Julia Programming for Operations Research

A Primer on Computing

Changhyun Kwon

Julia Programming for Operations Research: A Primer on Computing

http://www.chkwon.net/julia

Second Printing with Updates: November 2016

Published by Changhyun Kwon
Cover Design by Joo Yeon Woo / www.spacekite.net

ISBN: 153332879X (Paperback)
ISBN-13: 978-1533328793 (Paperback)

Contents

Preface

The main motivation of writing this book was to help myself. I am a professor in the field of operations research, and my daily activities involve building models of mathematical optimization, developing algorithms for solving the problems, implementing those algorithms using computer programming languages, experimenting with data, etc. Three languages are involved: human language, mathematical language, and computer language. My students and I need to go over three different languages. We need "translation" among the three languages.

When my students seek help on the tasks of "translation," I often provide them with my prior translation as an example, or find online resources that may be helpful to them. If students have proper background with proper mathematical education, sufficient computer programming experience, and good understanding of how numerical computing works, students can learn easier and my daily tasks in research and education would go smoothly.

To my frustration, however, many graduate students in operations research take long time to learn how to "translate." This book is to help them and help me to help them.

I'm neither a computer scientist nor a software engineer. Therefore, this book does not teach the best translation. Instead, I'll try to teach how one can finish some common tasks necessary in research and development works arising in the field of operations research and management science. It will be just one translation, not

the best for sure. But after reading this book, readers will certainly be able to get things done, one way or the other.

What this book teaches

This book is *neither* a textbook in numerical methods, a comprehensive introductory book to Julia programming, a textbook on numerical optimization, a complete manual of optimization solvers, *nor* an introductory book to computational science and engineering—it is a little bit of all.

This book will first teach how to install the Julia Language itself. This book teaches a little bit of syntax and standard libraries of Julia, a little bit of programming skills using Julia, a little bit of numerical methods, a little bit of optimization modeling, a little bit of Monte Carlo methods, a little bit of algorithms, and a little bit of optimization solvers.

This book by no means is complete and cannot serve as a standalone textbook for any of the above mentioned topics. In my opinion, it is best to use this book along with other major textbooks or reference books in operations research and management science. This book assumes that readers are already familiar with topics in optimization theory and algorithms, or are willing to learn by themselves from other references. Of course, I provide the best references of my knowledge to each topic.

After reading this book and some coding exercises, readers should be able to search and read many other technical documents available online. This book will just help the first step to computing in operations research and management science. This book is literally a primer on computing.

How this book can be used

This book will certainly help graduate students (and their advisors) for tasks in their research. First year graduate students may use this book as a *tutorial* that guides them to various optimization solvers and algorithms available. This book will also be a *companion* through their graduate study. While students take various courses during their graduate study, this book will be always a good starting point to learn how to solve certain optimization problems and implement algorithms they learned. Eventually, this book can be a helpful *reference* for their thesis research.

Advanced graduate students may use this book as a *reference*. For example, when they need to implement a Lagrangian relaxation method for their own problem, they can refer to a chapter in this book to see how I did it and learn how they may be able to do it.

It is also my hope that this book can be used for courses in operations research, analytics, linear programming, nonlinear programming, numerical optimization, network optimization, management science, and transportation engineering, as a *supplementary textbook*. If there is a short course with 1 or 2 credit hours for teaching numerical methods and computing tools in operations research and management science, this book can be *primary or secondary textbook*, depending on the instructor's main focus.

Notes to advanced programmers

If you are already familiar with computing and at least one computer programming language, I don't think this book will have much value for you. There are many resources available on the web, and you will be able to learn about the Julia Language and catch up with the state-of-the-art easily. If you want to learn and catch up even faster with much less troubles, this book can be helpful.

I had some experiences with MATLAB and Java before learning Julia. Learning Julia was not very difficult, but exciting and fun. I just needed to good "excuse" to learn and use Julia. Check what my excuse was in the first chapter.

Acknowledgment

I sincerely appreciate all the efforts from Julia developers. The Julia Language is a beautiful language that I love very much. It changed my daily computing life completely. I am thankful to the developers of the JuMP package, especially the three main developers: Iain Dunning, Joey Huchette, and Miles Lubin. After JuMP, I no longer look for better modeling languages. I am grateful to Joo Yeon Woo for designing the book cover. The cat is modified from the design of Freepik.

