Efficient Programming Workflow in Julia

This document explains how to set up your Julia environment for the fastest and smoothest workflow. This is not a tutorial on the Julia programming language. If that's what you want look here instead.

Julia is a programming language designed to solve the "two language problem". This refers to the practice of using a quick interactive programming language, such as Python, to rapidly prototype a soloution and then rewriting the prototype in a compiled language, such as C++ or Rust, for good runtime performance.

Julia promises the best of both worlds: quick interactive development and good runtime performance in a single language. If you set up your Julia environment properly then, as of late 2022, Julia largely delivers on this promise. Unfortunately, the default installation of Julia will not give you this experience.

Fortunately, with the proper system settings you can dramatically improve your programming workflow. This document will show you how to set up your programming environment to maximise interactivity and make your workflow smooth and quick.

Beginning Julia programmers have difficulty with a small set of problems that are more or less unique to Julia. The biggest, most annoying, problem is startup time of your project in the REPL, the interactive window which parses your text input and executes it as Julia code. If your project use large packages, especially those related to plotting, it can take a long time to load these packages.

The fundamental cause is that Julia is a compiled language, with compilation delayed until what would normally be considered run time in most compiled languages. The current Julia compilation system does not cache all the information generated during compilation. Consequently, code can be unnecessarily recompiled every time you start a new REPL session, even when the source code hasn't changed.

For example, a package I worked on, OpticSim.jl, takes 380 seconds to load on my computer. The OpticSim code itself takes only 600ms to load; the remaining 379.4 seconds is spent loading and compiling packages that OpticSim uses. After using the environment settings described later in this document OpticSim load time was reduced to .

The Julia community is actively working on this problem but for now it is a limitation that has to be worked around.

Another common beginner problem is how to organize your code so that Julia tools such as the VSCode IDE, the package manager, and the Revise package, work well together. These tools make assumptions about your code organization that are not well documented, or at least not well documented in one place.

If your code is organized according to these assumptions the tools mostly interact seamlessly. If not then the Julia programming experience can be confusing and frustrating.

The remainder of this document contains instructions for setting up your Julia environment for maximum efficiency and ease of use. You should perform these steps in the order they appear in this document. Some functionality will not work if you change this order.

Get an account on Julia Discourse

Don't skip this step in your eagerness to begin programming. Sooner or later, most likely sooner, you will hit a problem this guide won't help you fix. When this happens Julia Discourse is the place to go for answers.

Whether you are a beginner or an expert you should get an account on the Julia discourse forum. The Julia community is friendly and helpful and you are likely to get an answer to your question within 24 hours. Read this before posting.

Install Julia

You can install Julia manually but don't; instead install juliaup and then use `juliaup` to install Julia for you. Follow the instructions at the link to install juliaup.

After you have installed `juliaup` open a command window and type:

```
juliaup update
```

This should install the latest version of Julia. In the future you will use the same command to upgrade to the latest version of Julia. This is all you need from `juliaup` for now but if you are curious what other options are available type `juliaup` at the command line and then hit enter.

Once you have installed Julia open a command window and type `julia` at the prompt. This should start an interactive Julia REPL session and display something like this:

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.8.2 (2022-09-29)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/

julia>
```

If the Julia REPL doesn't start or your prompt doesn't look anything like this go here for troubleshooting suggestions.

Create a basic startup.jl file

Now that Julia is installed you should create a `startup.jl` file. The instructions in the `setup.jl` file will be executed at the beginning of every interactive Julia session and ensure that important packages are automatically loaded in your interactive REPL environment.

If you are a less experienced programmer the file you will create in this section is all you will need. More advanced programmers will want to also look at the Advanced setup.jl file section.

On linux the startup file is located at `~\.julia\config\startup.jl` . Create the `config` directory if it doesn't exist. Create a file named `startup.jl` in this directory and copy this text to it.

```
using Pkg

let
    pkgs = ["OhMyREPL", "Revise"]
    for pkg in pkgs
        if Base.find_package(pkg) === nothing
            Pkg.add(pkg)
        end
    end
end

using OhMyREPL

try
    using Revise
catch e
    @warn "Error initializing Revise" exception=(e, catch_backtrace())
end
```

This will automatically load the packages Revise and OhMyREPL every time you start a Julia REPL. Revise is a package that tracks changes to Julia source files in the background. When Revise detects a file change it automatically updates the dynamic state of any interactive REPL sessions you have running. Programming Julia without it is too painful to contemplate.

The OhMyREPL package enhances the REPL by adding support for context sensitive color text and color parenthesis matching. This makes it is easier to write and edit code at the REPL.

If you are a beginner you won't need more packages in your `setup.jl` file until you have much more experience with the language. You can skip the next section and go to the Install VSCode section.

Advanced setup.jl file

This `startup.jl` file adds packages to help you manage github repos and to benchmark and debug code:

```
using Pkg

let
    pkgs = ["BenchmarkTools", "OhMyREPL", "Revise", "PkgTemplates", "Infiltrator"]
    for pkg in pkgs
        if Base.find_package(pkg) === nothing
            Pkg.add(pkg)
        end
    end
end

using BenchmarkTools
using OhMyREPL
using PkgTemplates
using Infiltrator

try
    using Revise
catch e
    @warn "Error initializing Revise" exception=(e, catch_backtrace())
end

