Lecture Notes 2:
Git and Programming Tutorial

Ivan Rudik

ECON 509: Computational Methods

August 29, 2017

Programming boils down to writing a set of instructions. Unfortunately this set of instructions is not written in plain English, but one of many computing languages. Much like English, computing languages have hard and fast rules that can't be violated, or else the instructions will not make sense and the computer will throw out an error that it cannot continue. Similarly, there are also elements of style in programming as in conventional writing (think Strunk and White) that make for clearer and more efficient code. You want your code to be as simple, clear and efficient as possible while getting the job done. This will make it easier when you come back to your code in six months or if someone else pokes around your replication files in the *American Economic Review*. This lecture note will briefly go over how to use a version control and team coding tool called "Git." We will also take a 30,000 foot view of programming, and key Julia features and style suggestions (rules) for novice programmers.

1 Git

When coding up numerics for (or just writing) a paper, ideally you would be able to work on the project at the same time as your collaborators. Using standard cloud storage actually doesn't allow for this very well, only one person can alter files at a time. Moreover, you may want to keep track of all of the versions of your code as you progress from the early stages where you might just be simulating a dynamic two firm model of cournot competition, to future versions where you add in stochastic research developments and entry of new firms. Traditionally you would save the older versions with different file names like `model_old_datehere.jl`, but that clutters your directories and does not necessarily make it easy to take a few steps back if there's a bug you can't find in

your new version.

Enter git:[1] *Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.* What Git allows you to do is have only one set of files on your local machine while keeping track of all the different versions, or branches, of your code that you have worked through. Because of this, it also keeps track of all changes to your code so if you know a bug crept into your code around some specific date it's easy to find where it is. Git also allows for your code to be hosted remotely (GitHub, Bitbucket, etc) so that it is effectively backed up, and so that two people on a project can work on their code at the same time. How does this work? When two people make independent changes to their local versions of some file, they must eventually "sync" the remote version with their own local version. Upon "syncing", each person can select which changes of the other persons to keep. We will not go over the full functionality of Git, but only the basics and what you will need to effectively version control your own code. You will also need to be able to use Git to submit homework assignments. Figure 1 displays a graphic of what Git "looks like". As we move up the graphic we are moving forward in time, with each dot denoted an updated copy of the code with comments on what changed in that version and who made the changes. Moreover, there may be different "branches" of the code which signify different parts of your development environment. In this example, there's a branch for the climate-economy model DICE with 4 states, but also branches for different versions with 5 states, 7 states, and others that capture history states as well. We can switch between these different branches or move back in time along any given branch. We can do all of this while having only one folder on your computer and one set of files, no matter how many updates you might have on a given branch.

Why should you do this instead of what you've likely done before with Dropbox and what most researchers do? Because it's better. See http://www.brown.edu/Research/Shapiro/pdfs/CodeAndData.pdf for more detail on version control and a general meta view of research.

1.1 Installing Git

First, go to http://github.com and start an account using your Iowa State e-mail. Next, go to https://git-scm.com/downloads to download Git. From there, you can also download the SourceTree GUI client which we will be using in this course. In the future you may wish to switch to the GitHub client or just use terminal, or use an alternative to GitHub for your own work. Once you have installed Git and SourceTree you will want to go into SourceTree and link your remote GitHub account to your local SourceTree client by going to the Clone/New window and then browsing hosted projects.

[1] Fun fact: Git was developed by the creator of Linux, Linus Torvalds.

Figure 1: Repository in SourceTree.

1.2 Git Basics

You've installed Git and linked it up to your GitHub account. Now what *is* Git? Git is software that allows you to host a *local repository*, or local repo, while also having your code on a *remote repo*, which by default is called *origin*. In Git, each new update is called a *commit*, and they are represented by the big dots in Figure 1. The current commit your local repo is pointed at is designated *HEAD*. Within a repo, you can also be on different *branches* for when you are adding new large scale features that are in some sense their own project. Your main/initial branch is given the name *master*. If you are solving a big non-linear dynamic program using one scheme, and you wish to test out another to see if it's faster or gets you more accuracy, you might want to create a new branch for it. This allows you to have several versions of the same project where you are doing completely different things or just testing some things out. You can keep it on a totally separate branch if you want to have two different versions of your project, or you can *merge* it back into the master branch if it'll be your main code going forward.