Tampa, Florida
Changhyun Kwon

1

Introduction and Installation

This chapter will introduce what the Julia Language is and explain why I love it. More importantly, this chapter will teach you how to obtain Julia and install it in your machine. Well, at this moment, the most challenging task for using Julia in computing would probably be installing the language and other libraries and programs correctly in your own machine. I will go over every step with fine details with screenshots for both Windows and Mac machines. I assumed that Linux users can handle the installation process well enough without much help from this book by reading online manuals and googling. Perhaps the Mac section could be useful to Linux users.

All Julia codes in this book are shared as a git repository and are available at the book website: http://www.chkwon.net/julia. Codes are tested with

- `Julia v0.5.0`
- `JuMP v0.14.1`
- `Optim v0.6.1`
- `MathProgBase v0.5.7`

I will introduce what JuMP, Optim, and MathProgBase are gradually later in the book.

1.1 What is Julia and Why Julia?

The Julia Language is a very young language. As of May 9, 2016, the latest stable version is 0.4.5. The primary target of Julia is technical computing. It is developed for making technical computing more fun and more efficient. There are many good things about the Julia Language from the perspective of computer scientists and software engineers; you can read about the language at the official website[1].

Here is a quote from the creators of Julia from their first official blog article "Why We Created Julia"[2]:

> "We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.
>
> (Did we mention it should be as fast as C?)"

So this is how Julia is created, to serve all above greedy wishes.

Let me tell you my story. I used to be a Java developer for a few years before I joined a graduate school. My first computer codes for homework assignments and course projects were naturally written in Java; even before then, I used C for my homework assignments for computing when I was an undergraduate student. Later, in the graduate school, I started using MATLAB, mainly because my fellow graduate students in the lab were using MATLAB. I needed to learn from them, so I used MATLAB.

I liked MATLAB. Unlike in Java and C, I don't need to declare every single variable before I use it; I just use it in MATLAB. Arrays are not just arrays in the computer memory; arrays in MATLAB are just like vectors and matrices. Plotting computation results is easy. For modeling optimization problems, I used GAMS

[1] http://julialang.org
[2] http://julialang.org/blog/2012/02/why-we-created-julia

and connected with solvers like CPLEX. While the MATLAB-GAMS-CPLEX chain suited my purpose well, I wasn't that happy with the syntax of GAMS—I couldn't fully understand—and the slow speed of the interface between GAMS and MATLAB. While CPLEX provides complete connectivities with C, Java, and Python, it was very basic with MATLAB.

When I finished with my graduate degree, I seriously considered Python. It was—and still is—a very popular choice for many computational scientists. CPLEX also has a better support for Python than MATLAB. Unlike MATLAB, Python is free and open source software. However, I didn't go with Python and decided to stick with MATLAB. I personally don't like 0 being the first index of arrays in C and Java. In Python, it is also 0. In MATLAB, it is 1. For example, if we have a vector like:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ 3 \\ -1 \end{bmatrix}$$

it may be written in MATLAB as:

```
v = [1; 0; 3; -1]
```

The first element of this vector should be accessible by `v(1)`, not `v(0)`. The i-th element must be `v(i)`, not `v(i-1)`. So I stayed with MATLAB.

Later in 2012, the Julia Language was introduced and it looked attractive to me, since at least the array index begins with 1. After some investigations, I didn't move to Julia at that time. It was ugly in supporting optimization modeling and solvers. I kept using MATLAB.

