A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system – Gall’s Law.

Co-authored with Arnav Sood

This lecture discusses structuring a project as a Julia module, and testing it with tools from GitHub.

Benefits include

- Specifying dependencies (and their versions) so that your project works across Julia setups and over time.
- Being able to load your project’s functions from outside without copy/pasting.
- Writing tests that run locally, and automatically on the GitHub server.
- Having GitHub test your project across operating systems, Julia versions, etc.

2 Project Setup

2.1 Account Setup

2.1.1 Travis CI

As we’ll see later, Travis is a service that automatically tests your project on the GitHub server.
First, we need to make sure that your GitHub account is set up with Travis CI and Codecov.

As a reminder, make sure you signed up for the GitHub Student Developer Pack or Academic Plan if eligible.

Navigate to the travis-ci.com website and click “sign up with GitHub” – supply your credentials.

If you get stuck, see the Travis tutorial.

2.1.2 Codecov

Codecov is a service that tells you how comprehensive your tests are (i.e., how much of your code is actually tested).

To sign up, visit the Codecov website, and click “sign up”

Next, click “add a repository” and enable private scope (this allows Codecov to service your private projects).

The result should be
This is all we need for now.

2.2 Julia Setup

In [1]: using InstantiateFromURL  
   # optionally add arguments to force installation: instantiate = true,  
   # precompile = true  
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

Note: Before these steps, make sure that you’ve either completed the version control lecture or run.

```
git config --global user.name "Your Name"
```

Note: Throughout this lecture, important points and sequential workflow steps are listed as bullets.  
To set up a project on Julia:  
- Load the PkgTemplates package.

```
using PkgTemplates
```

- Create a template for your project.

This specifies metadata like the license we’ll be using (MIT by default), the location (~/.julia/dev by default), etc.

```
ourTemplate = Template(;user="quanteconuser", plugins = [TravisCI(), Codecov()])
```

Note: Make sure you replace the quanteconuser with your GitHub ID.

- Create a specific project based off this template
generate("ExamplePackage.jl", ourTemplate)

If we navigate to the package directory, we should see something like the following.

As a reminder, the location of your `.julia` folder can be found by running `DEPOT_PATH[1]` in a REPL.

**Note:** On Mac, this may be hidden; you can either start a terminal, `cd ~` and then `cd .julia`, or make hidden files visible in the Finder.

### 2.3 Adding a Project to Git

The next step is to add this project to Git version control.

- Open the repository screen in your account as discussed previously.

We’ll want the following settings

In particular

- The repo you create should have the same name as the project we added.
- We should leave the boxes unchecked for the `README.md`, `LICENSE`, and `.gitignore`, since these are handled by `PkgTemplates`.

Then,
 Drag and drop your folder from your ~/.julia/dev directory to GitHub Desktop.

- Click the “publish branch” button to upload your files to GitHub.

If you navigate to your git repo (ours is here), you should see something like

![GitHub Repository](image)

**Note:** Be sure that you don’t separately clone the repo you just added to another location (i.e., to your desktop).

A key note is that you have some set of files on your local machine (here in ~/.julia/dev/ExamplePackage.jl) and git is plugged into those files.

For convenience, you might want to create a shortcut to that location somewhere accessible.

### 2.4 Adding a Project to the Julia Package Manager

We also want Julia’s package manager to be aware of the project.

- Open a REPL in the newly created project directory, either by noting the path printed above, or by running the following in a REPL.

```
cd(joinpath(DEPOT_PATH[1], "dev", "ExamplePackage"))
```

Note the lack of .jl!

- Run the following

```
] activate
```

(to get into the main Julia environment (more on environments in the second half of this lecture).

- And run

```
] dev .
```

(to add the package.

You can see the change reflected in our default package list by running

```
] st
```

For more on the package mode, see the tools and editors lecture.)
2.5 Using the Package Manager

Now, from any Julia terminal in the future, we can run

```julia
using ExamplePackage
```

To use its exported functions.

