The Stata Journal (*yyyy*) *vv*, Number *ii*, pp. 1–27

Julia as a universal platform for statistical software development

David Roodman
Open Philanthropy
San Francisco, CA
david.roodman@openphilanthropy.org

Abstract. Like Python and Java, which are integrated into Stata, Julia is a free programming language that runs on all major operating systems. The `julia` package links Stata to Julia as well. Users can transfer data between Stata and Julia at high speed, issue Julia commands from Stata to analyze and plot, and pass results back to Stata. Julia's econometric software ecosystem is not as mature as Stata's or R's, or even Python's. But Julia is an excellent environment for developing high-performance numerical applications, which can then be called from many platforms. The `boottest` program for wild bootstrap–based inference (Roodman et al. 2019) can call a Julia back end for a 33–50% speed-up, even as the R package `fwildclusterboot` (Fischer and Roodman 2021) uses the same back end for inference after instrumental variables estimation. `reghdfejl` mimics `reghdfe` (Correia 2016) in fitting linear models with high-dimensional fixed effects but calls an independently developed Julia package for tenfold acceleration on hard problems. `reghdfejl` also supports nonlinear models—preliminarily, as the Julia package for that purpose matures.

NOTE: This article is formatted for submission to the Stata Journal. It has not been accepted.

Keywords: st0001, reghdfe, reghdfejl, boottest, Julia, high-dimensional fixed effects, cross-platform communication

1 Introduction

Whenever you issue a command to perform a statistical calculation, you initiate a chain of events of bewildering complexity: your request cascades through layers of code crafted over decades by thousands of programmers in multiple languages, as it is expanded into millions or billions of computational steps—steps that are executed, if you're lucky, in the time it takes to read this sentence.

In Stata, for example, a call to `regress` or `gsem` starts out in the ado programming language. The ado code translates your command line into a digital problem definition and passes that to routines written in C/C++ or Stata's built-in Mata language. In modern Stata, an ado program can even call Python or Java. The called code will probably invoke the LAPACK library, which is written in Fortran, to invert matrices or solve linear systems.

Here I introduce a package that forges a fresh link between Stata and another language, Julia. Julia is new: it reached version 1.0 in 2018. It is also free, and available for Windows, Linux, and macOS. Like Python code, Julia code is fully portable across those platforms and can run as soon as downloaded.

For creators of computationally intensive software, Julia has many strengths. More so than with Stata (but less so than with Python), if you need a tool to perform a fundamental task such as exploiting GPUs, there's a good chance someone has written and posted it. Julia's creators set out to solve the "two-language problem," the inefficiency that arises when programmers have to split a project between high- and low-level languages as they trade off ease of software development and maintenance for speed of execution. (Python and C++ is a common combination, as is R and C++.) Julia resembles Python in syntax, but was built from the start for just-in-time compilation, meaning that it translates Julia functions to machine language when they are first called, and stores that code for reuse. Like LISP, a language almost as old as Fortran, Julia can treat code as data. This feature opens the door to meta-programs that can, for example, automatically optimize one's code in certain ways by rewriting it before compilation, or generate new code to take derivatives of complex calculations already implemented.

Julia has disadvantages. While the Julia ecosystem contains the foundation for statistical work—packages to manage data sets, fit standard models, and generate plots—many packages are immature or poorly documented. Standards for passing results are less well developed than in Stata. Much less statistical functionality is available in Julia than in Stata or R, or even Python.

Since Julia is a great home for implementing and optimizing numerical software, but not (yet) a great home for the end user, a sweet spot for the new language is back end development. Methods of estimation and inference can be implemented in Julia, and called from Stata, R, and other platforms. The core work can implemented, optimized, and augmented in one place, rather than separately on each platform.

The `julia` package links Stata to Julia. It follows in the footsteps of many projects to link statistical software to other languages. StataCorp introduced the Stata plugin interface (SPI) for C/C++ in Stata 8.1, in 2003. It added the built-in Mata language in Stata 9. Fiedler (2012) envisioned running Python inside of Stata. Following the suggestion of a StataCorp employee, Fiedler (2013) partly realized that vision by writing a plugin for Stata. Perhaps inspired by Fiedler, Stata 16 fully integrated Python. Java plugins became possible in Stata 13 and the Java prompt debuted in the Stata results window in version 17. Haghish (2019) introduced `rcall`, which allows one to issue R commands from Stata; it passes content between the two systems by saving files to disk. Outside of Stata, the JuliaCall package (Li 2019) lets R users call Julia routines. An identically named package does the same for Python, and is paired with PyCall for Julia.[1]

The `julia` package was most inspired by Fiedler (2013). Its core too is a C++ plugin, which comes precompiled for Windows, Linux, macOS/Intel, and macOS/ARM. From the user's point of view, the package has three components:

- The `jl:` command for Stata, which prefixes single Julia lines or, when typed by itself (with or without the colon), starts a Julia session in Stata.
- `jl` subcommands for managing Julia packages and transferring data between Stata and Julia.
- Julia functions to read and write Stata macros, scalars, matrices, and variables.

The package is designed to hide the complexities of the Stata-Julia linkage, so that getting started requires just two quick steps, installing Julia and installing the `julia` package.

Two user-written Stata packages call Julia via `julia`. The `boottest` program for wild bootstrap–based inference (Roodman et al. 2019) now accepts a `julia` option, which causes it to use my Julia package WildBootTests.jl rather than the Mata back end. And the new **`reghdfejl`** mimics **`reghdfe`** and **`ivreghdfe`**, programs for fitting linear models with high-dimensional fixed effects (HDFE) (Correia 2016), while calling FixedEffectModels.jl, whose lead developer is Matthieu Gomez. **`reghdfejl`** also has the ability, in beta, to fit nonlinear HDFE models, such as the logit and negative binomial, via Johannes Boehm's GLFixedEffectModels.jl. In the last case, using Julia brings new functionality to Stata. In all cases, calling Julia cuts runtime on hard problems, sometimes by an order of magnitude.

Stata users can thus benefit from Julia and `julia` without using them directly. Still, this paper explains the new `julia` package. So it starts with a hands-on demonstration of running Julia from Stata, in section 2. Then section 3 demonstrates applications: it introduces **`reghdfejl`** and the Julia back end for `boottest`; and it demonstrates graphing via Julia. Section 4 gives tips and warns of some limitations. Section 5 more thoroughly describes the package's Stata commands and Julia functions.

1. See juliapy.github.io/PythonCall.jl/stable.

2 Hands-on introduction

2.1 Installation and first examples

Julia is an interpreted language in the sense that you can only run programs written in it if you have installed it.[2] In contrast, compiled C++ and Fortran programs run on computers without a C++ or Fortran compiler.

The preferred way to get Julia is to download the `juliaup` installation manager. You do that in Windows by installing "Julia" (not "juliaup") from the Windows Store.[3] In Linux and macOS, you can instead start Stata and type

```
! curl -fsSL https://install.julialang.org | sh -s -- -y
```

(including the "`!`" at the beginning). In Linux, you should then restart Stata and the shell to trigger the addition of Julia's location to the PATH environment variable.

To install the `julia` package in Stata, type:

```
ssc install julia
```

You can now call Julia from Stata:

```
. jl: "Hello world!"
 Starting Julia
 "Hello world!"
```

The output may take several minutes to appear. On first use, `jl` installs and precompiles two Julia packages, DataFrames.jl and CategoricalArrays.jl, along with all the packages *they* depend on. DataFrames.jl (Bouchet-Valat and Kamiński 2023) is the dominant Julia framework for data sets—i.e., rectangular grids whose columns have names and potentially different data types. CategoricalArrays.jl implements the Julia equivalent of Stata variables with data labels—ones coded, for example, so that 0 means "black" and 1 means "white." Both packages are hosted on GitHub and are maintained by users, who accept improvements from others.

In general when using Julia, you will experience delays at four points: when packages are installed; when they are first used; when they are first used in a Julia session; and when particular features within them are first used, such as clustering standard errors. A thrust of recent development in the Julia langauge has been to move the behind-the-scenes processing from later to earlier stages in order to shorten the lags during regular use.