In 2014, I came across several blog articles and tweets talking about Julia again. I gave it one more look. Then I found a package for modeling optimization problems in Julia, called JuMP—Julia for Mathematical Programming. After spending a few hours, I felt in love with JuMP and decided to go with Julia, well more with JuMP. Here is a part of my code for solving a network optimization problem:

```
m = Model()
@variable(m, 0<= x[links] <=1)
```

```
@objective(m, Min, sum(c[(i,j)] * x[(i,j)] for (i,j) in links) )

for i=1:no_node
  @constraint(m, sum(x[(ii,j)] for (ii,j) in links if ii==i )
                - sum(x[(j,ii)] for (j,ii) in links if ii==i ) == b[i])
end

solve(m)
```

This is indeed a direct "translation" of the following mathematical language:

$$\min \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

subject to

$$\sum_{(i,j)\in\mathcal{A}} x_{ij} - \sum_{(j,i)\in\mathcal{A}} x_{ji} = b_i \quad \forall i \in \mathcal{N}$$
$$0 \le x_{ij} \le 1 \quad \forall (i,j) \in \mathcal{A}$$

I think it is a very obvious translation. It is quite beautiful, isn't it?

CPLEX and its competitor Gurobi are also very smoothly connected with Julia via JuMP. Why should I hesitate? After two years of using Julia, I still love it—I even wrote a book, which you are reading now, about Julia!

1.2 Julia in the Cloud: JuliaBox

You can enjoy many features of the Julia Language on the web at http://juliabox.com. Log in with your Google account and create a "New Notebook".

First, install the `Clp` and `JuMP` packages.

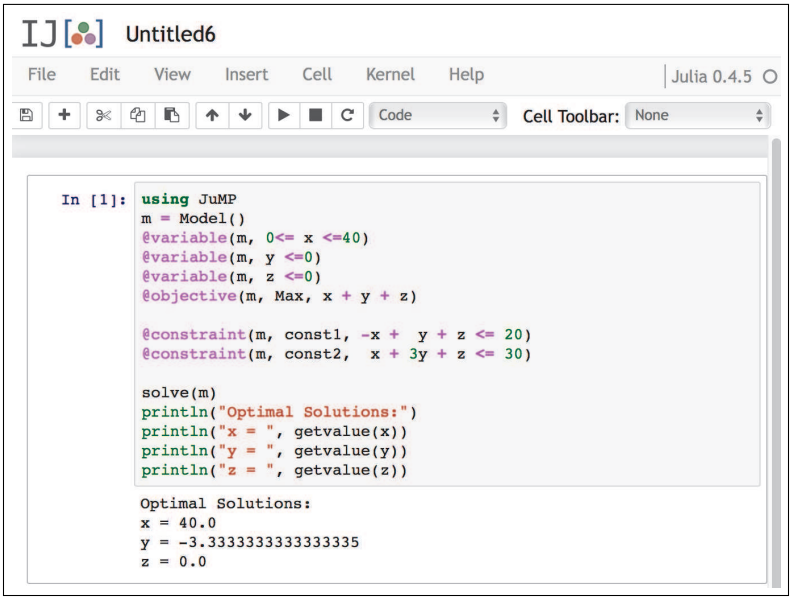
```
Pkg.add("Clp")
Pkg.add("JuMP")
```

and press `Shift+Enter` or click the "play" button to run your code. Clp provides an open source LP solver, and JuMP provides a nice modeling interface.

Copy this code to your screen:

```
using JuMP
m = Model()
@variable(m, 0<= x <=40)
@variable(m, y <=0)
@variable(m, z <=0)
@objective(m, Max, x + y + z)

@constraint(m, const1, -x +  y + z <= 20)
@constraint(m, const2,  x + 3y + z <= 30)

solve(m)
println("Optimal Solutions:")
println("x = ", getvalue(x))
println("y = ", getvalue(y))
println("z = ", getvalue(z))
```

and press **Shift+Enter** or click the "play" button to run your code. The result will look like:



If you want to use commercial solvers CPLEX or Gurobi, you have to install Julia in your computer. Please follow the instruction in the next section.

1.3 Installing Julia

Graduate students and researchers are strongly recommended to install Julia in their local computers. In this guide, we will first install the Gurobi optimizer and then Julia. Gurobi is a commercial optimization solver package for solving LP, MILP, QP, MIQP, etc, and it is free for students, teachers, professors, or anyone else related to educational organizations.

If you are not eligible for free licenses for Gurobi, please go ahead and install Julia. There are open-source solvers available. Windows users go to Section 1.3.2, and Mac users go to Section 1.3.3.

1.3.1 Installing Gurobi

First, install Gurobi Optimizer.

1. Download Gurobi Optimizer[3] and install in your computer. (You will need to register as an academic user, or purchase a license.)