So how do we do all of this? Figure 2 displays the flow of Git. The first step is to *clone* the remote repo for this class. What this does is it takes the files on our class's remote GitHub repo and puts them in a particular folder on your own machine. Once you've cloned `FALL2017` into a folder, you can take a look at the syllabus and lecture notes that are now all conveniently on your machine. This is how you can take publicly available repos that exist online and move them onto your own computer and also into your local repo. Alternatively, you can always create a new repo on Github, then you can go into your Git client or command line and clone it onto your machine. Call this repo "Test".

We can then go ahead and put a file into our Test repo. For example, you may wish to create

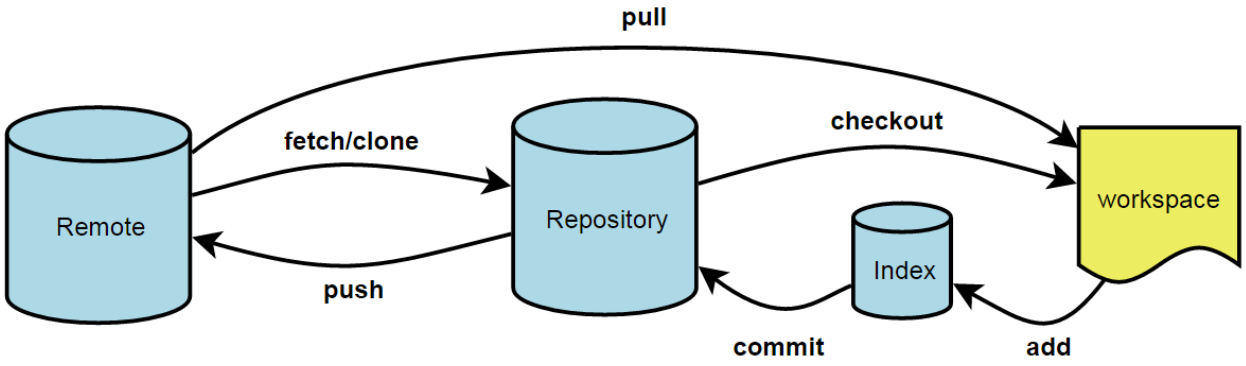

Figure 2: The flow of git: https://illustrated-git.readthedocs.org/en/latest/

a text file containing "Hello world!" and save it in your repo folder on your machine. Note that this file is now saved on your machine, but not in your local repo. You first have to *add/stage* it to the index and then can then *commit* this change to your local repo along with a note containing information on what you have changed.[2] Once you have committed the change, it has effectively been saved in your local repository. Your computer's Git now knows about this update and the previous version. Next you can *push* your changes to your remote repo. If you look at your Github repo using a browser, you will see the text file is now there!

What if you want to make changes? Just edit the file on your machine and add "Edit for changing!" Once you have made this change and saved it, GUI clients like SourceTree will tell you that there are uncommitted changes in your repo. If you want to save these changes to your local repo, you *stage* the changes you want to commit. Then you commit them. Once they are committed, the staged changes can be pushed to your remote repo. If you check Github from your browser again, you will see your changes are there.

This is the basics of using Git. Since Git saves all these changes to files in your repo, you can always move back to alternative versions of your files.

