Notes on the Julia Programming Language

Lutz Hendricks
UNC Chapel Hill
February 12, 2020

Abstract

This document summarizes my experience with the Julia language. Its main purpose is to document tips and tricks that are not covered in the official documentation.

1 My Setup (1.2)\(^1\)

My current setup is Julia 1.3 run from the terminal and Visual Studio Code as editor (augmented with BBEdit to overcome VsCode’s shortcomings in multi-file search and replace).

Running the Julia apps seems identical to running Julia from the terminal. My startup file loads the packages OhMyREPL and Revise. Revise comes after packages from the standard libraries, so it does not track changes to those.

It appears that the default editor is determined by the system wide file association. No need to set the JULIA_EDITOR environment variable.

One drawback: Links in the terminal REPL are not clickable (update: they are when using VsCode). A substantial drawback during debugging. So I end up using BBEdit as my main editor, but do some debugging in Juno. Not ideal.

\(^1\) Each section is labeled with the Julia version for which it was last updated.
1.1 Updating to a new version (1.2)

After starting the new version, basic packages need to be added so the startup code can be run (e.g., OhMyREPL).
The bash profile needs to be updated to point to the new version.
The Jill bash script automates this process.

2 Arrays (1.1)

2.1 Indexing

Extracting specific elements with indices given by vectors:

\[ A = \text{rand}(4,3,5); \]
\[ A[\text{CartesianIndex}([1,2], [2,2]), 1] \rightarrow A[1,2,1] \text{ and } A[2,2,1] \]

Similar to using sub2ind:

\[ \text{idxV} = \text{sub2ind}(\text{size}(A), [1,2], [2,2], [1,1]) \]
\[ A[\text{idxV}] \]

To extract a “row” of a multidimensional matrix without hard-coding the dimensions, generate a view using selectdim.
To drop dimensions, use dropdims.

3 Data Handling (1.1)

DataFrames has the rough equivalent of Matlab tables.
JuliaDB resembles the data handling capabilities of traditional stats packages (like STATA) where processing happens outside of RAM.

3.1 Dataframes (1.1)

Tutorials are collected here.
Stats packages often provide data in “long” format (i.e., each id occupies many rows to cover many variables). unstack and stack can go back and forth between “long” and “short” formats.
3.2 STATA files (1.2)

Can be read using StatFiles.jl.

4 Dates and Times (1.2)

Keeping track of elapsed time (e.g., since starting a computation):

- store the start time as a DateTime: `startTime = Dates.now()`
- `round(Dates.now() - startTime, Dates.Minute)` gives the elapsed time in minutes.

5 Debugging (1.3)

ArgCheck.jl is useful for sprinkling more expressive `@assert`s around the code. As far as I can tell, the difference compared with `@assert` is that ArgCheck produces more informative errors. It shows the values of the expressions to be compared.

Infiltrate.jl can set a breakpoint similar to Matlab’s `keyboard` statement. Very useful. But one cannot continue the run after a breakpoint.

The Juno debugger stopped working in V.1.1 (invoking it hangs Julia). But the command line debugger may well be the better option.

5.1 Debugger (1.2)

My favorite debugging option.

After using Debugger invoke `@enter foo(x)` to start a debugging session. Particularly useful:

- `break_on(:error)`
- `bp add func:line` with possible restrictions on particular argument types.

Breakpoints, including the `@bp` macro, do not work in my code (throws unparsable errors) but they work in simple examples.
5.2 Rebugger

The MacOs keybinding for *interpret* is *Fn-Esc-i*.

6 Documentation (1.2)

*DocStringExtensions.jl* makes it easier to write docs. In particular, function signatures are automatically created in docstrings. *Documenter.jl* is the package to write documentation.

7 External Programs (1.3)

One can execute *bash* commands with *run*.

Question: Trying to run a bash script using *run(‘. myscript.sh’) produces a permission denied error (even though permissions are set so that others can execute. Why?

Commands are constructed like strings with interpolation: *‘mkdir $my-Path’*. 

If the command contains quotes, first build a string and then interpolate it into the command:

```
fPath = "'abc .txt'";
'ls $fPath'
```

8 Formatted Output

8.1 Formatting individual numbers (1.3)

The *Formatting* package seems to be the best bet. It uses *Python* like syntax and can format multiple arguments simultaneously (not well documented).