2. Request a free academic license[4] and follow their instruction to activate it.

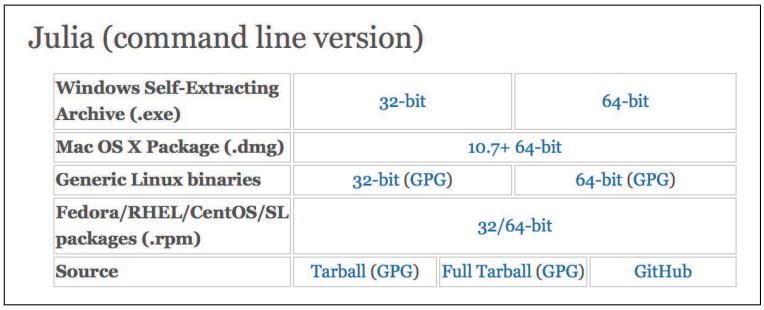
(Note to **Windows** users: The version you select, either 32-bit or 64-bit, needs to be consistent. That is, if you choose 64-bit Gurobi Optimizer, you will need to install 64-bit Julia in the next step. After installation, you *must reboot* your computer.)

The following two sections provide steps with screenshots to install the Julia Language, the JuMP package, and the Gurobi package. Windows users go to Section 1.3.2, and Mac users go to Section 1.3.3.

1.3.2 Installing Julia in Windows

- **Step 1.** Download Julia from the official website.[5] (Select an appropriate version: 32-bit or 64-bit, same as your Gurobi Optimizer version.)

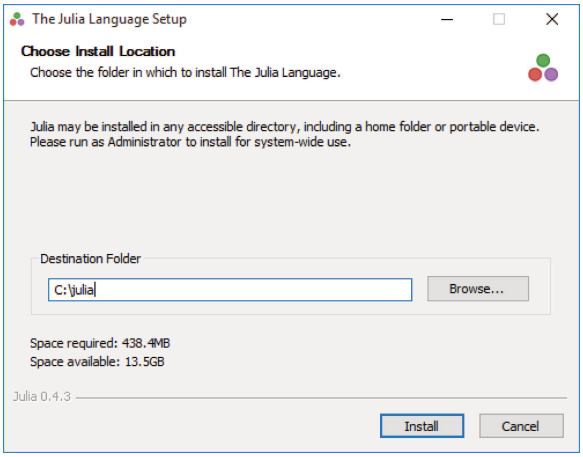
[3] http://user.gurobi.com/download/gurobi-optimizer
[4] http://user.gurobi.com/download/licenses/free-academic
[5] http://julialang.org/downloads/



Julia (command line version)

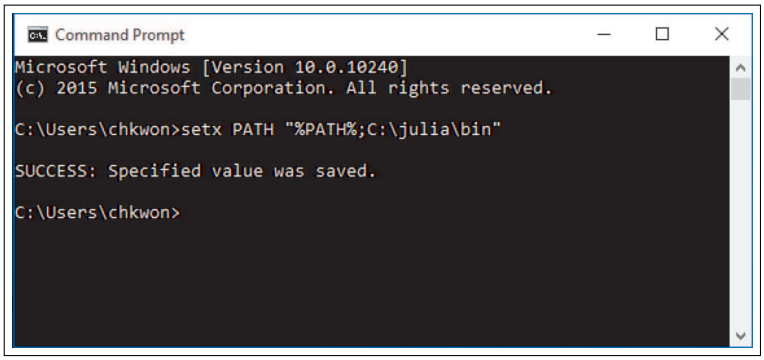
Windows Self-Extracting Archive (.exe)	32-bit	64-bit	
Mac OS X Package (.dmg)	10.7+ 64-bit		
Generic Linux binaries	32-bit (GPG)	64-bit (GPG)	
Fedora/RHEL/CentOS/SL packages (.rpm)	32/64-bit		
Source	Tarball (GPG)	Full Tarball (GPG)	GitHub

- **Step 2.** Install Julia in `C:\julia`. (You need to make the installation folder consistent with the path you set in Step 3.)