We can also get the path to this by running

```julia
using ExamplePackage
pathof(ExamplePackage) # returns path to src/ExamplePackage.jl
```

3 Project Structure

Let’s unpack the structure of the generated project

- The first directory, `.git`, holds the version control information.
- The `src` directory contains the project’s source code – it should contain only one file (`ExamplePackage.jl`), which reads

```julia
module ExamplePackage

greet() = print("Hello World!")
end # module
```

- Likewise, the `test` directory should have only one file (`runtests.jl`), which reads

```julia
using ExamplePackage
using Test

@testset "ExamplePackage.jl" begin
    # Write your own tests here.
end
```

In particular, the workflow is to export objects we want to test (`using ExamplePackage`), and test them using Julia’s `Test` module.

The other important text files for now are

- `Project.toml` and `Manifest.toml`, which contain dependency information.

In particular, the `Project.toml` contains a list of dependencies, and the `Manifest.toml` specifies their exact versions and sub-dependencies.

- The `.gitignore` file (which may display as an untitled file), which contains files and paths for `git` to ignore.
4 Project Workflow

4.1 Dependency Management

4.1.1 Environments

As before, the .toml files define an environment for our project, or a set of files which represent the dependency information.

The actual files are written in the TOML language, which is a lightweight format to specify configuration options.

This information is the name of every package we depend on, along with the exact versions of those packages.

This information (in practice, the result of package operations we execute) will be reflected in our ExamplePackage.jl directory’s TOML, once that environment is activated (selected).

This allows us to share the project with others, who can exactly reproduce the state used to build and test it.

See the Pkg3 docs for more information.

4.1.2 Pkg Operations

For now, let’s just try adding a dependency

- Activate the package environment (to be run from the base, v1.1 environment)

] activate ExamplePackage

This tells Julia to write the results of package operations to ExampleProject’s TOML, and use the versions of packages specified there.

Note that the base environment isn’t special, except that it’s what’s loaded by a freshly-started REPL or Jupyter notebook.

- Add a package

] add Expectations

We can track changes in the TOML, as before.

Here’s the Manifest.toml
We can also run other operations, like ] up, ] precompile, etc.

Package operations are listed in detail in the tools and editors lecture.

Recall that, to quit the active environment and return to the base (v1.1), simply run

] activate

4.2 Writing Code

The basic idea is to work in tests/runtests.jl, while reproducible functions should go in the src/ExamplePackage.jl.

For example, let’s say we add Distributions.jl

] activate ExamplePackage

] add Distributions

and edit the source (paste this into the file itself) to read as follows

```julia
module ExamplePackage

greet() = print("Hello World!")

using Expectations, Distributions

function foo(μ = 1., σ = 2.)
    d = Normal(μ, σ)
    E = expectation(d)
    return E(x -> sin(x))
end

export foo

end # module
```

Let’s try calling this

] activate

using ExamplePackage
ExamplePackage.greet()

foo() # exported, so don't need to qualify the namespace

Note: You may need to quit your REPL and load the package again.
4.3 Jupyter Workflow

We can also work with the package from a Jupyter notebook. Let’s create a new output directory in our project, and run `jupyter lab` from it. Create a new notebook `output.ipynb`.

From here, we can use our package’s functions as we would functions from other packages. This lets us produce neat output documents, without pasting the whole codebase. We can also run package operations inside the notebook.

The change will be reflected in the `Project.toml` file. Note that, as usual, we had to first activate `ExamplePackage` first before making our dependency changes.

```plaintext
name = "ExamplePackage"
uuid = "f85830d0-e1f0-11e8-2fad-8762162ab251"
authors = ["QuantEcon User <quanteconuser@gmail.com>"]
version = "0.1.0"

[deps]
Distributions = "31c24e10-a181-5473-b8eb-7969acd0382f"
Expectations = "2fe49d83-0758-5602-8f54-1f90ad0d522b"
Parameters = "d96e819e-fc66-5662-84c9c7592b0a"

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

[targets]
test = ["Test"]
```
There will also be changes in the Manifest as well.

Be sure to add `output/.ipynb_checkpoints` to your `.gitignore` file, so that’s not checked in.