Example:

```
fs = FormatExpr("\{1:.2f\} and \{2:.3f\}")
format(fs, 1.123, 2)
```
yields "1.12 and 2.000". This cannot be used to format a vector of numbers in one command. Broadcasting also does not work. The easiest approach for this:

```plaintext
println(round.(x, digits = 3))
```

### 8.2 Formatting tables (1.3)

Latex output can be produced with `LatexTables.jl`.

### 9 Functions and Methods

#### 9.1 Array inputs (1.3)

It is best to restrict inputs to `AbstractArray{T1}` rather than `Array{T1}`. This way, array transformations, such as `reshape`, and `ranges` are accepted. For example:

- `typeof(1:4) <: AbstractVector{T1} where T1 <: Integer`

#### 9.2 Keyword arguments (1.3)

Passing keyword arguments through to another function is easy:

```plaintext
function foo(x; kwargs...)  
    # This is how the args are accessed inside the function
    println(kwargs[:a]);
    # All passed through and expanded into individual args. Note the
    bar(x; kwargs...);
end

bar(x; a=1, b=2) = println((a,b));

# Now we can call
foo(1; b=5) == bar(x; a=1, b=5)
```
Allowing a function to ignore “excessive” keyword arguments is also easy:

```julia
function bar(x; a=1, kwargs...)
    println((x, a));
    println(kwargs);
end

# Can be called with any arguments as long as 'x' is provided:
bar(1, b=3)
```

This is useful for functions that pass similar sets of keyword arguments to several sub-functions.

The `CommonLH.KwArgs` type and associated methods are useful for defining default values for keyword arguments.

## 10 Installation (MacOS)

Install the `Julia_ver.app` as usual.
Change `bash_profile` to point to the new version’s path.
Open Julia from Finder to override MacOS’s refusal to start an unknown app.
Exit Julia.
To keep previous packages:

```
cp -r ~/.julia/environments/v1.2 ~/.julia/environments/v1.3
```

That also copied registries.
Then it’s not a bad idea to run `]pkg up` to get latest versions of packages that are used in `Main`.

## 11 Miscellaneous

Introductions and guides:

- From Zero to Julia

Useful collections of tips, tricks, and style suggestions:
• How my Julia coding style has changed. Note in particular:
  – using named tuples and `@unpack` for functions that return multiple arguments
  – pointer to `DocStringExtensions.jl`

• Traits

12 Modules

12.1 LOAD_PATH (1.1)

Only modules located somewhere along the LOAD_PATH can be loaded with `using`. But: If a directory contains `Project.toml`, it becomes a project directory and only entries listed in `Project.toml` can be loaded (even if the directory is on the LOAD_PATH).

As a general rule, though: If one has to fiddle with the LOAD_PATH, something is probably not right. Packages have all their dependencies in `Project.toml`. Anything that gets run from the REPL is included. That only leaves potential startup code that sits in a `module` as a candidate for being on the LOAD_PATH.

12.2 Sub-Modules (1.1)

Functions from sub-modules can be exported by the main module. Example:

```julia
module scratch
export foo

module inner1
    export foo
    function foo()
        println("foo")
    end
end

using .inner1
end
```
12.3 Extending a function in another module (1.1)

The problem:
- Module B defines type Tb and function \texttt{foo(x :: Tb)}.
- Module A contains a generic function \texttt{bar(x)} that calls \texttt{foo()}. It should use the \texttt{foo()} that matches the type of \texttt{x}. That is, when called as \texttt{foo(x :: Tb)}, we want to call \texttt{B.foo}.

Solution:
- Module A:
  - Define the stub: \texttt{function foo end}
  - Call \texttt{foo(x)} from within \texttt{bar}.
- Module B:
  - Define \texttt{function foo(x :: Tb)}
  - import \texttt{A.foo}

- Now \texttt{A.bar(x)} knows about \texttt{B.foo()} and calls it when the type matches the signature.

See Duck typing when ‘quack’ is not in ‘Base’.

13 Operators

13.1 Logical (1.1)

\&\& is the logical AND operator, but in broadcasting use .\& (even though \& is a bitwise AND).

14 Optimization (Mathematical)

JuMP is a popular interface, but it requires (as of v.0.2) analytical derivatives for all objective functions.