In the session shown below, I load Stata's venerable "auto" data set, extract five variables, run a regression on them, and then copy the data to a Julia DataFrame named `auto`:

```
. sysuse auto
(1978 automobile data)

. keep price headroom foreign mpg turn
```

2. PackageCompiler.jl can build stand-alone Julia apps, but the files take hundreds of megabytes because they contain complete copies of Julia and its libraries.
3. The instructions might change. See github.com/JuliaLang/juliaup#installation for the latest.

```
. reg price headroom foreign#c.mpg, cluster(turn)
Linear regression                               Number of obs     =         74
                                                F(3, 17)          =       9.69
                                                Prob > F          =     0.0019
                                                R-squared         =     0.2939
                                                Root MSE          =       2531
                                   (Std. err. adjusted for 18 clusters in turn)
--------------------------------------------------------------------------------
               |               Robust
         price | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
---------------+----------------------------------------------------------------
      headroom |   -273.2403   274.4352    -1.00   0.333     -852.248    305.7674
               |
 foreign#c.mpg |
     Domestic  |   -344.0717   72.40801    -4.75   0.000    -496.8392   -191.3041
      Foreign  |   -267.0448   57.09596    -4.68   0.000    -387.5067   -146.5828
               |
         _cons |    13743.64     2176.8     6.31   0.000      9150.99    18336.28
--------------------------------------------------------------------------------
. jl save auto
Data saved to DataFrame auto in Julia
```

`jl save` is not a Julia command. It is a subcommand of the Stata command `jl`, which is part of the `julia` package. If an object named `auto` already exists in the Julia environment, it is overwritten.

Next I type `jl` by itself, which changes the prompt in much the same way that typing `mata` or `python` or `java` does. Then I display the new Julia DataFrame by typing its name:

```
. jl
------------------------ Julia (type exit() to exit) ------------------------
jl> . auto
74×5 DataFrame
 Row │ price   mpg     headroom  turn    foreign
     │ Int16?  Int16?  Float32?  Int16?  Cat…?
─────┼───────────────────────────────────────────────────────────────────────
   1 │   4099      22       2.5      40  Domestic
   2 │   4749      17       3.0      40  Domestic
   3 │   3799      22       3.0      35  Domestic
   4 │   4816      20       4.5      40  Domestic
   5 │   7827      15       4.0      43  Domestic
   6 │   5788      18       4.0      43  Domestic
   7 │   4453      26       3.0      34  Domestic
   8 │   5189      20       2.0      42  Domestic
     │
  68 │   3748      31       3.0      35  Foreign
  69 │   5719      18       2.0      36  Foreign
  70 │   7140      23       2.5      36  Foreign
  71 │   5397      41       3.0      35  Foreign
  72 │   4697      25       3.0      35  Foreign
  73 │   6850      25       2.0      36  Foreign
  74 │  11995      17       2.5      37  Foreign
                                        59 rows omitted
```

Along the top are the column names, and below them the types. `Int16` means a 16-bit integer, equivalent to Stata `int`. `Float32` corresponds to Stata `float`. The question marks at the ends of the type names indicate that the types have been modified to allow the special value `missing`. By default, `jl save` made all the columns accept missing. `jl save` also converted the `foreign` variable, which is coded as 0/1 in Stata and has value labels "Domestic" and "Foreign," to a `CategoricalVector`. While the Julia version of the variable also has an internal, integer representation, the vector presents as a text variable.[4] The type label "CategoricalVector?" does not fit in the column head, so it is shortened to "Cat...?".

Next I run the same regression in Julia, with the `reg()` function from the Julia package FixedEffectModels.jl:

```
jl> . using FixedEffectModels

jl> . reg(auto, term(:price) ~ term(:headroom) + term(:foreign) & term(:mpg),
  ...     cov.cluster(:turn))
FixedEffectModel
====================================================================================
Number of obs:                        74   Converged:                           true
dof (model):                           3   dof (residuals):                       16
R²:                                0.294   R² adjusted:                        0.264
F-statistic:                      9.6942   P-value:                            0.001
====================================================================================
                          Estimate  Std. Error    t-stat  Pr(>|t|)  Lower 95%  Upper 95%
------------------------------------------------------------------------------------
headroom                  -273.24      274.435  -0.995646    0.3342   -855.017    308.536
foreign: Domestic & mpg   -344.072      72.408  -4.75185     0.0002   -497.57    -190.574
foreign: Foreign & mpg    -267.045      57.096  -4.67712     0.0003   -388.083   -146.007
(Intercept)              13743.6      2176.8     6.31369     <1e-04   9129.03   18358.2
====================================================================================
```

The Stata and Julia results match. But that Julia regression command needs explaining. First some cosmetic comments:

- Because the regression command was long, I broke it in two. After I hit the enter key at the end of the first line, `jl` detected that the line was not grammatical by itself and printed "`...`" to prompt me to continue.
- Stata has the `#` and `##` interaction operators. The corresponding symbols in Julia are `&` and `*`. The first appears here.
- All variable names in the command line are prefixed with "`:`". That indicates that the names are *symbols*, not stand-alone Julia objects like the DataFrame `auto`.
- Because `foreign` is a `CategoricalVector`, `reg()` automatically treats it as a factor variable, generating a pair of dummies to be interacted with `mpg`. There is no need for an "`i.`" prefix or the equivalent as in Stata. However, treating non-`CategoricalVectors` as factor variables requires a more awkward syntax. One must pass `reg()` an argument with a dictionary mapping the variables to coding types, such as with `contrasts=Dict(:headroom=>DummyCoding())`.

The command line contains deeper complexity. In Julia, as in R and Python, every regression command is a statement in a full-fledged programming language. This tends

4. Julia categorical variables are not quite comparable to Stata variables with data labels. Code involving the Julia variable refers to display values, such as "Domestic," not to numerical codes.

to complicate the syntax for even basic operations. It is hard to implement an R or Python line as clean as `regress y x`. In the Julia command above, `term(:price)`, `term(:headroom)`, ..., etc., create instances of the object type `Term` to represent variables within a statistical model. The `Term` structure is defined in the StatsModels.jl package, which FixedEffectModels.jl loads and surfaces for the user. The content of each `Term` is just a symbol, such as `:price`. What give `Term`'s life are the operations that StatsModels.jl defines *on* them. The `+` operator is repurposed ("overloaded") to form tuples of `Term`'s—pairs, triplets, and so on. The ∼ operator binds together two `Term` tuples to make a `FormulaTerm`, a structure whose distinctive feature is having designated left and right sides. The entire `FormulaTerm` is passed as the second argument to `reg()`. In this way, the primitives of a flexible programming language are composed to communicate a statistical model.

This standard for representing regression models is not intrinsic to Julia. It has emerged from the work of users. And it is transparent. In Stata, one could type `jl: dump(term(:c) ∼ term(:a) + term(:b))` to reveal the contents of a `FormulaTerm`. The software that builds and applies these structures is on GitHub, all written in Julia. The StatsModels.jl formula language is also extensible[5]. As we will see, FixedEffectModels.jl defines a way to mark variables whose fixed effects are to be absorbed.

Yet a powerful feature of Julia makes it practical to ignore most of this complexity. The regression command above can be simplified with a *macro*. Where macros in Stata are strings, in Julia they are code for manipulating code—meta-programs. StatsModels.jl defines `@formula`, which lets one write models in a more R-like way. Thus:

```
jl> . m=reg(auto, @formula(price ~ headroom + foreign & mpg), Vcov.cluster(:turn))
[identical output omitted]
```

Before this new command is compiled and executed, `@formula` turns it into something closer to the previous one.[6]

Notice one other change to the command line: it now starts with `m=` in order to store the fitted model in the Julia variable `m`. This lets me extract return values and pass them back to Stata:

```
jl> . st_numscalar("adjR2", adjr2(m))
jl> . exit()
```

```
. display "Adjusted R2 = " adjR2
Adjusted R2 = .26365506
```

The Julia function `st_numscalar()` is part of the `julia` package; like its Mata namesake, it reads and writes Stata scalars.

5. See juliastats.org/StatsModels.jl/stable/internals/#extending.
6. Johannes Boehm's experimental Douglass.jl package includes macros to translate Stata-like commands into Julia ones. An example: `d"bysort :Species : egen :z = sum(:SepalLength) if :SepalWidth .> 3.0"`.

2.2 Built for speed

To partially convey why Julia is a good environment for developing numerical software, I develop another example. Suppose we have a data set with 10 million rows and 10 columns named `x1-x10`. Given a 10×10 matrix $\mathbf{Q}$, we want to compute the norm of each row according to the quadratic form defined by $\mathbf{Q}$, and store all the results in a new column. That is, for each row x_i, now viewed as a column vector, we want $x_i'\mathbf{Q}x_i$. It is natural to implement this computation with triply nested `for` loops. That's a good way to do it in Julia (inside Stata):

```
jl AddPkg LoopVectorization  // one-time package installation from GitHub
jl: using LoopVectorization  // load package for use now
jl: function XQX(Q, X)                              ///
      N, M = size(X);                               ///
      retval = zeros(N,1);                          ///
      @tturbo for i in 1:N                          ///
        for j in 1:M                                ///
          for k in 1:M                              ///
            retval[i] += X[i,j] * Q[j,k] * X[i,k]  ///
          end                                       ///
        end                                         ///
      end;                                          ///
      return retval                                 ///
    end
```

Unlike in Python, indentation has no semantic meaning; this code snippet is indented only for readability.