ENV["JULIA_EDITOR"] = "code.cmd"
```

- PkgTemplates will be covered in the section Create a new project. It adds functionality that is especially useful for github repos.
- Infiltrator is a low overhead way of inserting breakpoints in your code. It will be covered in the section Debugging. Julia has an interactive debugger but it can be painfully slow if your code does significant computation. Infiltrator is fast but not as full featured as the debugger.

- BenchmarkTools adds tools for benchmarking code. These tools will help you find and fix inefficient code.

Other startup packages to consider

InteractiveCodeSearch is great for finding the source locations of functions from the REPL. It requires the installation of peco, which is easy on Linux but more complicated on Windows.
AbbreviatedStackTraces reduces the size of Julia stack traces, which can be incredibly long. Most of the time you only need to see a small part of the trace directly connected to your code. AbbreviatedStackTraces shrinks stack traces to reduce irrelevant text.

Don't add too many packages to your `startup.jl` file. Each package takes time to load and this overhead is incurred every time you start the REPL.

Install VSCode

This document assumes you will be using VSCode as your Julia IDE. It has the best support and features of the available IDE's. With VSCode installed you will rarely need to leave the IDE since the Julia language extension adds support for an interactive REPL panel and for plotting.

Install VSCode from here https://code.visualstudio.com/Download. Start VSCode and click on the extension manager. Type `julia` in the extension manager search window and install the Julia language extension.

The remaining setup instructions in this section are executed from within VSCode. You will need a running instance of VSCode to make these changes.

Set `Auto-Save` to `onFocusChange`

Type `ctrl-shift-p` to open the command palette. Type `settings` in the command palette search window and select `Preferences:Open Settings (UI)`. Type `Files:Auto Save` in the settings search bar. On the drop down menu select `onFocusChange`. This will configure VSCode to automatically save your files when the cursor moves from a text editor window to a Julia command prompt window.

This setting will make the view of your source in the VSCode text editor and the dynamic state of the interactive REPL session more consistent. One of the disadvantages of interactive programming systems is that the source text you are editing in VSCode and the definitions of functions, constants, etc., in the interactive session window can get out of sync. You change a function definition in the text editor and save the file but forget to evaluate the new definition in the interactive window so it still has the old definition. This can be very confusing.

The combination of Revise and auto save on focus change can almost completely eliminate this problem. Whenever the cursor moves out of the text editor window the file will be saved. Revise will detect this change and update the interactive environment in the REPL. What you see in the text editor will be a close match to what is defined in the REPL.

There are a some changes Revise will not update for you:

- changes to a `struct` definition
- changes to a `const`

Revise will warn you when you make changes like this. If you want them updated in the REPL you must kill your current REPL and start a new one.

Set `Julia:Num Threads`

By default VSCode starts Julia with 1 thread enabled. To automatically enable the full number of threads available on your machine open the command palette (ctrl-shift-p), type `settings` in the command palette search bar and select `Preferences:Open Settings (UI)`. Type `julia:num` in the settings search bar. Select `Edit in settings.json` in the `Julia:Num Threads` option. Type this line into the settings.json file:

```
"julia.NumThreads": "auto"
```

Set `Julia:use an existing custom sysimage when starting the REPL`

Some packages, such as Plots, take a long time to load. If your project uses several such packages the startup time for a new REPL session can easily be tens of seconds. You can precompile these packages into what is called a sysimage which loads more quickly. A custom sysimage can reduce startup time from tens of seconds to less than a second. If startup time is an issue, which it almost always is, then you should use a precompiled sysimages. This can be down with a script or, more conveniently, VSCode has built in tools to generate the custom sysimage with a button click.

Open the command palette and type `Preferences:Open Settings (UI)`. Type `julia:use custom sysimage` in the settings search bar. Select the box `use an existing custom sysimage when starting the REPL`. This will make VSCode use a custom sysimage if one is available.

There is one more step to complete the custom sysimage setup, starting a `Task` that computes the sysimage. But this command palette option only is displayed for Julia projects and you don't have one of those yet. Once you've made your first project you can finish this step and compile the sysimage.

Create your first project

Projects are not required for writing and executing Julia code but I strongly recommend that you use projects for all your Julia code. Many of the Julia tools assume that you have your code organized as projects and won't work well, or sometimes at all, if you don't have a project. Your workflow will be smoother and less error prone if you use projects.

You don't need to understand what projects are or how they work until you become more experienced. For now you only need to set them up.

beginner use generate, advanced use PkgTemplates.

Finish setting up a custom sysimage

Now let's finish setting up a custom sysimage. Recall that this step couldn't be completed until you had created your first project.

To create a custom sysimage open the command palette and type `Tasks:Run Build Task`. Select the option (currently the only one) `Julia: Build sysimage for current environment (experimental)`.

This will compile the files necessary to start your project into a single large file which will be loaded when you type using "YourProjectName" at the REPL. If your project has many dependencies this can reduce the load time for your project from minutes to seconds.

At times the sysimage compilation process will error. This usually happens because you have not updated your packages in a while. To update your packages type `]` at the `julia` command prompt to enter the package manager and type `update`. Then try running the sysimage task again.

If you add a new package in the package editor your sysimage will not be automatically updated. Every time you add a package you should rerun `Tasks:Run Build Task`, `Julia: build custom sysimage for current (experimental)`.