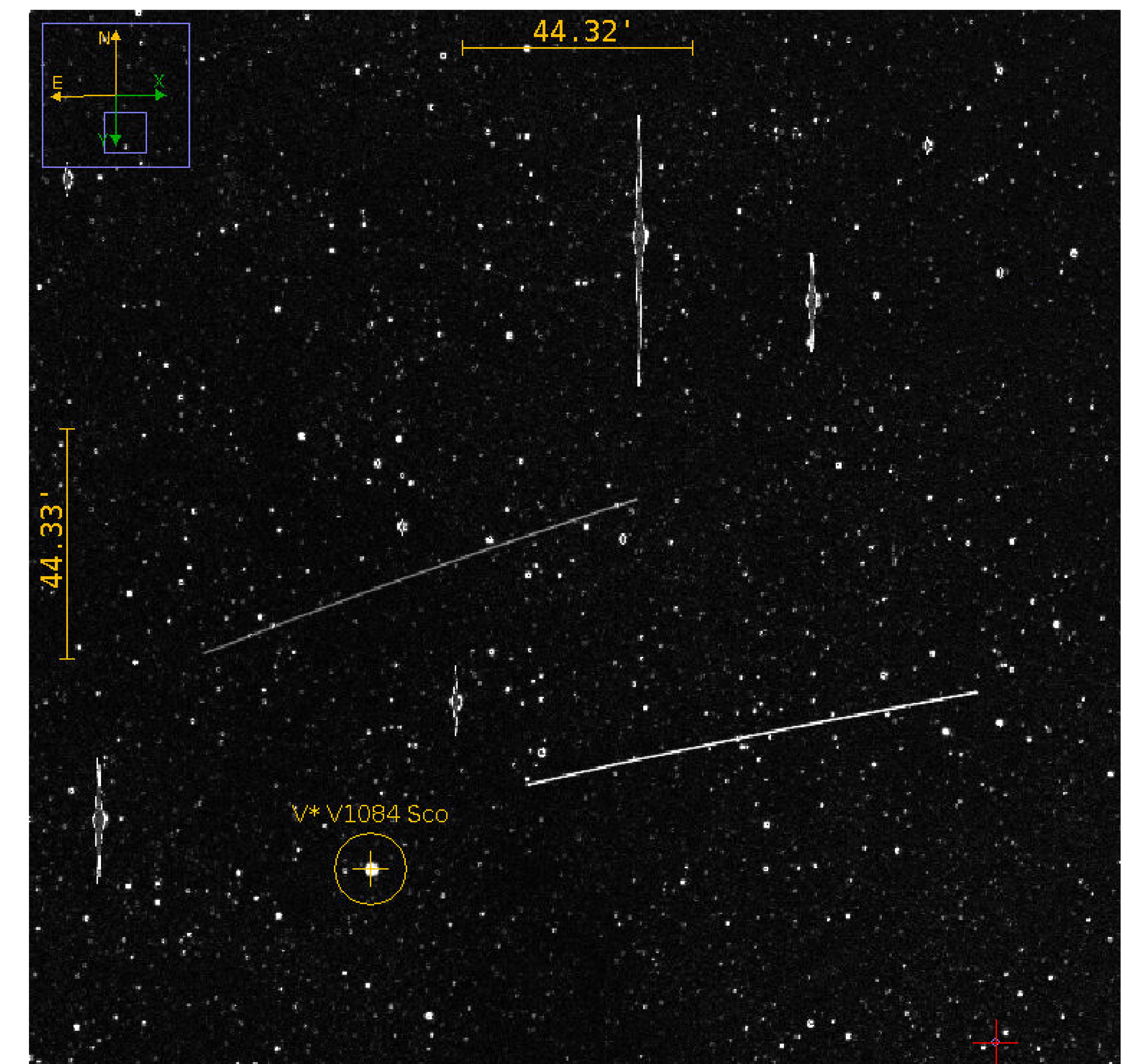
A closer look at the difference image for the stack shows the distinctive character of variable star images, and the background "noise" in stars that are not strongly variable. Both well-known and previously unsuspected variables appear in large numbers for each camera and CCD image stack. Short period variations over one orbit are most distinctive.

Alternatively, the stack may be processed to do a fast Fourier transform (FFT) for all pixels in the time domain when the images are evenly cadenced. FFTs are done in Julia with the FFTW library and are very efficiently computed without adding special handling other than the usual methods of optimizing for high performance. We select a sequence of images (typically 512 or 1024), inspect them for quality, and replace poor frames with null ones. For a data cube "images", the FFT is performed in one line of code