We saw in the previous example that one can start an interactive Julia session inside Stata by typing `jl` (and end it with `exit()`). But the interactive mode is not accessible to do files. To exemplify Julia code in a do file, the above snippet is therefore written effectively as a single line. It is typographically split across multiple lines through Stata's continuation token, `///`. When putting many Julia commands on what is syntactically one line, most not immediately followed by `end` need to terminate with a semicolon.

The code for this new Julia function, `XQX()`, is mostly straightforward. It initializes the return value to a column of 0's, performs the calculation through nested loops, and returns the result. One detail is unusual: another macro is invoked. `@tturbo` is part of Chris Elrod's LoopVectorization.jl package. `@tturbo` analyzes the code that follows it and substantially—but invisibly—rewrites it for speed. It may *unroll* a loop to reduce the number of jumps back to the top. The macro, or the compiler, may spot the opportunity to move `X[i,j] *` out of the inner loop since it does not depend on the loop's index, `k`. The macro may even reorder the nesting of the loops with an eye toward the fact that in Julia, matrices are stored column by column, and on-chip memory caching makes it is faster to access adjacent memory locations in sequence. The LoopVectorization package is called that because it also tailors code so that the resulting machine language exploits the ability of modern chips to vectorize operations *within* each core, in 256- or 512-bit registers—what is called single instruction, multiple data (SIMD) execution. And `@tturbo` will exploit multithreading, the subdivision of work *across* cores; that is what the first "t" in `tturbo` stands for.

Triply nested implementations in Mata and Python are much slower, I assume because the low-level calculations in the innermost loop have to be interpreted a billion times. The best alternatives I have found in the two languages are essentially the same:

```
mata
function XQX(Q, X)
  return (rowsum((X * Q) :* X))
end

python
def XQX(Q,X):
  return ((X @ Q) * X).sum(axis=1)
end
```

Both delegate the computation to built-in matrix operations, which are fast because they are implemented in compiled C or Fortran. However, both contain a subtle inefficiency. They create the large, temporary matrices $\mathbf{XQ}$ and $\mathbf{XQ} \otimes \mathbf{X}$. Since in our example $\mathbf{X}$ is $10^6 \times 10$, and since double-precision numbers take 8 bytes, the temporary matrices require some 800 megabytes each. Allocating and deallocating memory itself takes time. Expanding the memory footprint of an algorithm can also reduce speed by exceeding the capacity of a CPU's memory caches, forcing the CPU to idle while data are written to and read from the computer's main memory. The triply-looped Julia code demands essentially no temporary storage.

To create a test bed for these functions, I construct $\mathbf{X}$ as a Stata data set and $\mathbf{Q}$ as a Stata matrix, then copy them into Mata, Python, and Julia. The `set rmsg on` command generates timings:

```
. python: import numpy as np
. python: from sfi import Data, Matrix
. set obs 10000000
Number of observations (_N) was 0, now 10,000,000.
. drawnorm double x1-x10
. mata  : st_matrix("Q", makesymmetric(runiform(10,10)))
. mata  : Q = st_matrix("Q")
. python: Q = np.asarray(Matrix.get("Q"))
. jl    : Q = st_matrix("Q");
. set rmsg on
. mata: X = st_data(.,"x1-x10")
r; t=0.07
. python: X = np.asarray(Data.get(var="x1 x2 x3 x4 x5 x6 x7 x8 x9 x10"))
r; t=6.73
. jl: X = st_data("x" .* string.(1:10));
r; t=0.24
. jl: X = st_data("x" .* string.(1:10));
r; t=0.15
```

Under a Julia convention, the semicolons at the end of the `jl` lines suppress the printing of the results of the assignments. The Julia data copying command is run twice in order to show that `st_data()`, which is also part of the `julia` package, takes extra time on first use because of just-in-time compilation.

On a Windows laptop with an Intel i9-13900H, importing the 100 million data points into a matrix takes just 0.07 seconds for Mata, 0.15 seconds for Julia (the second time), and a substantial 6.73 seconds for Python. In `jl`, the data copying is performed by C++ code that is multithreaded across columns.

Next I test the three functions I defined:

```
. mata: y = XQX(Q,X)
r; t=0.31
. python: y = XQX(Q,X)
r; t=0.48
. jl: y = XQX(Q,X);
r; t=1.39
. jl: y = XQX(Q,X);
r; t=0.05
```

The Julia function is especially slow on first use because that is when `@tturbo` performs its magic. On later calls, Julia is fast: it takes 0.05 seconds, which works out to about 40 billion floating point operations per second. Mata takes 0.31 seconds and Python, 0.48. To reduce the burden of the first call, if the Julia `XQX()` function were part of a shared package, it could be accompanied by a precompilation script much like the test above. In order to trigger immediate compilation, the Julia package manager would run the script after installing the package. Julia would save the generated machine code for future use.

3 Applications

This section presents three applications of `julia`. They involve HDFE modeling, wild bootstrap–based inference, and plotting.

3.1 Estimation with high-dimensional fixed effects

The introduction of `reghdfe` in 2014 (Correia 2016) was an important event in applied econometrics. Building on Guimarães and Portugal (2010) and Gaure (2011), the Mata-based program proved the computational tractability of linear models with many sets of fixed effects. Since then, other authors have developed alternative algorithms for absorbing fixed effects, several of which have been incorporated into `reghdfe`. The ideas have been extended to the Poisson model through `ppmlhdfe` for Stata (Correia, Guimarães, and Zylkin 2019), and to other generalized linear models, in `alpaca` and `fixest` for R (Stammann 2018; Bergé 2018). Julia packages for HDFE models are also available: Matthieu Gomez's FixedEffectModels.jl and Johannes Boehm's GLFixedEffectModels.jl. With regard to linear models, according to benchmarking results on the GitHub pages of `fixest` and FixedEffectModels.jl, those two packages are 1–2 orders of magnitude faster than `regdhfe`.[7]

These developments have given R and Julia users faster and more general implementations of HDFE modeling than were available in Stata. That is what motivated the development of the `julia` and `reghdfejl` packages. The idea was to not reinvent

7. See github.com/lrberge/fixest#benchmarking and github.com/FixedEffects/FixedEffectModels.jl#benchmarks.

wheels, but to fashion a familiar-looking chassis onto existing wheels.

The extended example in section 2.1 followed a sequence that is the essence of `reghdfejl`: copy data from Stata to Julia, estimate in Julia, return results to Stata. Here I complete the narrative arc of that example. With a small typographic change to the last Julia estimation line—wrapping a variable name in `fe()`—I exploit the ability of FixedEffectModels.jl to absorb fixed effects:

```
. jl: reg(auto, @formula(price ~ headroom + fe(foreign) & mpg), Vcov.cluster(:turn))
FixedEffectModel
============================================================================
Number of obs:                     74  Converged:                       true
dof (model):                        1  dof (residuals):                   34
R²:                             0.294  R² adjusted:                    0.264
F-statistic:                 0.991311  P-value:                        0.334
R² within:                      0.583  Iterations:                         1
============================================================================
             Estimate  Std. Error     t-stat  Pr(>|t|)  Lower 95%  Upper 95%
────────────────────────────────────────────────────────────────────────────
headroom      -273.24     274.435  -0.995646    0.3342   -855.017    308.536
(Intercept)  13743.6     2176.8     6.31369     <1e-04   9129.03   18358.2
============================================================================
```

Back in Stata, here is `reghdfejl` running the same regression:

```
. reghdfejl price headroom, absorb(foreign#c.mpg) cluster(turn)
(MWFE estimator converged in 1 iterations)

HDFE linear regression with Julia                 Number of obs   =         74
Absorbing 1 HDFE group                            F(   2,      15) =       0.99
Statistics cluster-robust                         Prob > F        =     0.3941
                                                  R-squared       =     0.2939
                                                  Adj R-squared   =     0.2637
Number of clusters (turn)    =         18         Within R-sq.    =     0.5829
                                                  Root MSE        =  2530.9785

                                  (Std. err. adjusted for 18 clusters in turn)
------------------------------------------------------------------------------
             |               Robust
       price | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
    headroom |  -273.2403   274.4352    -1.00   0.334     -855.017    308.5364
       _cons |   13743.64     2176.8     6.31   0.000     9129.026    18358.24
------------------------------------------------------------------------------
```