Collections:
- NLopt
14.1 NLopt (1.2)

Objective function requires gradient as input, even if it is not used. If gradient is not provided, NLopt returns \texttt{FORCED\_STOP} without error message. When objective function errors, return value is \texttt{STOPVAL\_REACHED} and \texttt{fVal}=0.0.

14.2 Noisy objectives

Useful discourse threads: here

**SPSA:**

- according to the author: specifically made for simulation type problems
- basic idea seems to approximate derivatives, but instead of perturbing each parameter one-by-one (expensive), all are perturbed in the same step.
- extremely easy to implement
- can vary the distribution of step sizes (main algorithm uses step sizes 1 or 2 times a \(c(k)\)).

**COBYLA**

- implemented in NLopt COBYLA
- uses a linear approximation of the function

**Subplex**

- implemented in NLopt Sbplx
- similar to Nelder-Mead, but claims to be more robust

**Bayesian optimization**
14.3 Global algorithms

QuadDIRECT

- combines ideas of DIRECT with local search
- points from local search are used to form boxes for the global search

NODAL

- global optimization algorithms that can run in parallel
- possibly abandoned

Controlled Random Search

- implemented as NLopt CRS
- starts from a random population of points
- the evolves these using heuristic rules.

MLSL

- implemented as NLopt MLSL
- basic idea: multistart a local solver, avoiding resolving points that are close to each other

BlackBoxOptim

- implements SPSA
- currently no documentation of algorithms (2019/09).
15 Packages

15.1 Environments (1.2)

An environment is anything with a `Project.toml`. When you start Julia, you enter the version’s environment (e.g. 1.1). When you add a package, you effectively edit `Project.toml`.

You can add additional environments using `Pkg.activate()` or `pkg> activate`. and then `Pkg.add` to initialize a `Manifest.toml` in that directory.

The environment determines how code is loaded.

- When you type `using M` Julia looks for module `M` in all directories that are listed in `LOAD_PATH`.
- Julia also looks in the directory of the currently activate package (which is not added to the `LOAD_PATH`). Exactly what `Pkg.activate()` does internally is not clear. Once you activate another package, previously activated packages are no longer considered during code loading.
- Note: Julia does not look in the current directory (unlike Matlab). In fact, the current directory really does nothing at all, except it is the base directory for REPL commands such as `cd()` or `include()`.

When examining a particular directory in `LOAD_PATH`, what happens depends on whether the directory contains `Manifest.toml` (or `Project.toml`; two go together).

- If it does not, Julia looks for `M.jl` in this directory.
- Otherwise, Julia only looks in `Manifest.toml`. The only part is key. Julia does not look in the directory itself.

15.1.1 Stacked environments

When you activate an environment, you do not deactivate previous environments. Instead, you now operate in a sort of union of all the environments that you activated during a session. This matters when both environments list the same packages in the Manifests.
Example: Start in environment 1.1 and Pkg.add(D), Pkg.activate(P) and Pkg.add(D) with a different version of D (or using the local path for D). Which version of D is used after using D? The answer turns out to be that the most recent environment wins (that would usually be the currently activated project).

I encountered a case where I could not convince Julia to update an unregistered package, even using Pkg.rm followed by Pkg.add. The reason was that 1.1 referenced the same package, pointing to a fixed github commit.

### 15.2 Creating a package

#### 15.2.1 PkgTemplates.jl (1.2)

See the [Documentation](#).

#### 15.2.2 PkgSkeleton.jl (1.2)

The easiest way is PkgSkeleton.jl. You need to set your github info (user.name etc) using

```bash
git config --global user.name YourName
```

This must be done inside a git directory. Then generate generates the directory structure and the required files (Project.toml etc). Example:

```bash
PkgSkeleton.generate("dir1/MyPackage")
```

Details:

- I first create the repo on github and clone it to the local dir.

- Then I use

```bash
PkgSkeleton.generate("MyPackage", skip_existing_dir = false)
```

- This way everything is linked to github from the start.
15.3 Package workflow (1.1)

Your packages will generally be unregistered. Your workflow needs to account for the fact that Pkg does not track versions for unregistered packages.

Here are the steps:

1. Initialize a package in a folder pDir; call the package P. This generates a directory structure with src, test, etc. If you plan on using this package as a dependency, it is best to place it in a sub-folder of JULIA_PKG_DEVDIR (~/.julia/dev by default). The reason is that Pkg.develop wants to download your code there.