Make sure you’ve activated the project environment (`] activate ExamplePackage`) before you try to propagate changes.

### 4.4 Collaborative Work

For someone else to get the package, they simply need to

- Run the following command

  ```
  ] dev https://github.com/quanteconuser/ExamplePackage.jl.git
  ```

  This will place the repository inside their `~/.julia/dev` folder.
  - Drag-and-drop the folder to GitHub desktop in the usual way.

Recall that the path to your `~/.julia` folder is

```
In [2]: DEPOT_PATH[1]
```

```
Out[2]: "/home/ubuntu/.julia"
```

They can then collaborate as they would on other git repositories.
In particular, they can run

```
] activate ExamplePackage
```

 to load the list of dependencies, and

```
] instantiate
```

to make sure they are installed on the local machine.

### 5 Unit Testing

It’s important to make sure that your code is well-tested.

There are a few different kinds of test, each with different purposes

- *Unit testing* makes sure that individual pieces of a project work as expected.
- *Integration testing* makes sure that they fit together as expected.
- *Regression testing* makes sure that behavior is unchanged over time.

In this lecture, we’ll focus on unit testing.

In general, well-written unit tests (which also guard against regression, for example by comparing function output to hardcoded values) are sufficient for most small projects.
5.1 The **Test** Module

Julia provides testing features through a built-in package called **Test**, which we get by **using Test**.

The basic object is the macro **@test**

```julia
using Test
@test 1 == 1
@test 1 ≈ 1
```

**Out[3]: Test Passed**

Tests will pass if the condition is **true**, or fail otherwise.

If a test is failing, we should flag it with **@test_broken** as below

```julia
@test_broken 1 == 2
```

**Out[4]: Test Broken**

```
Expression: 1 == 2
```

This way, we still have access to information about the test, instead of just deleting it or commenting it out.

There are other test macros, that check for things like error handling and type-stability.

Advanced users can check the **Julia docs**.

5.2 Example

Let’s add some unit tests for the **foo()** function we defined earlier.

Our **tests/runtests.jl** file should look like this.

As before, this should be pasted into the file directly

```julia
using ExamplePackage
using Test
@test foo() ≈ 0.11388071406436832
@test foo(1, 1.5) ≈ 0.2731856314283442
@test_broken foo(1, 0) # tells us this is broken
```

And run it by typing `] test` into an activated REPL (i.e., a REPL where you’ve run `] activate ExamplePackage`).

5.3 Test Sets

By default, the **runtests.jl** folder starts off with a **@testset**.

This is useful for organizing different batches of tests, but for now we can simply ignore it.

To learn more about test sets, see the **docs**.
5.4 Running Tests

There are a few different ways to run the tests for your package.

- Run the actual runtests.jl, say by hitting \texttt{shift-enter} on it in Atom.
- From a fresh (v1.1) REPL, run \texttt{] test ExamplePackage}.
- From an activated (ExamplePackage) REPL, simply run \texttt{] test} (recall that you can activate with \texttt{] activate ExamplePackage}).

6 Continuous Integration with Travis

6.1 Setup

By default, Travis should have access to all your repositories and deploy automatically.

This includes private repos if you’re on a student developer pack or an academic plan (Travis detects this automatically).

To change this, go to “settings” under your GitHub profile

Click “Applications,” then “Travis CI,” then “Configure,” and choose the repos you want to be tracked.

6.2 Build Options

By default, Travis will compile and test your project (i.e., “build” it) for new commits and PRs for every tracked repo with a \texttt{.travis.yml} file.
We can see ours by opening it in Atom

```yaml
# Documentation: http://docs.travis-ci.com/user/languages/julia/
language: julia
os:
  - linux
  - osx
julia:
  - 1.1
  - nightly
matrix:
  allow_failures:
    - julia: nightly
fast_finish: true
notifications:
  email: false
after_success:
  - julia -e 'using Pkg; Pkg.add("Coverage"); using Coverage; Codecov.submit(process_folder())'
```

This is telling Travis to build the project in Julia, on OSX and Linux, using Julia v1.1 and the latest development build (“nightly”).