In spirit, `reghdfejl` is a slot-in replacement for `reghdfe`, with the same syntax, output, and `e()` return values. In practice, the two differ outwardly as well as inwardly. As `reghdfjle` depends on the feature set of FixedEffectModels.jl, it offers only one optimization technique, LSMR (Fong and Saunders 2011)—though this has never been a problem in my experience. `reghdfejl` does not support "group fixed effects." And it does not adjust the reported degrees of freedom for any collinearity of the fixed effects with each other or with the error clustering. So it often reports slightly larger standard errors. While `reghdfejl` can perform instrumental variables (IV) estimation, unlike `ivregdfe` it does not by default work as a wrapper for `ivreg2` (Baum, Schaffer, and Stillman 2007). As a result, it can only perform two-stage least squares (2SLS); and for weak identification diagnosis, it only reports the Kleibergen-Paap first-stage F statistic. `reghdfejl` *does* accept an `ivreg2` option, which directs the program to call

FixedEffectModels.jl to absorb the fixed effects and pass the results to `ivreg2`. But in this mode, `reghdfejl` is no faster than `ivreghdfe`. For on hard problems, `ivreg2` dominates the runtime.

`reghdfe` and `reghdfejl` share the following options: `absorb()`, `vce()`, `residuals()`, `tolerance()`, `iterate(#)`, `nosample`, `keepsingletons`, `compact`, `level(#)`, and standard Stata regression display options. In particular, `reghdfejl`'s `absorb()` can save some or all of the estimated fixed effects. For instance, `a(firm year_fe=year)` will absorb firm and year fixed effects, and store estimates of the latter in the variable `year_fe`. For both `reghdfe` and `reghdfejl`, the `tolerance()` option sets a precision threshold for the fitting algorithms to declare convergence; however, the semantics are not necessarily the same, so that, say, `tol(1e-6)` could elicit slightly different results from the two.

`reghdfejl` accepts two distinctive (documented) options:

`gpu` requests the use of a GPU. Currently the option saves more time on NVIDIA than Apple GPUs.

`threads(#)` lowers the limit on the number of threads that can be used. It cannot increase the thread count above the limit set when Julia is started (see section 4.2). This option is rarely used.

`verbose` directs `reghdfejl` to show more of its work—to display the Julia copy of the data set, the formula for the regression model, and the regression command. The data set and formula are left behind for the user to work with through `jl`, not erased as they otherwise would be.

In addition, `reghdfejl`'s `vce()` option supports the `bootstrap` (or `bs`) variance-covariance estimator. `vce(bs)` is standard within Stata; `regress` accepts it. As an example, for `reghdfejl`, the clause

```
vce(bs, reps(1000) seed(42) procs(4) cluster(year))
```

specifies that 1000 bootstrap replications be performed, split across four copies of Julia running in parallel, with the bootstrap data drawn groupwise from clusters defined by `year`. One could attain the same results with `bootstrap:reghdfe` or `bs:reghdfe`. But `reghdfejl` implements `vce(bs)` in a radically faster way. After launching the extra instances of Julia to split the work, it copies the primary data just once to each of them, and effects the sampling in Julia by updating a weight variable (cluster 1's weights are doubled if drawn two times, etc.).

To compare `reghdfe` and `reghdfejl` on speed, I create a data set with 10 million rows and two sets of fixed effects. I run the same regression with both commands. I explore the effect of switching from one to multiple processors in Stata/MP, and of adding `reghdfejl`'s `gpu` option:

```
set obs 10000000
gen id1 = runiformint(1, 100000)
gen id2 = runiformint(1, 100)
drawnorm x1 x2
gen double y = 3 * x1 + 2 * x2 + sin(id1) + cos(id2) + runiform()
```

```
set processors 1
reghdfe   y x1 x2, a(id1 id2) cluster(id1 id2) dof(none)
reghdfejl y x1 x2, a(id1 id2) cluster(id1 id2)
reghdfejl y x1 x2, a(id1 id2) cluster(id1 id2) gpu
set processors 6
reghdfe   y x1 x2, a(id1 id2) cluster(id1 id2) dof(none)
reghdfejl y x1 x2, a(id1 id2) cluster(id1 id2)
reghdfejl y x1 x2, a(id1 id2) cluster(id1 id2) gpu
```

For comparability, the `reghdfe` lines include `dof(none)` to disable the degrees-of-freedom correction, since `reghdfejl` lacks that feature. The first triplet runs on one CPU core, as under a non-MP flavor of Stata—except that the Julia components still run across 6 threads. On the laptop with an Intel i9-13900H CPU and NVIDIA RTX 4070 GPU, `reghdfe` takes 32 seconds while `reghdfejl` takes 3.8 seconds without the `gpu` option and 2.8 with. Moving to 6 processors in Stata and Mata (the Intel chip has 6 "performance" cores) lowers the times to 24, 3.3, and 2.7 seconds.

Another script benchmarks `reghdfejl`'s `vce(bs)` option:

```
webuse nlswork
xtset, clear
bs, cluster(occ_code) reps(1000): ///
  reghdfe ln_wage age ttl_exp tenure not_smsa south, absorb(year occ_code) dof(none)
parallel initialize 6
parallel bs, cluster(occ_code) reps(1000): ///
  reghdfe ln_wage age ttl_exp tenure not_smsa south, absorb(year occ_code) dof(none)
reghdfejl ln_wage age ttl_exp tenure not_smsa south, absorb(year occ_code) ///
  vce(bs, cluster(occ_code) reps(1000) procs(6))
```

Using Stata's `bs` prefix command with `reghdfe` takes 76 seconds. Switching to the `parallel` package of George Vega and Brian Quistorff in order to split the work among 6 copies of Stata[8] cuts the time to 23 seconds. `reghdfejl` takes just 2.5 seconds. But, as usual, the Julia function is slower the first time it is invoked in a Stata session: this one spins up 6 worker copies of Julia and triggers some just-in-time compilation in each. The workers remain active after completing their first task, so subsequent bootstrapping is fast as long as the number of workers is kept at 6.

The `reghdfejl` package includes a `partialhdfejl` command for absorbing fixed effects out of other variables. `partialhdfejl` command lines must contain either a `generate()` option to name the variables that will hold the results, or a `prefix()` option for making the new variable names out of the old. If any of the new variables already exists, `partialhdfejl` will error, unless the `replace` option is also included. The following two runs produce the same point estimates:

```
reghdfejl ln_wage age ttl_exp tenure south, absorb(year occ_code) cluster(occ_code)

partialhdfejl ln_wage age ttl_exp tenure south, absorb(year occ_code) prefix(_)
regress _ln_wage _age _ttl_exp _tenure _south, cluster(occ_code) nocons
```

To fit nonlinear HDFE models with `reghdfejl`, one includes `family()` and/or `link()` options in the command line, following the syntax of the built-in Stata command `glm`. This code below fits an HDFE Poisson model with the Mata-based `ppmlhdfe` as

8. See github.com/gvegayon/parallel.

well as with `reghdfejl`. The latter runs about twice as fast:

```
use http://fmwww.bc.edu/RePEc/bocode/e/EXAMPLE_TRADE_FTA_DATA
egen imp = group(isoimp)
egen exp = group(isoexp)
expand 100  // expand data set 100-fold for tougher test
ppmlhdfe  trade fta, a(imp#year exp#year imp#exp) cluster(imp#exp)
reghdfejl trade fta, a(imp#year exp#year imp#exp) cluster(imp#exp) family(poisson)
```

In the same way, one can fit HDFE logit, binomial, negative binomial, and other models. However, at this writing, the underlying Julia package, GLFixedEffects.jl, lacks key features such as observation weighting and the equivalent of Stata's `exposure()` option for Poisson models. So `reghdfejl`'s ability to fit nonlinear models is currently undocumented.