2. While the code is being worked on: Pkg.activate(ps). This makes sure that changes are written to the package’s environment (Project.toml).

3. To add registered dependencies, simply use Pkg.add(pkgName). No problem.

4. To add unregistered dependencies D that may change as you work on your project, use Pkg.develop instead.
   
   (a) Write code that makes a PackageSpec for D. This simplifies managing the package. Call this ps. ps should point to D’s local directory, not to a github url. Otherwise, you end up tracking what is on github rather than your local edits.

   (b) Pkg.develop(ps) simply changes the entry for D in Project.toml from pointing at the github repo to pointing at the local dir. Key point: This is only operative while the environment P is active.

   (c) Pkg.develop is an alternative to Pkg.add, which edits Project.toml to point at github.

5. To freeze the state of the code:

   (a) push P and D to github.

   (b) in the environment for P: Pkg.add(ps) where ps should now point at the github url for D.

   (c) Even if you continue to push updates for unregistered dependencies to github, your package should track the fixed versions (identified by the sha key that defines the commit). Just don’t run Pkg.update.
15.4 Unregistered packages as dependencies (1.1)

Important point: Unregistered packages need to be added as dependencies “by hand.” Pkg cannot track when other packages depend on them. This is a known issue 810. That means:

- Suppose you are working in P with dependency D that depends on E.
- Pkg.add(D) does not add E to P’s Project.toml.
- You need to explicitly Pkg.add(E).

Tracking changes in unregistered packages can be done in several ways:

- The solution suggested on discourse suggests to always develop packages and to have relative paths in Manifest.toml. That would be relative paths of the form ../MyPackage. User directory expansion, as in ~/abc does not work.
- Pkg.add(url = https://github.com/myUser/MyPkg) downloads the lastest master and recompiles the code. One option is therefore: run the code on the remote on a new environment. Add each unregistered dependency and then the main package. This is cumbersome, but can be done in a script. The key is to manually add all unregistered dependencies through that script. Pkg cannot do so automatically.
- Create your own package registry (not as hard as it sounds). Register all your packages. Then a simple Pkg.add for the code that is actually to be run will automatically download all dependencies (which are now registered).

Note:

- Pkg.update does nothing for unregistered dependencies.
- Deleting the corresponding subdirectory in ~/.julia/compiled sometimes triggers a recompile, but not always.
- revise(MyPkg) does not trigger a recompile.

For small functions that are themselves stand-alone, it seems best to simply copy them into the project. This is the old trade-off between duplication and dependencies.
15.5 Multiple Modules in one Package (1.2)

The cleanest approach is sub-modules. I.e.,

```julia
module Foo
    include("That.jl")
    include("Bar.jl")
    using .Bar, .That

    <code>
end
```

```julia
# In Bar.jl
module Bar
    using ..That
    <code>
end
```

One can still `import Foo.Bar` to only use the sub-module (especially for testing). In the test function, non-exported functions can be called as `Bar.f()`.

15.6 Testing a package (1.2)

Activate the package by issuing `activate` in the package’s directory (not in `src`). Then type `test`.

Note that the package needs the following in `Project.toml`:

```toml
[extras] Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
[targets] test = ["Test"]
```

These are not automatically added. You need to hand-edit `Project.toml`. Or simply add `Test` as a dependency directly.

Placing test code inside a module:

- This can be useful when the test code defines structs that one would like to be able to modify without having to restart Julia all the time.
Note that objects defined in tests are no longer visible once Pkg is exited.

- Place the module definition into test. Add `push(LOAD_PATH, @__DIR__)`. This has to be done in each module. Not elegant. This apparently no longer works in V1.2. It appears that tests cannot be in modules any more.

## 15.7 Creating a package registry (1.2)

Any registry that lives in `~/.julia/registries` is automatically used by Pkg.

In principle, it is easy to create your own registry (see discourse for a guide). The key to making it practical is `Registrator.jl` (not the official one, but the one from Gunnar Farneback; see below).

### 15.7.1 Using `Registrator.jl` (1.2)

This is based on the workflow figured out by a discourse user. What I am writing up here copies their code almost one-for-one.