- **Step 3.** Open a Command Prompt and enter the following command:

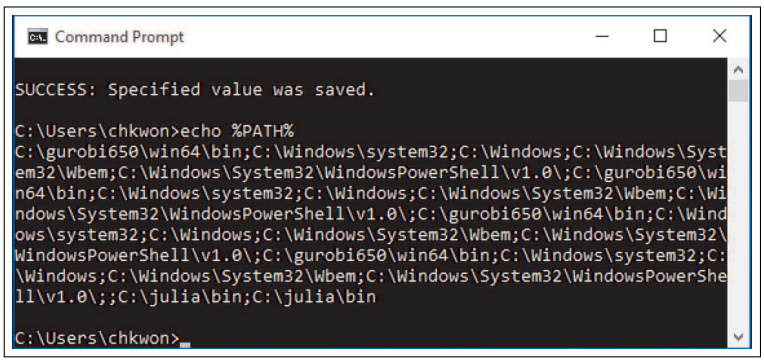
```
setx PATH "%PATH%;C:\julia\bin"
```



If you dont know how to open a Command Prompt, see this link[6] (choose your Windows version, and see "How do I get a command prompt").

- **Step 4.** Open a **NEW** command prompt and type

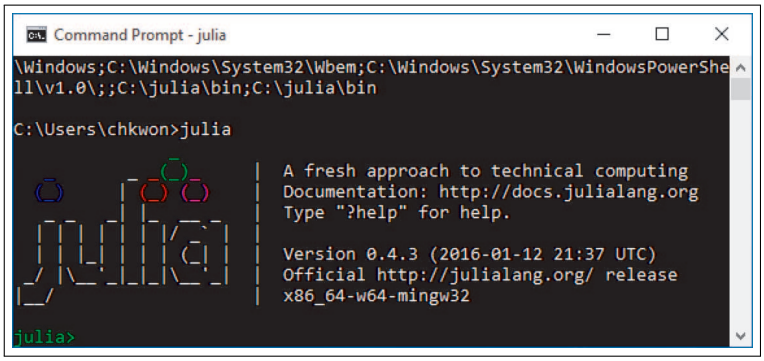
```
echo %PATH%
```



The output must include `C:\julia\bin` in the end. If not, you must have something wrong.

- **Step 5.** Run `julia`.

[6] http://windows.microsoft.com/en-us/windows/command-prompt-faq

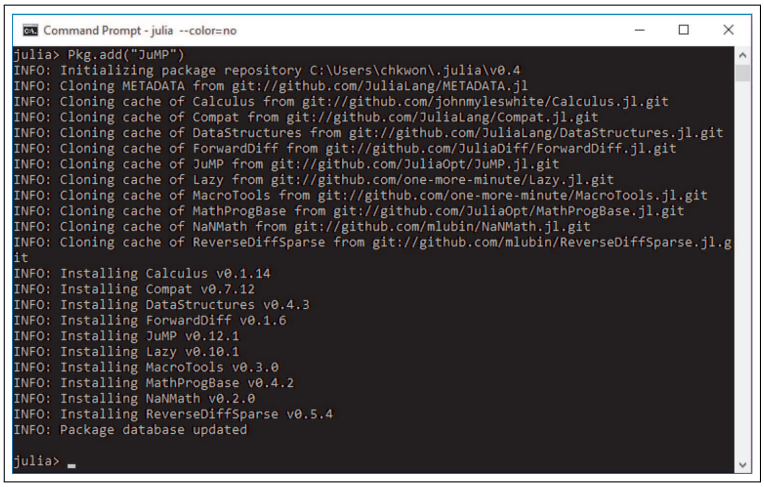


You have successfully installed the Julia Language on your Windows computer. Now its time for installing additional packages for mathematical programming.

- **Step 6.** If you have not installed Gurobi yet in your system, please install it first. On your julia prompt, type

```
Pkg.add("JuMP")
Pkg.add("Gurobi")
```

(If you are ineligible to use a free license of Gurobi, use the `Cbc` solver. You can install it by entering the following command: `Pkg.add("Cbc")`)



- **Step 7.** Open Notepad (or any other text editor, for example Atom[7]) and type the following, and save the file as `script.jl` in some folder of your choice.