3.2 Wild bootstrapping

Roodman et al. (2019) introduces the Stata package `boottest` for inference via the wild bootstrap. Among the methods `boottest` implements is the "wild restricted efficient" bootstrap (WRE), which is the Davidson and MacKinnon (2010) adaptation for instrumental variables (IV) estimation. Since Roodman et al. (2019), `boottest` has been optimized in ways that especially benefit the WRE. Using the Frisch-Waugh-Lowell theorem, exogenous controls are now partialled out of the other variables using the same algebraic tricks that accelerate `boottest`'s wild bootstrap for ordinary least squares. The partialling-out accelerates the application of the user's IV estimator despite its irreducible nonlinearity (Roodman et al. 2019, section 6.1). Also incorporated (more fully), is MacKinnon (2023)'s point that the data only enter wild-bootstrapped inferences through indexed sets of matrix products such as $\{\mathbf{X}_g'\mathbf{Y}_g : g = 1, ..., G\}$ where $\mathbf{X}_g$ and $\mathbf{Y}_g$ hold the rows of data matrices $\mathbf{X}$ and $\mathbf{Y}$ for cluster g. Reducing the data to these products early in the calculation complicates the algebra the code must implement, but speeds execution when clusters are few.

Finally, the back-end functionality in `boottest` has been transplanted to the Julia package WildBootTests.jl. Adding the `julia` option to a `boottest` command line switches to the new back end.

Section 8.4 of Roodman et al. (2019) approximately replicates two 2SLS regressions in the Levitt (1996) study of the short-term impact of decarceration on crime.[9] Here is a version of that example that substitutes `reghdfejl` for `ivregress` and sets the stage for adding the `julia` option to the `boottest` command line:

```
use Levitt, clear
set seed 8723419
foreach crimevar in Violent Property {
  reghdfejl D.l`crimevar'pop (LD.lpris_totpop = ibnL.stage#i(1/3)L.substage) ///
    D.(lincomepop unemp lpolicepop metrop black a*pop) i.year, a(state) clust(state year)
  boottest LD.lpris_totpop,bootclust(year) ptype(equaltail) ///
    gridmin(-2) gridmax(2) nograph
}
```

9. Data and preparation code are available at davidroodman.com/david/Levitt.zip.

On a single CPU core, version 2.3.5 of `boottest`, the one published with Roodman et al. (2019), needs 210 seconds to perform the two `boottest` calls. The latest version lowers finishes in 7.4 seconds. Adding the `julia` option to the `boottest` line brings the time to 4.1 seconds.

The R package `fwildclusterboot` (Fischer and Roodman 2021) also calls WildBootTests.jl to perform the WRE. This may be the first instance of a Julia statistical package serving as a back end on multiple platforms.

3.3 Plotting

Many data visualization packages are available for Julia. Surveying them is beyond the scope of this article. To indicate their potential for Stata users, I generate two graphs with Makie.jl.

Makie works with several plotting back ends, which render output in formats that are optimized for different applications, such as interactive graphs in web pages and publication-quality image files. The Cairo back end is best for the latter. It is bundled with Makie in CairoMakie.jl:

```
jl AddPkg CairoMakie  // one-time installation
jl: using CairoMakie   // use now
```

In 2020, Chuck Huber of StataCorp blogged about using Python to visualize results from a Stata regression.[10] In Huber's example, a binary indicator of high blood pressure is regressed on age, weight, and their product using the `logistic` command. `margins` is then called to compute the predicted probability of high blood pressure according to the fitted model, across a grid of ages and weights. The predictions are copied to Python and depicted in a three-dimensional surface plot. A Julia adaptation of that example:

```
webuse nhanes2
svy: logistic highbp c.age##c.weight
margins, at(age=(20(5)80) weight=(40(5)180))
matrix xyz = r(at), r(b)'  // x, y, z values in 3 columns
jl
  df = DataFrame(st_matrix("xyz"), [:age, :weight, :pr_highbp])  # get data
  f = surface(df.age, df.weight, df.pr_highbp,
              axis=(type=Axis3,
                    title = "Probability of Hypertension by Age and Weight",
                    xlabel = "Age (years)",
                    ylabel = "Weight (kg)",
                    zlabel = "Probability of Hypertension"))
  f |> display  # equivalent to display(f)
  save("fig1.png", f)
exit()
```

This time I format the Julia commands not as they would be written in a do file, but as they would be typed—or pasted—into Stata's Command window, That is, I drop the ";" and "///" from the ends of lines. The first Julia line imports the data as a matrix and then converts it to a DataFrame with named columns. Then the surface plot is built and stored in the Julia variable `f`. The call to `surface()` refers to columns of the

10. blog.stata.com/2020/09/14/stata-python-integration-part-5-three-dimensional-surface-plots-of-marginal-predictions

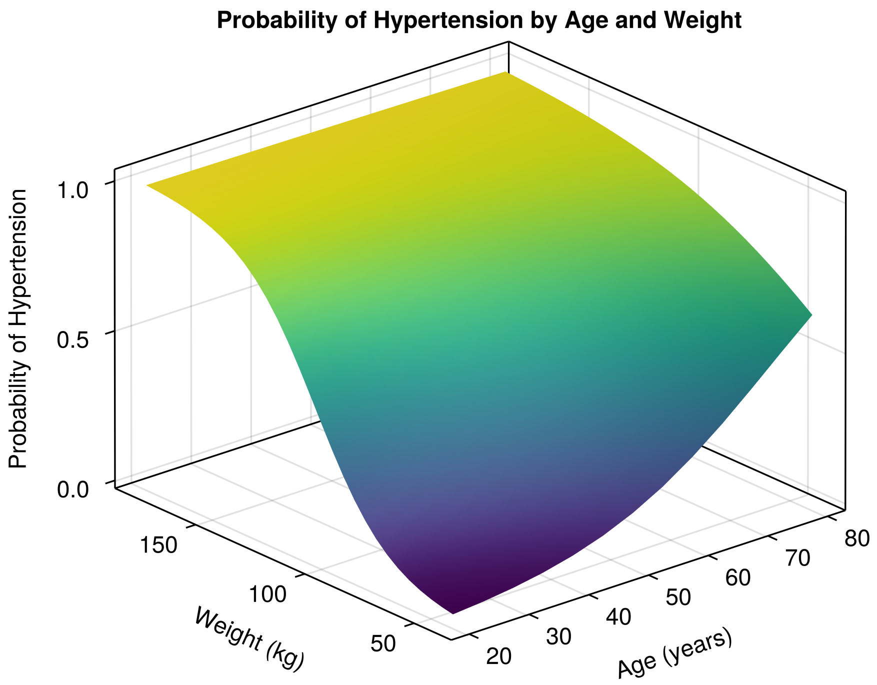

Figure 1: Contour plot made with Makie in Julia

DataFrame with the convenient dot syntax, as in `df.age`; here, column names are not prefixed with a colon. The next command displays the figure in a new window, which might pop up behind the Stata window. The line takes advantage of Julia's piping syntax for function calls. Last, the image is saved. It appears in Figure 1.

From the same data set, I plot the distribution of body mass index (BMI) by race. This time, instead of pulling the data into Julia with `st_matrix()`, I copy the entire data set from Stata with `jl save`. Since I specify no name for the target DataFrame, it defaults to `df`:

```
jl save
jl
  dropmissing!(df, :bmi)
  races = ["Other", "White", "Black"]
  f = Figure()  # make new, empty figure
  Axis(f[1, 1], yticks=((1:3)/20, races), xlabel="BMI")  # add empty plot region
  for (i,r) in enumerate(races)
    density!(df.bmi[df.race .== r], offset=i/20, color=(:slategray, .4), bandwidth=1)
  end
  f |> display
exit()
```