**Creating the registry:**

```julia
Pkg.add(Pkg.PackageSpec(url="https://github.com/GunnarFarneback/Registrator.jl")
using Registrator

# Switch to the parent directory of the registry repo
cd(joinpath(homedir(), "Documents", "julia"))
regName = "registryLH"
regUrl = "https://github.com/hendi54/$regName"
regPath = joinpath(homedir(), "Documents", "julia", "registryLH")
Registrator.create_registry(regName, regUrl)
```

Create the `registryLH` repo on github.com (not sure why this is necessary). Push the registry to github using

```shell
git -C registryLH push -u -f origin master
```
Check that `Registry.toml` appears on github in the repo. Add the registry (cloning it to `.julia/registries`)

```julia
Pkg.Registry.add(Pkg.RegistrySpec(url = regUrl))
```

We now have an empty registry. Check that it can be used:

```julia
Pkg.update()
```

This should now show `registryLH` being updated.

**Adding packages to the registry.** I am using `TestPkgLH` for testing.

```
# Somehow get to the point where `using TestPkgLH` can be issued
pkgName = "TestPkgLH"
pkgRepo = "https://github.com/hendri54/TestPkgLH"
# Somehow Julia knows where this is located (how?)
Pkg.activate(pkgName)
using TestPkgLH

# Register TestPkgLH
Registrator.register(TestPkgLH, regPath)
run('git -C registryLH push origin master')
# Check on github that T/TestPkgLH appears in the repo

# Now we wish to 'add' TestPkgLH to TestPkg2LH
pkgName2 = "TestPkg2LH"
pkgRepo2 = "https://github.com/hendri54/TestPkg2LH"
Pkg.activate(pkgName2)

# Without the 'update' the 'add' fails
Pkg.Registry.update()
# Make sure that the latest version of TestPkgLH has been pushed to github
Pkg.add(pkgName)
using TestPkg2LH
Registrator.register(TestPkg2LH, regPath)
run('git -C registryLH push origin master')
Pkg.Registry.update()
```
It is currently not possible to run this from inside a module. For some reason, the wrong `Registrator` is called (or `Registrator` has no methods). So one has to do this “by hand” from the REPL:

```julia
julia> activate_pkg("UtilityFunctionsLH")
Activating environment at ‘˜/Documents/julia/UtilityFunctionsLH/Project.toml’
/Users/lutz/Documents/julia/registryLH
julia> isdir(regPath)
true
julia> using UtilityFunctionsLH
julia> register(UtilityFunctionsLH, regPath)
# These last two steps are done by ‘PackageToolsLH.update_registry()’
# Push to github
julia> Pkg.Registry.update()
```

I packaged this into a function which is used as follows:

```julia
using TestPkgLH
include("shared/ register_package.jl")
register_package(TestPkgLH)
```

Updating an existing package works in the same way. Increment the version number in `Project.toml` and register the package again.

### 15.8 Miscellaneous

Find out if a package is installed (present in current `Manifest`):

- `d = Pkg.installed()` returns a `Dict` with package names as keys.
- `haskey(d, MyPackage)` returns true if package is installed.

Adding a private repo does not work with the standard `add` syntax.

- It yields an error (“redirect from HTTPS to HTTP no allowed”).
- One needs to construct a `PackageSpec(url = ssh://git@github.com/user/MyPackage.git)`.
  This will ask for the location of the ssh key.
Multiple packages depend on the same package.

- The question: which version of the dependency gets used?
- Answer (due to Gunnar Farneback): “The general idea is that packages state which versions of their dependencies they are compatible with and it’s the job of the resolver to find a set of versions that works for all packages in the dependency chain. Another key idea is that only one version of each package can be loaded at a time, which is precisely the one that the resolver has chosen. If there are incompatible version requirements in the dependency chain the resolver will fail and you can’t load your package at all.”

16 Parallel Computing

Useful overviews: Bruel 2019

16.1 Threads

Use shared memory. Simply place Threads.@threads in front of a code section (typically a loop).

Runs on a single processor (with multiple cores).

Julia needs to be started with a command line argument that indicates the number of cores to use. Or issue `export JULIA_NUM_THREADS=8` in shell (not persistent across sessions unless written into `bash_profile`).

Keeping track of progress:

- Standard print statements are not displayed (or written to a log file) until the entire computation finishes.

- This discourse thread suggests that the solution is to use `Core.println` (as opposed to `Base.println` which is not thread-safe). Note that `Core.println` does not accept `stdout` as argument.