2 A Broad View of Programming

A program is the end product of programming. It can be used to develop an e-mail client or a dynamic stochastic model of the economy with non-convexities. A program is made up of different components and these components are also made of their own subcomponents. The most basic component (for our intents and purposes) is a *statement*. A statement is analogous to a sentence

[2]These are combined commands in SourceTree.

in a book. It has its own structure and intent, but it doesn't mean that much without the other statements that go with it. A statement is more commonly called *a line of code* because statements are written on individual lines. Here's a program:

```
deck = [''4 of Hearts'', ''King of Clubs'', ''Ace of Spades'']
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println(first_card)
```

This program is really simple and short: it creates a deck of cards, shuffles them, and displays the top one. But, there's a lot going on underneath the surface you don't see. What are the parentheses and why are they different from square brackets? How does shuffle work? What's println? We don't need to answer these questions but it's important to know that a good program has code, like the above, which is understandable to other programmers while performing efficiently.

2.1 Data Types

The next step up from statements is *data types*. Data types differentiate between numbers, strings of letters or other characters, logical values, and other ways to store information. Handling different data types is usually easy. Strings are in quotes "like this". Numbers are just numbers: 3.14159. And logical values, called Boolean variables, are just TRUE or FALSE.

In Julia things can get a bit more complicated in that Julia is what's called a *strongly typed* language: a variable's type will matter quite of a lot for how efficient your code is. For example, try the following code

```
typeof(1) == typeof(1.0)
```

This will return false even though both numbers are one. The reason is that 1 is an integer (Int64), while 1.0 is a floating point number (Float64). In theory this shouldn't matter much but in practice it does.[3] Julia is fast because a strongly typed language allows programmers to maintain *type stability*: the compiler can know that a variable that starts off an integer will stay as an integer, allowing it to effectively behave like assembly/C/FORTRAN. How can Julia do this when we need to define functions like addition (+) on floating point numbers and integers? It does it by *multiple-dispatch*: allowing functions to have different methods. We can define what a function does when it's given two integers, or two floating point numbers. So when the compiler knows that the type of the variable is stable, and it's told the type, it can immediately go to the

[3]Hopefully you did the reading and will have a coarse understanding of why.

right version of the function without having to check the type of the inputs or having to convert a variable from one type to another.

The key ones that are of use to us are:

1. Arrays: these allow us to store information like numbers, boolean variables, and strings. These can be used for matrix operations.
   - `A = [1 2; 3 4]`
2. Vectors: These are dimensionless arrays.
   - `A = [1, 2, 3, 4]`
3. Dicts: These are variables with key-value relationships. You can call a dictionary value by a name and it'll return a value for you.
   - `A = Dict{String,Float64}(''keyword'' => value)`
4. Tuples: Tuples are like arrays but are *immutable*. Once they're defined, the values cannot be changed.
   - `A = (1, 3, 4, 5)`
5. Your own: A neat feature of Julia is that you can create your own types. These can be the union of two others (an Array of Float64 and Int8), or something entirely new (geographic coordinates).

Data types can be nested within each other. You can have an array of vectors, or an array of tuples, or a tuple of arrays.

2.2 Variables

We need a way to label our data. Having a symbol for pi out to 100 digits is much easier to handle than continually re-typing the actual number. Moreover, we won't know what values our data or parameters will take ahead of time. This allows the value (but not the type for fast code!) of a quantity to change as we work through the program. This is very important for recursive modeling in economics but is often taken for granted.

2.3 Control Flow

Programs are read from top to bottom and left to right like the English language. But sometimes in programming we don't know ahead of time if we want to execute a line of code or not. Sometimes we want to execute line of code millions of times with slight differences (indexing, parameter value, etc) but writing out a million lines of code is clearly not ideal. There are several basic tools of control flow. The first is the if statement:

```
if marginal_benefit > marginal_cost
        quantity_produced = quantity_produced + 1
end
```

If the logical argument in the if statement is true (marginal benefit > marginal cost), then the program will execute the code within the if statement. However we may want to execute some other code when the logical statement is false,

```
if marginal_benefit > marginal_cost
        quantity_produced = quantity_produced + 1
else
        quantity_produced = quantity_produced - 1
end
```

But that isn't exactly what we want to do since we may actually reduce the quantity produced when we're at the optimality condition of marginal benefit == marginal cost. This is where we need elseif statements,

```
if marginal_benefit > marginal_cost
        quantity_produced = quantity_produced + 1
elseif marginal_benefit < marginal_cost
        quantity_produced = quantity_produced - 1
else
        println("We are at the optimum.")
end
```

This allows us to test a sequence of logical statements, only progressing to the next logical statement if the current logical statement is false.