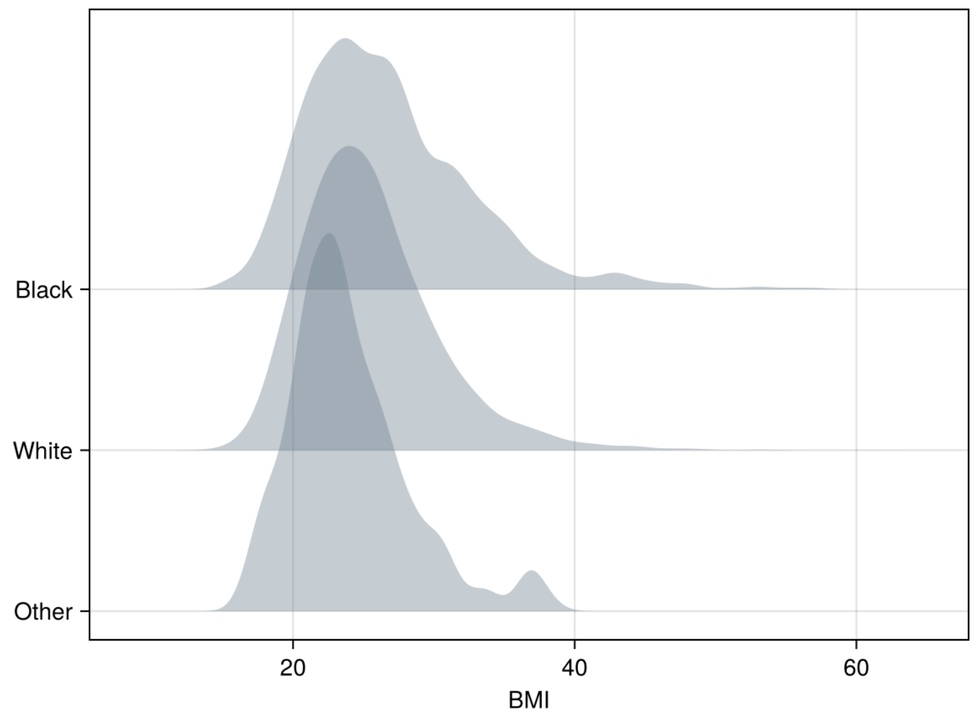

Figure 2: Overlapping density plots made with Makie in Julia

`dropmissing!(df, :bmi)` is similar to `drop if bmi==.` in Stata. However, it changes the type of `df.bmi` from `Float32?` to `Float32`, meaning that the column no longer accepts missing values. That is apparently necessary to make Makie work. The "`!`" in "`dropmissing!`" has no special meaning in the syntax of Julia: it is just a character in the function's name. By convention, functions that modify their arguments have names ending in "`!`".

The `for...enumerate()` loop iterates over the racial groups. `r` takes race names as string values while `i` is a counter starting from 1. In the Stata data set, the `race` column is coded with 1, 2, and 3. Those are displayed through a value label as "White", "Black", and "Other". Since `jl save` stores `race` as a CategoricalVector, the Julia column's contents are outwardly strings, not numbers. `df.bmi[df.race .== r]` uses Boolean indexing to extract the elements of the `bmi` column for which `race` takes the string value in `r`.

Each iteration plots a density in slate gray with 40% opacity. The spacing offset of 1/20 is interpreted in the vertical axis's units, BMI density, and was chosen by trial and error to cause overlapping. See Figure 2.

4 Tips on using the `jl` command

4.1 The JuliaMono font

Julia output routinely contains certain Unicode characters, such as vertical and diagonal ellipses, that Stata fonts do not cover. (However, this is not necessarily the case for Stata packages that call Julia back ends.) The JuliaMono font is a free, monospaced font that includes those characters, and looks reasonably good in Stata's results window. The font can be installed from github.com/cormullion/juliamono/releases. Type `help fonts` in Stata to learn how to change the Results window font.

4.2 Multithreading

Julia includes several facilities for multitasking. One is distributed processing, which involves launching multiple copies of Julia that can pass messages and data to each other. `reghdfejl`'s `vce(bs)` option uses that feature. A lighter form is multithreading, in which multiple execution lines run within one Julia process, largely sharing the same data space. That is the sort `@tturbo` exploits. The maximum number of threads Julia allows is set when it launches, and it can default to 1. A good guess for the optimum is the number of "performance" or "big" cores on the CPU.

How best to set Julia's thread limit depends on the operating system. In Windows, you can open the Environment Variables control panel and assign `JULIA_NUM_THREADS`. In Linux, you can add an entry such as `export JULIA_NUM_THREADS = 6` to "~/.bashrc" or the equivalent. In macOS, environment variables do not affect graphical applications such as Stata when launched from the desktop. One can instead set the limit in each Stata session through the `threads()` option of the `jl start` command. Example: `jl start, threads(8)`. This must be the first `jl` command issued since starting Stata. It must also precede invocation of Julia-dependent programs such as `reghdfejl`.[11]

4.3 Limitations on accessing Stata objects

The `julia` package lets you transfer information between Stata and Julia with commands issued in either language; section 5 will lay out more details. However initiated, the job of copying falls to the C++ plugin. To read and write Stata macros, scalars, matrices, and variables, the plugin must in turn call the official Stata plugin interface.[12] The SPI imposes certain constraints. For one, it makes public no functions that can create Stata variables or matrices. As a result, you can edit existing Stata variables and matrices from Julia, but you cannot create new ones. The Stata-side `jl` subcommands can and do evade this constraint because they start from the Stata environment. For instance, `jl GetMatFromMat` creates a destination Stata matrix before calling the plugin to copy from a Julia matrix.

11. In macOS, one can in fact assign JULIA_NUM_THREADS in "~/.zshenv". This will take effect when one launches Stata from the terminal. For example `open -a StataMP` will launch Stata/MP.
12. See stata.com/plugins.

A subtler limitation pertains to local macros, and explaining it requires a deeper dive. The “jl.ado” file declares the plugin “jl.plugin” with this line:

```
program _julia, plugin using(jl.plugin)
```

You are not supposed to call the plugin directly. But you could. For example:

```
plugin call _julia, eval "1+1"
```

The designated entry point to the plugin, the C++ function `stata_call()`, would then receive and processe the two arguments after the comma, which tell it to have Julia evaluate “1+1”. Since calls to the plugin are in general arcane and sometimes require preparatory steps, the `jl` program serves as an essential and more user-friendly wrapper. That extra layer complicates access to local macros. If Julia, having been invoked by `jl` and the plugin, uses the SPI function `SF_macro_save()` to write a Stata local named `foo`, then `foo` will appear in the macro name space of the program that called the plugin, which is `jl`. It will *not* appear in the context a step up, which could be your code. As a workaround, every time the Julia function `st_local()` is called, it adds the name of the macro it is writing to another local, `__jllocals`. Before exiting, `jl` copies every macro in that list to its caller’s context using the undocumented Stata command `c_local`.

However, the same complication prevents one from *reading* Stata locals from Julia. The workaround just described only goes in one direction, as I know of no way for an ado program to read its caller’s local macros. In partial compensation, single-line `jl:` commands can quote locals, because Stata processes the lines before passing them to `jl`. For example:

```
local varname x1
jl: sum(df.`varname')
```

This trick will not work in an interactive Julia session.

4.4 Speed considerations in copying data: type conversions, missingness, and subsample restrictions

The specification of the Stata plugin interface introduces several wrinkles into the copying of data between Stata and Julia.

One is that the functions for reading and writing numeric Stata variables, `SF_vdata()` and `SF_vstore()`, only work with double-precision values, regardless of whether a Stata-side variable is stored as `byte`, `int`, `long`, `float`, or `double`. By default, the `jl` commands that copy from the active Stata data set to Julia DataFrames—`jl save` and `jl PutVarsToDF`—convert the double-precision values back to the original data types.[13] So a Stata `byte` variable will be converted to `double` and back on the way to appearing in a Julia DataFrame. The `jl` commands can do nothing about the first conversion,

13. Because the ranges of Stata’s `byte`, `int`, and `long` types are narrowed in the way described in the next paragraph, when transferring data from Julia to Stata, the corresponding Julia types are mapped to larger types in Stata. For example, columns of type `Int8`, which have an allowed range of $[-128, +127]$, are copied to Stata’s two-byte `int` type, since Stata `bytes` only hold $[-128, 100]$.

but they accept a `doubleonly` option to prevent the second and instead leave all copied values in double precision. This typically saves time, both by reducing conversions and by allowing Julia to allocate a contiguous block of memory to receive all the data, rather than a separate block for each of variously typed columns.

Separately, Stata and Julia represent missing values in different ways. Stata reserves certain high values of each data type for 27 flavors of missing: `.` and `.a`, ..., `.z`. That is why the maximum value of a Stata `byte` is 100, not than 127. Julia only supports one flavor of missing, and stores missingness information alongside rather than in data values. It also allows observations of string variables to be missing. The `julia` package translates missings when copying between Stata data sets and Julia DataFrames (but not Julia matrices). For data known to contain no missing values, adding the `nomissing` option will save time by skipping this step.

Finally, the SPI lets plugin calls include *`if`* or *`in`* clauses to restrict the sample on the Stata side. The `julia` plugin contains optimized code that runs when such as clauses are not specified.

Data transfers are thus fastest when they map double precision to double precision and include no missing values or sample restrictions. Consider this example, which creates a 10 million $\times$ 10 set of double-precision numbers and copies them to a Julia DataFrame named `demo`:

```
set obs 100000000
drawnorm double x1-x10
jl PutVarsToDF x1-x10, nomissing doubleonly dest(demo)
```

On my computer, the last line takes 1.3 seconds, for a transfer rate of 6 gigabytes/second.