- Could also try to `flush(stdout)` periodically.
16.2 Distributed computation

Issue using Distributed, SharedArrays.

In front of the parallel loop, place @sync @distributed. The @sync macro ensures that the code waits for completion of the loop before it continues.

17 Performance

The compiler does not optimize out if false statements. Hence, defining a constant that switches self-testing code on and off does not result in no-ops. Of course, the overhead is quite small.

17.1 Profiling (1.3)

The output generated by the built-in profiler is hard to read. ProfileView now does compile, taking a surprisingly long time. Personally, I find the presentation of StatProfilerHTML more convenient, though.

StatProfilerHTML is a good alternative (1.1).

- It provides a flame graph with clickable links that show which lines in a function take up most time.
- Need to locate index.html and open it by hand in the browser after running statprofilehtml().

PProf.jl:

- requires Graphviz. On MacOS, install using brew install graphviz. But it has TONS of dependencies and did not install on my system. Then PProf cannot be used.

TimerOutputs.jl

- can be used to time selected lines of code
- produces a nicely formatted table that is much easier to digest than profiler output.
17.2 Type stability

One can automate checking for type stability using the `code_warntype()` function. Example:

- For function `foo(x)`, call `code_warntype(stdout, foo, (Int,1))`.
- This can be written to a file by changing the IO argument.
- It generates output even if no issues are found.
- The amount of output generated is overwhelming. Signs of trouble are `Union` types, especially return types (at `Body:`).

*Cthulhu.jl* is a tool for debugging type instability.

18 Regressions

RegressionTables.jl produces formatted regression tables.

18.1 GLM (1.2)

GLM.jl is the package to run regressions.

To save just the regression results (without the data, which could be a lot of memory), use `coeftable(mdl)`. This produces a `StatsBase.CoeffTable`.

Alternative, use `RegressionTable` from *EconometricsLH*.

Categorical regressors return names such as `Symbol(school: 3)`.

A useful introduction is in *cookbooks*.

19 Remote Clusters

19.1 Getting started with a test script

How to get your code to run on a typical Linux cluster?

- Get started by writing a simple test script (*Test3.jl*) so we can test running from the command line.
• Add the Julia binary to the PATH using (on Macos, editing ~/.bash_profile):

```bash
PATH="/Applications/Julia-1.1.app/Contents/Resources/julia/bin:$PATH"
```

• Then make sure you can run the test script locally with

```
julia /full/path/to/Test3.jl
```

Now copy Test3.jl to a directory on the cluster and repeat the same.

• You may need to add the Julia binary to the path.
  
  – On Longleaf (editing ~/.bash_profile):
    
    ```bash
    export PATH="/nas/longleaf/apps/julia/1.3.0/bin:$PATH"
    ```
  
  – The more robust approach is module add julia/1.3.0.

• Then run julia "/full/path/to/Test3.jl"

Now run the test script via batch file:

```
sbatch -p general -N 1 -J "test_job" -t 3-00 --mem 16384 -n 1 --mail-type=end --mail-user=lhendri@email.unc.edu -o "test1.out" - -wrap="julia /full/path/to/Test3.jl"
```

### 19.2 Generate an ssh key

This allows log on without password. Instructions on the web.

Now you can use the terminal to log in with ssh user@longleaf.unc.edu.

### 19.3 Rsync File Transfer

A reliable command line transfer option is rsync. The command would be something like

```
rsync -atuzv "/someDirectory/sourceDir/" "username@longleaf.unc.edu:someDirectorySourceDir"
```

Notes:

• The source dir should end in “/”; the target dir should not.

• Excluding .git speeds up the transfer.

• --delete ensures that no old files remain on the server.

To transfer an individual file: run(``scp $filename hostname:/path/to/newfile.txt’’)
19.4 Git File Transfer

1. Change into the package directory (which is already a git repo).

2. Add a remote destination (once):
   \[
   \text{git remote add longleaf ssh://lhendri@longleaf.unc.edu/nas/longleaf/home/lhendri/julia/SampleModel}
   \]

3. Initialize the remote directory with a bare repo: \texttt{git init --bare}.
   Bare means that the actual files are not copied there. It needs to be bare so \texttt{push} does not produce errors later.

4. Verify the remote: \texttt{git remote show longleaf}

When files have changed:

1. Change into the package directory

2. \texttt{git commit -am \textit{commit message}}

3. \texttt{git push longleaf master}

Note that this does not upload any files! So this only works for packages, not for code that should be run outside of packages.