The second control flow tool is a *loop*. Loops allow you to execute the same lines of code an arbitrary number of times:

```
quantity_produced = 0
```

```
while marginal_benefit > marginal_cost
        quantity_produced = quantity_produced + 1
        marginal_benefit, marginal_cost = FOCs(quantity_produced)
end
```

Instead of having an unknown number of if statements to try to find the optimum quantity, we can use a while loop to iterate as many times as we need. A while loop executes the code inside the loop until the logical statement (marginal benefit > marginal cost) is no longer true. So what we can do to find the optimum in this example is to increase the quantity produced by 1, plug the quantity into a function that calculates the first order conditions and then see if marginal benefit is still greater than marginal cost. If it is, continue the process. If not, stop and we have our optimal quantity produced.[4]

The more widely used loop is the *for loop*. A for loop differs from a while loop in that it introduces an new index variable, and moves through a vector of values[5] of arbitrary length. Consider this example where we have a constant marginal cost, and we know the intercept term and a set of slope coefficients of our linear marginal revenue curve. We want to see how the optimal quantity changes in the slope of the marginal revenue curve.

```
slope_set = [0, 1, 4, 9, 13]
intercept = 6
marginal_cost = 5
output=zeros(size(slope_set))

for slope = 1:length(slope_set)
        output[slope] = (marginal_cost - intercept)/slope_set[slope]
end
```

The for loop allows us to loop through all of the elements in the slope set, plug each element into the optimal quantity formula, and then store the answer.

2.4 Functions

Programming functions are just like mathematical functions: they take inputs and give you outputs after applying operations to the inputs you used. Functions are what allow us to organize our code,

[4] This is ignoring the fact that the optimal quantity might not be an integer.

[5] In Julia you can directly loop through collections of strings or arrays of arbitrary contents. Here you could replace the first line of the loop with something like: for slope in slope_set and it will iterate through the elements of slope_set.

every time we want to perform a certain action, like maximizing a Bellman equation, we can call a function that does the maximization instead of continually re-writing the maximization problem. It also gives the maximization procedure a meaningful label if you name your functions well.

```
function solver(mc::Float64,inter::Float64,slope::Float64)
        return (mc-inter)/slope
end
```

This function named "solver" takes in three arguments: marginal cost, the intercept, and the slope, and then calculates/returns the optimum quantity as the output. After the equals sign is the function name that you will use when you call the function and then the inputs in parentheses. Notice that after the three inputs is `::Float64`. This lets the compiler know the inputs are all floats. If they aren't, Julia will throw an error because it will attempt to call arithmetic functions (methods) for floats when it might be getting ints. You do not need to put the type of the inputs in the function since it's tricky to get used to. However not doing this will slow down your code.

2.5 Commenting

Code should be commented and the comments should be informative. If you come back to code you wrote one year ago and there are no comments, it will take you much longer to understand. It will also be nearly impossible for others (me) to know what you're doing. In Julia lines are commented by beginning with a #.

2.6 Optimize

Your code is working, but you want to make it faster. How can you do so? Use the Profiler, `@profile`.[6] The profiler tells you how much time your code spends in each one of your functions, and how many times each of your functions are called. Functions that are called many times, and/or appear to be taking up a large fraction of the overall compute time are prime areas to trying to shore up inefficiencies in your code with preallocation and vectorization.

Julia is also very fast with loops because of how it compiles code and handles memory. Unlike other languages, you often want your code to not be vectorized, but have loops. A cheap way to devectorize code is the devectorize package and macro `@devec`.

[6] The ProfileView package is very nice for looking at where your code is slow due to type instability.