4.5 Simulated read-eval-print loop

The Julia language system has many components—for interpreting and compiling code, managing memory, and so on. The core parts are written in C++ and distributed as compiled libraries. Other components are themselves written in Julia, including the front end, which users often think of as Julia per se. The front end is usually called the REPL, for "Read-Eval-Print Loop." Indeed, the REPL's core job is to read in commands, evaluate them, print the results, and repeat. But the REPL provides other conveniences: typing "?" starts help mode; typing "]" instead switches to the Julia package manager; some use is made of color; arrow keys allow one to explore one's command history.

While there may be a way to run the full Julia REPL inside of Stata, the `jl` command does not do that. It gins up a limited REPL. That is because The plugin directly accesses lower-level components of Julia, through the C-language entry points that loosely comprise Julia's interface for "embedding" in C. A workhorse for `jl` is the Julia system's `jl_eval_string()` function. It accepts a C string holding a Julia expression to be evaluated and returns the result in a C structure that represents a generic Julia object. All Julia lines submitted to `jl` pass through this function. As its name suggests, the job of the function is to evaluate strings, not provide a user experience.

Especially when in interactive mode, `jl` takes steps to create an ersatz Julia REPL. As described in the first example in section 2, it checks whether lines are syntactically complete and prompt the user to continue typing if not. That allows one to enter a multiline code block without triggering a syntax error before the end. `jl` also captures printed output (from `stdout`) in order to display not only the return value of a command but anything printed while running it. `jl` translates lines beginning with "?" into help requests and suppresses output from lines ending with ";".

Another REPL feature that `jl` simulates matters for a code block such as this one:

```
jl
  S = 0
  for i in 1:10
    S += i
  end
  print(S)
exit()
```

This snippet prints the sum of the first 10 whole numbers—at least when run in the Julia REPL. Actually, for reasons having nothing to do with Stata, the snippet will crash if run non-interactively, as part of a stored program. Under Julia's "soft scoping" rule, since the variable `S` is not declared or initialized within the `for` loop before the first attempt to increment it, the first attempt to increment it will trigger an undefined variable error. Because that rule is counterintuitive, Julia's REPL engineers an exception for interactive use.[14] `jl:` imitates the exception by calling the Julia `softscope()` function on the user's input after parsing it. `softscope()` is itself part of the code base for the Julia REPL.

Still, `jl`'s REPL simulation is incomplete. Typing `?` and `]` alone have no effect. Plots do not automatically appear, which is why the examples in section 3.3 explicitly call `display()`.

And the processing required for the simulation, along with the special handling of Stata locals, adds time to the execution of each Julia line from Stata. The extra 0.02 seconds or so is not noticeable in an interactive session. But it can add up in a program that calls `jl:` many times. The `_jl:` variant of the prefix command bypasses all of this processing, at the price of a worse REPL simulation.

5 The `julia` package

5.1 Architecture

The `julia` package consists of three intertwined pieces, all of which have been mentioned. The core is a shared library written in C++ according to the specification for Stata plugins. It comes precompiled for Windows, Linux, and macOS with Intel or Apple silicon. The "julia.pkg" manifest file instructs Stata to select the copy of the library that will run on the target computer, and to name it "jl.plugin." The plugin

14. See the documentation at docs.julialang.org/en/v1/manual/variables-and-scoping/#local-scope.

performs two main jobs: efficiently copying data among matrices and data sets in Julia and Stata, and sending commands to Julia and returning the results.

The file “jl.ado” serves as a Stata front end for the plugin, providing a more intuitive interface and performing tasks better done in the ado language than in C++. At the suggestion of Jeffrey Pitblado at StataCorp, this program is called `jl` rather than `julia` because Stata might introduce an official `julia` command someday. When `jl` is first called in a Stata session, it briefly runs the Julia REPL through a shell command in order to query the location on disk of Julia’s shared code libraries. It passes the location to the plugin, which loads the libraries. This lets the plugin access Julia.

`jl` also calls Stata’s `findfile` to track down the third key piece of the package. “statapluginінterface.jl” defines a module of Stata interface functions in Julia, such as `st_local()` and `st_data()`. These work by calling entry points in the Stata plugin interface, which are exported to Julia by the plugin. (The Julia language includes a way to call C functions.) At start-up, `jl` therefore stores the location of “jl.plugin” in the statapluginінterface module.

5.2 Stata commands in the `julia` package

As we have seen, `jl:` and the faster `_jl:` prefix commands run lines of Julia code:

```
jl: juliaexpr
_jl: juliaexpr
```

where *juliaexpr* is an expression to be evaluated in Julia.

`jl` also accepts subcommands. As discussed in section 4.2, one is especially useful in macOS, because it allows users to set the upper limit on the number of threads:

```
jl start [, threads(# | auto)]
```

Specifying `threads(auto)` instructs Julia to use its own heuristic to pick the number of threads. Executing `jl start` is optional, for `jl` will initialize itself as needed.

Several subcommands help manage Julia packages. Just as Stata downloads packages into folders such as “/home/droodman/ado/plus,” Julia downloads and stores packages in directories called *environments*. Julia also makes it easy to change the current environment. This is useful for the developer of a Julia-calling Stata program because it allows the Stata program to install needed Julia packages without interfering with the user’s package configuration. The package management subcommands are:

```
jl GetEnv
jl SetEnv [name]
jl AddPkg name, [minver(X.Y.Z)]
```

`GetEnv` displays the directory of the current package environment and returns it in `r()` macros. On my computer, which is running Julia 1.10, the command displays:

```
. jl GetEnv
Current environment: v1.10, at C:\Users\drood\.julia\environments\v1.10
```

`jl SetEnv` moves to a chosen, named environment, which is merely a subdirectory of the default environment directory:

```
. jl SetEnv MyEnv
Current environment: MyEnv, at C:\Users\drood\.julia\environments\v1.10\MyEnv
```

If the environment does not exist, it is created and populated with the DataFrames.jl and CategoricalArrays.jl packages. One can return to the default environment with `SetEnv .` or just `SetEnv`.[15]

Last in the package management family is the `AddPkg` option, which installs a package in the current environment or, if `minver()` is specified, updates it to the latest version if the current version falls below the minimum. Versions *X.Y.Z* are in the three-part semantic versioning format.

The remaining `jl` subcommands copy data between Stata and Julia. The first six listed below are meant more as programmer's commands while the last two are higher-level. `jl GetVarsFromDF`, for example, requires that the Stata data set already be at least as tall as the DataFrame from which it will receive variables. In contrast, `jl use` works like Stata's `use` command: it replaces the current data set, and will fail unless the `replace` option is included, or the current data set has not changed since it was last saved.

```
jl PutVarsToDF [varlist] [if] [in], [destination(string) cols(string) nolabel
                                     nomissing doubleonly]
jl GetVarsFromDF varlist [if] [in], [cols(string) source(string) replace nomissing]
jl PutVarsToMat [varlist] [if] [in], destination(string)
jl GetVarsFromMat varlist [if] [in], source(string)
jl PutMatToMat matname, [destination(string)]
jl GetMatFromMat matname, [source(string)]
jl save [dataframename], [nolabel nomissing doubleonly]
jl use dataframename, [clear]
jl use varlist using dataframename, [clear]
```

The *varlist*s and *matname*s before the commas always refer to Stata variables or matrices. If a *varlist* is omitted where it is optional, it defaults to `*`, meaning all variables in the current data set in their current order. A `destination()` or `source()` option after the comma refers to a Julia matrix or DataFrame. When an optional DataFrame name is not provided, it defaults to `df`. The `cols()` option specifies the DataFrame columns to be copied to or from. It defaults to the Stata *varlist* before the comma. Destination Stata matrices and Julia matrices and DataFrames are entirely replaced. The low-level commands will create any destination Stata variables or, if `replace` is specified, overwrite them subject to any `[if]` or `[in]` restriction.

5.3 Julia functions in the `julia` package

Functions in the statapluginInterface module can also be characterized as low- or high-level. The low-level ones wrap and mimic functions in the Stata plugin interface. For example, `SF_vdata()` extracts a single observation on a single variable. The higher-level ones build on the low-level ones in order to imitate some of Mata's Stata interface

15. In distributed computing, Julia worker processes do not automatically inherit the master's package environment. One can pass on the master's environment through command line switches when launching the workers, such as in `addprocs(4, exeflags="--project=$(Base.active_project())")`.

functions. `st_view()`, for instance, returns a Julia view onto one or more numeric Stata variables, optionally restricting to a subsample. Through the view, Julia code can read and write Stata variables while treating them as part of a matrix.