19.5 Running code on the cluster

Steps:

1. Copy your code and all of its dependencies to the cluster (see Section 19.3). This is not needed when all dependencies are registered.

2. Write a Julia script that contains the startup code for the project and then runs the actual computation (call this \texttt{batch.jl}).

3. Write a batch file that submits \texttt{julia batch.jl} as a job to the cluster’s job scheduler. For UNC’s \texttt{longleaf} cluster, this would be slurm. So you need to write \texttt{job.sl} that will be submitted using \texttt{sbatch job.sl}.  

23
19.5.1 The Julia script

Submitting a job is (almost) equivalent to `julia batch.jl` from the terminal.

- Note: `cd()` does not work in these command files. To include a file, provide a full path.

If you only use registered packages, life is easy. Your code would simply say:

```julia
using Pkg
# This needs to be run only once
Pkg.add(MyPackage)
# If you want the latest version each time
Pkg.update()
using MyPackage
MyPackage.run()
```

If the code for `MyPackage` has been copied to the remote, then

```julia
julia --project="/path/to/MyPackage" --startup-file=no batch.jl
```

activates `MyPackage` and runs `batch.jl`. The `--project` option is equivalent to `Pkg.activate`.

- Julia looks for `batch.jl` in the directory that was active when Julia was invoked (in this case: when `sbatch` was invoked).

- Disabling the `startup-file` prevents surprises where the `startup-file` changes the directory before looking for `batch.jl`.

- `~` is not expanded when relative paths are used.

If `MyPackage` contains is unregistered or contains unregistered dependencies, things get more difficult. Now `batch.jl` must:

1. Activate the package’s environment.

2. `develop` all unregistered dependencies. This replaces the invalid paths to directories on the local machine (e.g. `/Users/lutz/julia/...) with the corresponding paths on the cluster (e.g. `/nas/longleaf/...`).

   Note: I verified that one cannot replace `homedir()` with `~` in `Manifest.toml`. 

24
3. using MyPackage

4. MyPackage.run()

Developing MyPackage in a blank folder does not work (for reasons I do not understand). It results in errors indicating that dependencies of MyPackage could not be found.

This approach requires you to keep track of all unregistered dependencies and where they are located on the remote machine. My way of doing this is contained in PackageTools.jl in the shared repo (this is not a package b/c its very purpose is to facilitate loading of unregistered packages).

For an example implementation of the entire process, see batch_commands.jl in TestPkg2LH.

- This uses PackageToolsLH to handle directories on different computers and file transfer.
- write_command_file() writes the julia file that is to be executed remotely (command_file.jl).
- write_sbatch writes the sbatch file that will be submitted to slurm.
- project_upload() uses rsync to copy the code of the project, its dependencies, and some general purpose code that is required at startup (mainly PackageToolsLH itself) to the remote machine.

### 19.5.2 The sbatch file

How this works can be looked up online. The only trick is that the Julia command requires a full path (or a relative path, but that’s a little risky) on the remote machine.

FilesLH keeps track of where things are on each machine. It is used to build the full paths.

### 19.5.3 Instantiating Packages

If all dependencies are registered, simply activate an environment and `pkg add https://github.com/user/MyPackage.git` followed by `using MyPackage`. 
When packages are run, all dependencies must be installed. This would usually be done with `instantiate`. But this fails when the package is developed rather than added. Therefore: if a package fails to build or test (for example, after its first upload, or after new dependencies are installed that the remote machine does not have installed):

1. An indicator that a dependency is missing is the error message: **ERROR:**
   
   ```
   MethodError: Cannot ‘convert’ an object of type Nothing to an object of type Base.SHA1
   ```

2. Switch to a test environment where one can mess up the `Project.toml`

3. `Pkg.add(ps)` where `ps` is the `PackageSpec` for the package that does not build. It must point at the `github` url.

4. This is not always enough. In that case, activate the package that does not build. Use `>pkg st -m` to show the packages that are not loaded and simply `add` them until the package builds and tests.

Now the package can be built or developed everywhere.

Sometimes old versions of `Project.toml` lie around somewhere (where?) in the Julia installation. They may contain dependencies that don’t exist anymore. Then the package does not build. The only solution that seems to work: `Pkg.add` the package from somewhere with a `PackageSpec` that points at `github`.