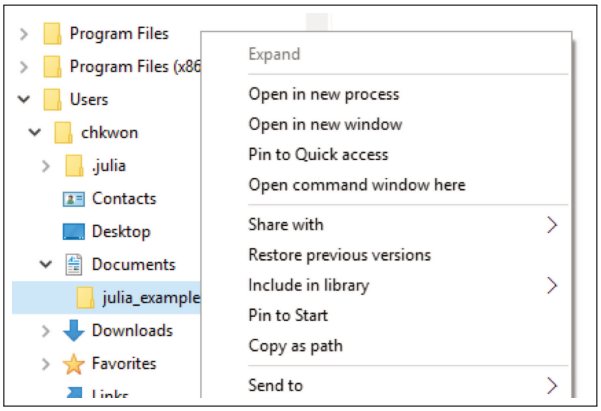
```
using JuMP, Gurobi
m = Model(solver=GurobiSolver())

@variable(m, 0 <= x <= 2 )
@variable(m, 0 <= y <= 30 )

@objective(m, Max, 5x + 3*y )

@constraint(m, 1x + 5y <= 3.0 )

print(m)

status = solve(m)

println("Objective value: ", getobjectivevalue(m))
println("x = ", getvalue(x))
println("y = ", getvalue(y))
```

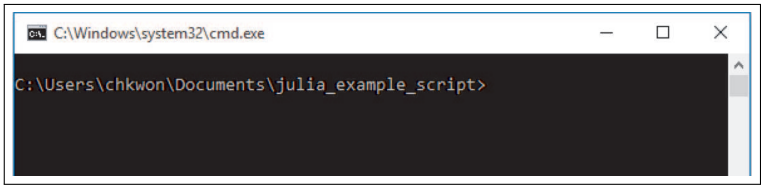
If you are ineligible to use a free license of Gurobi, replace the first two lines by

```
using JuMP, Cbc
m = Model(solver=CbcSolver())
```

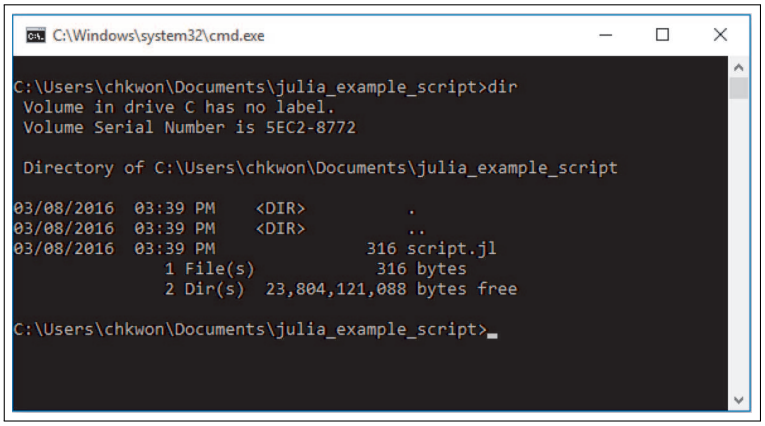
- **Step 8.** Press and hold your `Shift` Key and right-click the folder name, and choose "Open command window here."

[7] http://atom.io





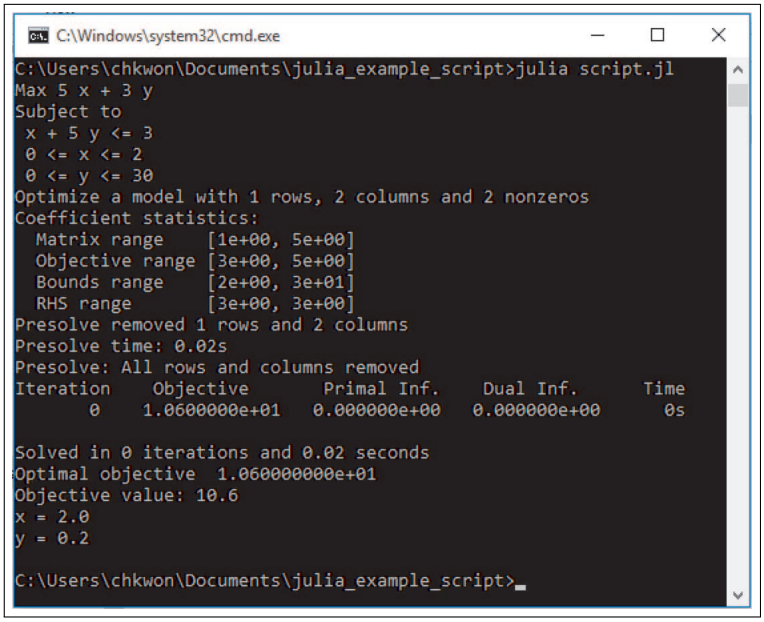
- **Step 9.** Type `dir` to see your script file `script.jl`.