$$ffts = abs.(rfft(images, 1))$$

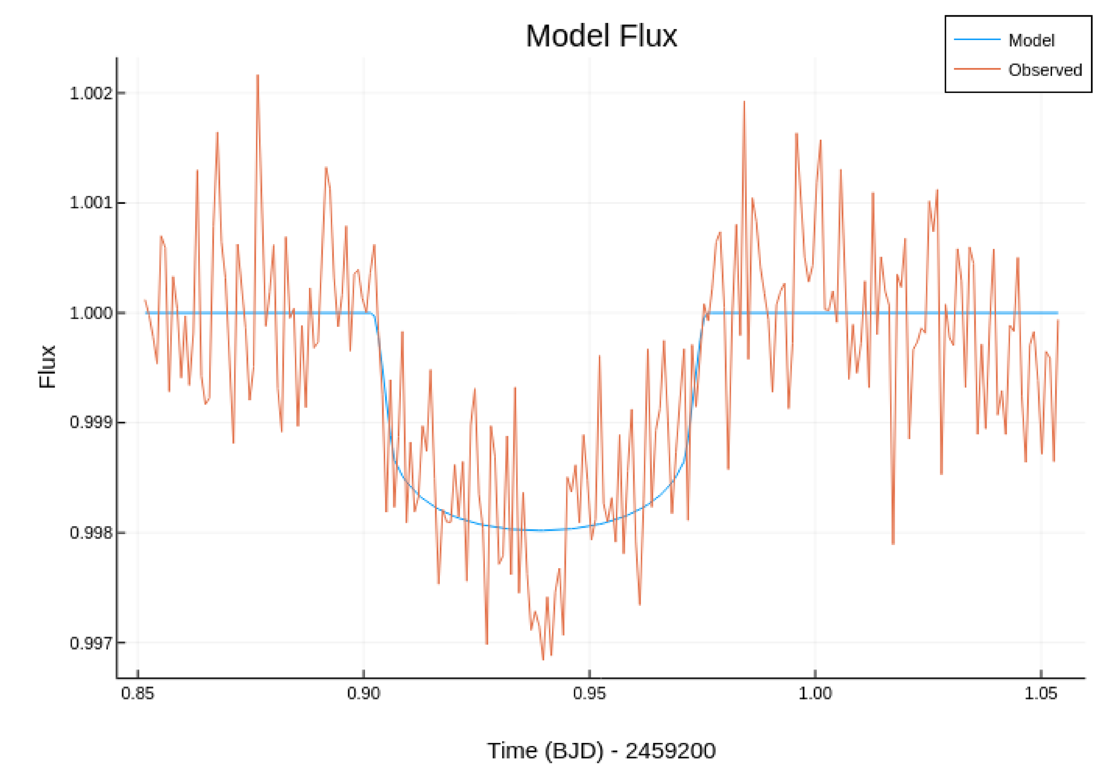
which returns another data cube ("ffts") with the transform along the first axis, that is the time series axis in this case. Each frame of the new cube is a frequency slice of the time dependent cube.

Time Dependence



This a slice of a FFT of 1024 images from orbit 1 of sector 39, camera 1, ccd 3. The slice has been chosen to show the eclipsing binary star V1084 Scorpii at its brightest in the frequency space, that is, at the fundamental frequency of its periodicity. Typically short period eclipsing binaries (EBs) show recognizable maxima in their Fourier transforms because of the strong modulation of their light curves. This method is very effective at finding EBs in TESS FFIs. While Python can generate FFT stacks, its processing has been prohibitively slow.

As another example of Julia's potential for TESS data analysis we consider light curve modeling using the Mandel and Agol protocol [8 – 9, 7]. The analytical formulation is widely used and potentially fast for Markov chain Monte Carlo (MCMC) methods of fitting. It is also used in AstroImageJ to fit light curves to photometry of transiting planets in real time. Our implementation in Julia has all of the orbital elements and in principle allows fitting on any parameter. While not yet optimized, it reproduces the light curves from AstroimageJ



The observed light curve is from transit observations of a TESS planet candidate taken in December 2020 with our Mt. Lemmon telescope. The model curve, computed with Julia, is based on parameters from the AstroImageJ fit that produce the same fitted curve. The fit was done without adjusting limb darkening coefficients, and does not allow for stellar surface structure or even orbital eccentricity, though those parameters are in the code. Validation of the transitmodel code and its application to explore more fully multiparameter fitting of transit photometry and radial velocities is continuing.

References and Followup

1. https://julialang.org/
2. https://www.nature.com/articles/d41586-019-02310-3
3. https://en.wikipedia.org/wiki/Julia(programminglanguage)
4. Emmett Boudreau, *What You Need to Know about Julia in 2021*, Towards Data Science (May 11, 2021)
5. Avik Sengupta, *Julia High Performance Computing*, Packt (2019)
6. Michael Fausnaugh, et al., *Calibrated Full-frame Images for the TESS Quick Look Pipeline*, Research Notes of the American Astronomical Society 4 251 (2020)
7. Karen Collins, et al., *AstroImageJ: Image Processing and Photometric Extraction for Ultra-Precise Astronomical Light Curves*, Astronomical Journal 153 77 (2017)
8. Kaisey Mandel and Eric Agol, *Analytic Light Curves for Planetary Transit Searches*, Astrophysical Journal 580 L171 (2002)
9. Avi Shporer, Matlab version *matransit.m* from 2015

If you would like to explore Julia online, or download the compiled binary and try it on your own computer, this is the place to start

https://julialang.org/ **The Julia Programming Language website**

A helpful Wikibook may be your guide

Introducing Julia

A collection of Julia code for astronomy is on github

JuliaAstro

kielkopf@louisville.edu