- For this purpose, it is useful to have an environment lying around that is just for adding packages that need to be downloaded.

### 20 Types (1.3)

I find it easiest to write model specific code NOT using parametric types. Instead, I define type aliases for the types used in custom types (e.g., `Double=Float64`). Then I hardwire the use of `Double` everywhere. This removes two problems:

1. Possible type instability as the compiler tries to figure out the types of the custom type fields.

2. It becomes possible to call constructors with, say, integers of all kinds without raising method errors.
20.1 Constructors (1.1)

Constructing objects with many fields:

- Define an inner constructor that leaves the object (partially) uninitialized. It is legal to have new(x) even if the object contains additional fields.

Parameters.jl is useful for objects with default values.

- Constructor must then provide all arguments that do not have defaults.

- Note that @with_kw automatically defines show(). Use @with_kw_noshow to avoid this.

20.2 Inheritance (1.1)

There is no inheritance in Julia. Abstract types have no fields and concrete types have no subtypes.

There are various discussions about how to implement types that share common fields.

For simple cases, it is probably best to just repeat the fields in all types. This can be automated using @forward in Lazy.jl.

One good piece of advice: ensure that methods are generally defined on the abstract type, so that all concrete types have the same interface (kind of the point of having an abstract type).

20.3 Loading and saving (1.1)

using FileIO and extension .jld2 automatically saves in jld2 format. This can save used defined types.

Loading user defined types is more complicated. All modules needed to construct the loaded types need to be known in the loading module and in Main. See Issue 134. It is not possible to use Core.eval(Main, :(using Module)) for unclear reasons.

Implications:

1. Each user defined type needs its own load function.
2. All dependencies need to imported into Main “by hand” for each loaded object.

An alternative is BSON.jl. It has the same limitation. One could save the ParamVectors in each object and reconstruct the object from those (recursively). This, of course, only works for objects that can be constructed from ParamVectors. Each ParamVector could be stored as a Dict{Symbol, Any}. But even easier: store the ParamVectors directly. Constructing them after loading only requires modelLH. The approach would then be:

1. Collect the ParamVectors from all model objects into a Dict{Symbol,ParamVector}. The symbol identifies the associated model object.
2. Save the Dict.
3. In Main: using modelLH, so that loading works.
4. Function that loads the model:
   (a) Construct the model object with arbitrary default values.
   (b) Load the ParamVectors.
   (c) Sync each ParamVector’s parameters into the correct model object. Essentially, the model object needs a constructor that accepts a ParamVector.

21 Unit Testing (1.2)

Goals:

1. Ensure that tests are self-contained, so that each can be run independently.

My current approach:

1. Place each group of tests into a module, so the tests are independent of each other and can be run independently. SafeTestsets.jl has a similar idea, but I find it cleaner to explicitly write out the modules.
2. `runtests.jl` simply contains a list of `include` statements; one for each test module. Those are wrapped in a `@testset` for nice display and to ensure that errors don’t stop the tests.

3. Each test module also contains a `@testset`.

4. When `runtests` is run, it displays a single success summary. But when there are errors, they are nicely broken down by `testset`.

5. To run tests selectively, simply `include` the file that contains the `@testset` at the REPL.

Errors in the code to be tested (but not caught by `@test`) cause the entire test run to crash. Preventing this requires all tests to be enclosed in a `@testset`. A sequence of `@testset` does not do the trick. An error in one prevents all others from being run. Nested `@testset`s produce nested error reports (nice).

`@test` statements can be placed inside functions. To preserve result reporting, the function should contain a `@testset` and return its result.

Test dependencies now need to be added to the `Project.toml` file in `./test`.

### 21.1 Travis CI (1.2)

Travis can automatically test all branches uploaded to github.

Need to customize `travis.yml` to only build for the current Julia version.

Building with unregistered dependencies is tricky. Probably ok if the dependencies are added (so they point to a github url), but not if they are developed.

### 22 Workflow (1.2)

`Revise` is key. It is now possible to simply use `using` on any module once. `Revise` then automatically keeps track of changes. Using `includet` creates problems for me.

But keep in mind that `Revise` cannot handle:

1. changes in file structure (you factor out some code into a new file that is `included` in the main file);
2. changes in structs.

Those still require restarting the REPL.