The full list of functions:

`SF_nobs()`	Number of observations in Stata data set
`SF_nvar()`	Number of variables in Stata data set
`SF_var_is_string(i)`	Whether variable `i` is string
`SF_var_is_strl(i)`	Whether variable `i` is a strL
`SF_var_is_binary(i, j)`	Whether observation `i` of variable `j` is a binary strL
`SF_sdatalen(j, i)`	String length of variable `i`, observation `j`
`SF_is_missing()`	Whether a `Float64` value is Stata missing
`SV_missval()`	Stata floating-point value for missing
`SF_vstore(j, i, val)`	Set observation `j` of variable `i` to `val` (numeric)
`SF_sstore(j, i, s)`	Set observation `j` of variable `i` to `s` (string)
`SF_vdata(j, i)`	Get observation `j` of variable `i` (numeric)
`SF_sdata(j, i)`	Get observation `j` of variable `i` (string)
`SF_macro_save(mac, tosave)`	Set macro `mac`
`SF_macro_use(mac, maxlen)`	First maxlen characters of macro `mac`
`SF_scal_save(scal, val)`	Set value of scalar `scal`
`SF_scal_use(scal)`	Get scalar `scal`
`SF_row(mat)`	Number of rows of matrix `mat`
`SF_col(mat)`	Number of columns of matrix `mat`
`SF_mat_store(mat, i, j, val)`	`mat[i,j] = val`
`SF_mat_el(mat, i, j)`	Get `mat[i,j]`
`SF_display(s)`	Print to Stata results window
`SF_error(s)`	Print error to Stata results window
`st_nobs()`	Same as `SF_nobs()`
`st_nvar()`	Same as `SF_nvar()`
`st_varindex(s)`	Index of variable named `s` in data set
`st_global(mac)`	Get global macro `mac`
`st_global(mac, tosave)`	Set global macro `mac`
`st_local(mac, tosave)`	Set local macro `mac`
`st_numscalar(scal)`	Get scalar `scal`; same as `SF_scal_use()`
`st_numscalar(scal, val)`	Set scalar scal; same as `SF_scal_save()`
`st_matrix(matname)`	Get numeric Stata matrix
`st_matrix(matname, jlmat)`	Put Julia matrix in pre-existing Stata matrix
`st_data(varnames)`	Get Stata variables, as matrix
`st_data(varnames, sample)`	Get Stata variables, with sample restriction, as matrix
`st_view(varnames)`	Get Stata variables, as view
`st_view(varnames, sample)`	Get Stata variables, with sample restriction, as view

The limitations noted in section 4.3 can be seen in the list. While `st_global()` can read and write globals, `st_local()` can only write locals. `st_matrix()` can only write to Stata matrices that already exist.

An interactive session demonstrates the use of some of the functions:

```
sysuse auto
jl
  sample = rand(Bool, st_nobs())    # random Boolean vector defining subsample
  v = st_view("price mpg", sample)  # view onto subsample of two vars
  v ./= 2                           # halve these Stata data points
  st_numscalar("s", sum(v))         # sum the halved data into Stata scalar s
  st_local("m", string(st_numscalar("s"))) # put sum as string in Stata local m
exit()
```

One can obtain somewhat more information about each function through the help facility, for example with "`jl: ?st_view`".

6 Conclusions

Julia is a strong option for universal, back-end development of statistical applications. Working in Julia comes with some drawbacks, including the challenge of learning a complex new language, the delays upon installation and first use of programs, and the immaturity of some packages and their documentation. But Julia and nearly all packages written for it are free, open, and compatible with popular operating systems. The language blends features for maximizing performance where it matters with a Python-like expressiveness that makes code easier to write and maintain. The package ecosystem covers domains essential to efficient statistical work, such as multiprocessing, linear algebra, and optimization.

Instead of implementing new methods separately in Stata or R or Python, developers could write the back ends once, in Julia. Here, WildBootTests.jl shows the way: it is an optional back end for `boottest` in Stata and a required back end for instrumental variables inference with `fwildclusterboot` in R.

For Stata users and developers, the `julia` package demonstrates the practicality and value of bridging from Stata to Julia. Through a plugin, data can be piped at high speed. Most features needed to make Stata objects accessible in Julia can and have been implemented. The linkage gives Stata users access to Julia packages for estimation and plotting that offer greater speed, new features, or more flexible syntax (as for plotting). `reghdfejl` is a perhaps the "killer app" for the Julia-Stata link. It fits linear models much faster than the pioneering `reghdfe`, at least for the problems that are time consuming for `reghdfe`. And `reghdfejl` preliminarily handles nonlinear HDFE models, which brings new functionality to Stata users.

StataCorp could strengthen Stata's bond to Julia, as it has done for Python and Java. Stata's developers could, for example, extend the plugin interface to allow creation of Stata matrices and variables. They could add a Julia mode to the do file processor, so that Julia code could be embedded without awkward semicolons and line continuation symbols. They might be able to grant more access to Stata variable storage *en bloc*, so that Julia code could read and write Stata data without copying it first. And Stata could also improve on `jl`'s simulated Julia REPL.

The goal for the `julia` package was to prove a concept. In that I think it has succeeded.

7 References

Baum, C. F., M. E. Schaffer, and S. Stillman. 2007. Enhanced routines for instrumental variables/GMM estimation and testing. *Stata Journal* 7: 465–506.

Bergé, L. 2018. Efficient estimation of maximum likelihood models with multiple fixed-effects: the R package FENmlm. DEM Discussion Paper Series 18-13, Department of Economics at the University of Luxembourg. https://ideas.repec.org/p/luc/wpaper/18-13.html.

Bouchet-Valat, M., and B. Kamiński. 2023. DataFrames.jl: Flexible and Fast Tabular Data in Julia. *Journal of Statistical Software* 107(4): 1–32. https://www.jstatsoft.org/index.php/jss/article/view/v107i04.

Correia, S. 2016. Linear models with high-dimensional fixed effects: An efficient and feasible estimator. Working paper, Duke University.

Correia, S., P. Guimarães, and T. Zylkin. 2019. ppmlhdfe: Fast Poisson estimation with high-dimensional fixed effects, arXiv.org .

Davidson, R., and J. G. MacKinnon. 2010. Wild bootstrap tests for IV regression. *Journal of Business & Economic Statistics* 28: 128–144.

Fiedler, J. 2012. Imagining a Stata / Python combination. SAN12 Stata Conference 6, Stata Users Group. https://ideas.repec.org/p/boc/scon12/6.html.

———. 2013. Re-imagining a Stata/Python combination. 2013 Stata Conference 3, Stata Users Group. https://ideas.repec.org/p/boc/norl13/3.html.

Fischer, A., and D. Roodman. 2021. fwildclusterboot: Fast Wild Cluster Bootstrap Inference for Linear Regression Models (Version 0.14.0). https://cran.r-project.org/package=fwildclusterboot.

Fong, D. C.-L., and M. Saunders. 2011. LSMR: An Iterative Algorithm for Sparse Least-Squares Problems. *SIAM J. Sci. Comput.* 33(5): 2950–2971.

Gaure, S. 2011. OLS with Multiple High Dimensional Category Dummies. Memorandum 14/2010, Oslo University, Department of Economics. https://ideas.repec.org/p/hhs/osloec/2010_014.html.

Guimarães, P., and P. Portugal. 2010. A simple feasible procedure to fit models with high-dimensional fixed effects. *Stata Journal* 10(4): 628–649(22). https://www.stata-journal.com/article.html?article=st0212.

Haghish, E. F. 2019. Seamless interactive language interfacing between R and Stata. *Stata J.* 19(1): 61–82.

Levitt, S. D. 1996. The effect of prison population size on crime rates: Evidence from prison overcrowding litigation. *Quarterly Journal of Economics* 111: 319–351.

Li, C. 2019. JuliaCall: an R package for seamless integration between R and Julia. *The Journal of Open Source Software* 4(35): 1284.

MacKinnon, J. G. 2023. Fast cluster bootstrap methods for linear regression models. *Econometrics and Statistics* 26: 52–71. https://www.sciencedirect.com/science/article/pii/S2452306221001404.

Roodman, D., J. G. MacKinnon, M. Ø. Nielsen, and M. D. Webb. 2019. Fast and wild: Bootstrap inference in Stata using boottest. *Stata Journal* 19: 4–60.

Stammann, A. 2018. Fast and Feasible Estimation of Generalized Linear Models with High-Dimensional k-way Fixed Effects.

About the author

David Roodman is a senior advisor at Open Philanthropy. He wrote `xtabond2`, `cmp`, `boottest`, and other Stata programs.