It also says that if the nightly version doesn’t work, that shouldn’t register as a failure.

**Note** You won’t need OSX unless you’re building something Mac-specific, like iOS or Swift.

You can delete those lines to speed up the build, likewise for the nightly Julia version.

### 6.3 Working with Builds

As above, builds are triggered whenever we push changes or open a pull request.

For example, if we push our changes to the server and then click the Travis badge (the one which says “build”) on the README, we should see something like
Note that you may need to wait a bit and/or refresh your browser. This gives us an overview of all the builds running for that commit. To inspect a build more closely (say, if it fails), we can click on it and expand the log options.

Note that the build times here aren’t informative, because we can’t generally control the hardware to which our job is allocated. We can also cancel specific jobs, either from their specific pages or by clicking the grey “x” button on the dashboard. Lastly, we can trigger builds manually (without a new commit or PR) from the Travis overview.

To commit *without* triggering a build, simply add “[ci skip]” somewhere inside the commit message.

### 6.4 Travis and Pull Requests

One key feature of Travis is the ability to see at-a-glance whether PRs pass tests before merging them. This happens automatically when Travis is enabled on a repository. For an example of this feature, see this PR in the [Games.jl](https://github.com/JuliaGPU/Games.jl) repository.
7 Code Coverage

Beyond the success or failure of our test suite, we also want to know how much of our code the tests cover.

The tool we use to do this is called Codecov.

7.1 Setup

You’ll find that Codecov is automatically enabled for public repos with Travis.

For private ones, you’ll need to first get an access token.

Add private scope in the Codecov website, just like we did for Travis.

Navigate to the repo settings page (i.e., https://codecov.io/gh/quanteconuser/ExamplePackage for our repo) and copy the token.

Next, go to your Travis settings and add an environment variable as below

7.2 Interpreting Results

Click the Codecov badge to see the build page for your project.

This shows us that our tests cover 50% of our functions in src/.

Note: To get a more detailed view, we can click the src/ and the resultant filename.

Note: Codecov may take a few minutes to run for the first time
This shows us precisely which methods (and parts of methods) are untested.

8 Pull Requests to External Julia Projects

As mentioned in version control, sometimes we’ll want to work on external repos that are also Julia projects.

• ] dev the git URL (or package name, if the project is a registered Julia package), which will both clone the git repo to ~/.julia/dev and sync it with the Julia package manager.

For example, running

] dev Expectations

will clone the repo https://github.com/quantecon/Expectations.jl to ~/.julia/dev/Expectations.

Make sure you do this from the base Julia environment (i.e., after running ] activate without arguments).

As a reminder, you can find the location of your ~/.julia folder (called the “user depot”), by running

In [5]: DEPOT_PATH[1]

Out[5]: "/home/ubuntu/.julia"

The ] dev command will also add the target to the package manager, so that whenever we run using Expectations, Julia will load our cloned copy from that location

using Expectations
pathof(Expectations)  # points to our git clone
• Drag that folder to GitHub Desktop.
• The next step is to fork the original (external) package from its website (i.e., https://github.com/quantecon/Expectations.jl) to your account (https://github.com/quanteconuser/Expectations.jl in our case).

• Edit the settings in GitHub Desktop (from the “Repository” dropdown) to reflect the new URL.

Here, we’d change the highlighted text to read quanteconuser, or whatever our GitHub ID is.

• If you make some changes in a text editor and return to GitHub Desktop, you’ll see something like.

  ![GitHub Desktop settings](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.

  ![Revision in text editor](image)

  Here, for example, we’re revising the README.
Clicking “commit to master” (recall that the checkboxes next to each file indicate whether it’s to be committed) and then pushing (e.g., hitting “push” under the “Repository” dropdown) will add the committed changes to your account.

To confirm this, we can check the history on our account here; for more on working with git repositories, see the version control lecture.

The green check mark indicates that Travis tests passed for this commit.

Clicking “new pull request” from the pull requests tab will show us a snapshot of the changes, and let us create a pull request for project maintainers to review and approve.