If you see a filename such as `script.jl.txt`, use the following command to rename:

```
ren script.jl.txt script.jl
```

- **Step 10.** Type `julia script.jl` to run your julia script.



After a few seconds, the result of your julia script will be printed. Done.

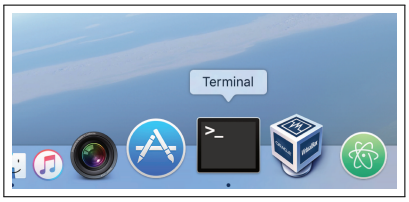
Please proceed to Section 1.3.4.

1.3.3 Installing Julia in macOS

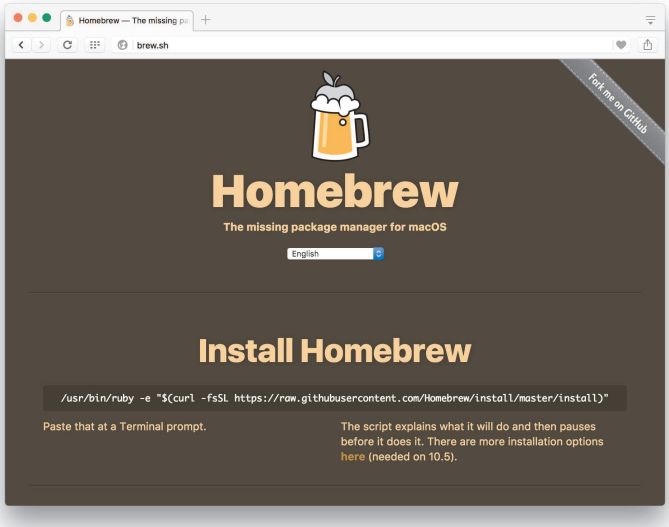
In macOS, we will use a package manager, called Homebrew. It provides a very convenient way of installing software in macOS.

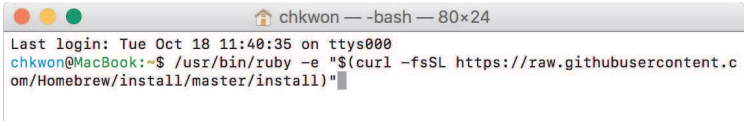
- **Step 1.** Open "Terminal.app" from your Applications folder. (If you dont know how to open it, see this video.[8] It is convenience to place "Terminal.app" in your dock.

[8] https://www.youtube.com/watch?v=zw7Nd67_aFw "How to open the terminal window on a Mac"



- **Step 2.** Visit http://brew.sh and follow the instruction to install Homebrew.





- **Step 3.** Installing Julia using Homebrew: In your terminal, enter the following command:

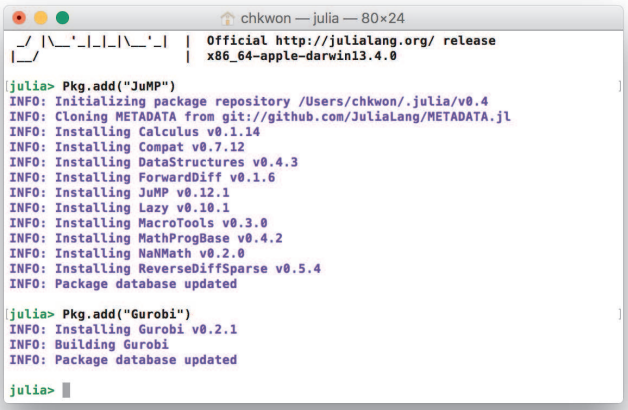
```
brew cask install julia
```

- **Step 5.** In your terminal, enter `julia`.