For more on PRs, see the relevant section of the version control lecture.

For more on forking, see the docs on GitHub Desktop and the GitHub Website.

### 8.1 Case with Write Access

If you have write access to the repo, we can skip the preceding steps about forking and changing the URL.

You can use `] dev` on a package name or the URL of the package

```
] dev Expectations
```

or `] dev https://github.com/quanteconuser/Expectations.jl.git` as an example for an unreleased package by URL.

Which will again clone the repo to `~/.julia/dev`, and use it as a Julia package.
using Expectations
pathof(Expectations) # points to our git clone

Next, drag that folder to GitHub Desktop as before.
Then, in order to work with the package locally, all we need to do is open the 
~/.julia/dev/Expectations in a text editor (like Atom)

From here, we can edit this package just like we created it ourselves and use GitHub Desktop to track versions of our package files (say, after ] up, or editing source code, ] add Package, etc.).

8.2 Removing a Julia Package

To “un-dev” a Julia package (say, if we want to use our old Expectations.jl), you can simply run

] free Expectations

To delete it entirely, simply run

] rm Expectations

From a REPL where that package is in the active environment.

9 Benchmarking

Another goal of testing is to make sure that code doesn’t slow down significantly from one version to the next.
We can do this using tools provided by the BenchmarkTools.jl package.
See the need for speed lecture for more details.
10 Additional Notes

- The JuliaCI organization provides more Julia utilities for continuous integration and testing.

11 Review

To review the workflow for creating, versioning, and testing a new project end-to-end.

1. Create the local package directory using the PkgTemplates.jl.

2. Add that package to the Julia package manager, by opening a Julia REPL in the ~/.julia/dev/ExamplePackage.jl, making sure the active environment is the default one (v1.1), and hitting ] dev .

3. Drag-and-drop that folder to GitHub Desktop.

4. Create an empty repository with the same name on the GitHub server.

5. Push from GitHub Desktop to the server.

6. Open the original project folder (e.g., ~/.julia/dev/ExamplePackage.jl) in Atom.

7. Make changes, test, iterate on it, etc. As a rule, functions like should live in the src/ directory once they’re stable, and you should export them from that file with export func1, func2. This will export all methods of func1, func2, etc.

8. Commit them in GitHub Desktop as you go (i.e., you can and should use version control to track intermediate states).

9. Push to the server, and see the Travis and Codecov results (note that these may take a few minutes the first time).

12 Exercises

12.1 Exercise 1

Following the instructions for a new project, create a new package on your github account called NewtonsMethod.jl.

In this package, you should create a simple package to do Newton’s Method using the code you did in the Newton’s method exercise in Introductory Examples.

In particular, within your package you should have two functions

- newtonroot(f, f'; x₀, tol = 1E-7, maxiter = 1000)
- newtonroot(f; x₀, tol = 1E-7, maxiter = 1000)

Where the second function uses Automatic Differentiation to call the first.

The package should include

- implementations of those functions in the /src directory
• comprehensive set of tests
• project and manifest files to replicate your development environment
• automated running of the tests with Travis CI in GitHub

For the tests, you should have at the very minimum

• a way to handle non-convergence (e.g. return back nothing as discussed in error handling)
• several @test for the root of a known function, given the $f$ and analytical $f'$ derivatives
• tests of those roots using the automatic differentiation version of the function
• test of finding those roots with a BigFloat and not just a Float64
• test of non-convergence for a function without a root (e.g. $f(x) = 2 + x^2$)
• test to ensure that the maxiter is working (e.g. what happens if you call `maxiter = 5`)
• test to ensure that tol is working

And anything else you can think of. You should be able to run ] test for the project to check that the test-suite is running, and then ensure that it is running automatically on Travis CI.

Push a commit to the repository which breaks one of the tests and see what the Travis CI reports after running the build.

12.2 Exercise 2

Watch the youtube video Developing Julia Packages from Chris Rackauckas. The demonstration goes through many of the same concepts as this lecture, but with more background in test-driven development and providing more details for open-source projects.