- **Step 6.** If you have not installed Gurobi in your system yet, go back and install it first. Then, on your Julia prompt, type

```
Pkg.add("JuMP")
Pkg.add("Gurobi")
```

(If you are ineligible to use a free license of Gurobi, use the `Cbc` solver. You can install it by entering the following command: `Pkg.add("Cbc")`)



- **Step 7.** Open TextEdit (or any other text editor, for example Atom[9]) and type the following, and save the file as `script.jl` in some folder of your choice.

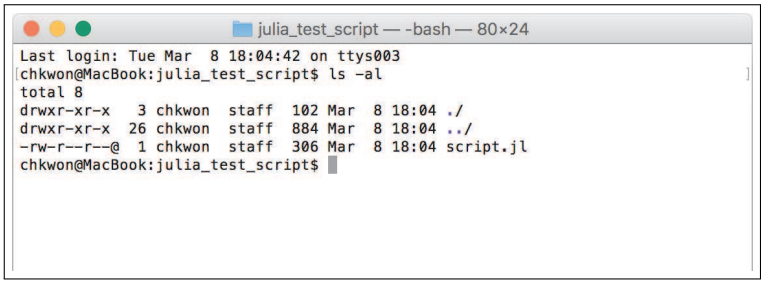
```
using JuMP, Gurobi
m = Model(solver=GurobiSolver())

@variable(m, 0 <= x <= 2 )
@variable(m, 0 <= y <= 30 )

@objective(m, Max, 5x + 3*y )

@constraint(m, 1x + 5y <= 3.0 )

print(m)
status = solve(m)

println("Objective value: ", getobjectivevalue(m))
println("x = ", getvalue(x))
println("y = ", getvalue(y))
```

If you are ineligible to use a free license of Gurobi, replace the first two lines by

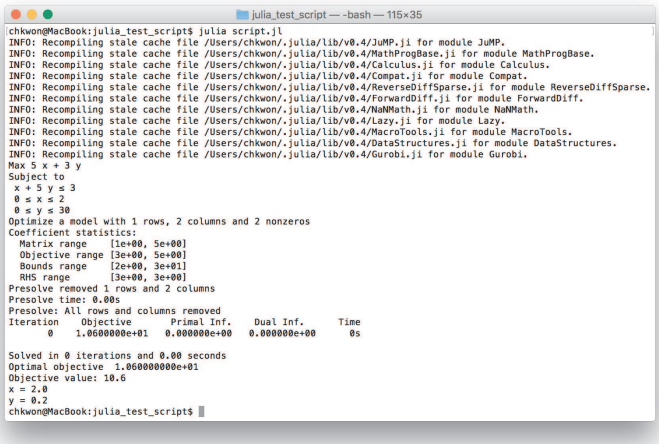
[9] http://atom.io

```
using JuMP, Cbc
m = Model(solver=CbcSolver())
```

- **Step 8.** Open a terminal window[10] at the folder that contains your `script.jl`.
- **Step 9.** Type `ls -al` to check your script file.



- **Step 10.** Type `julia script.jl` to run your script.



[10] To do this, you can drag the folder to the Terminal.app icon in your dock, or see http://osxdaily.com/2011/12/07/open-a-selected-finder-folder-in-a-new-terminal-window/

After a few seconds, the result of your julia script will be printed. Done.

Please proceed to Section 1.3.4.

1.3.4 Running Julia Scripts

When you are ready, there are basically two methods to run your Julia script:

- In your Command Prompt or Terminal, enter `C:> julia your-script.jl`
- In your Julia prompt, enter `julia> include("your-script.jl")`.

1.3.5 Installing CPLEX

Instead of Gurobi, you can install and connect the CPLEX solver, which is also free to academics. Installing CPLEX is a little more complicated task.

CPLEX in Windows

You can follow this step by step guide to install:

1. Log in at the academic initiative page[11] at the IBM website.
2. Follow the instructions on the page.
3. Download an appropriate version to your system:
   - `cplex_studio126.win-x86-32.exe` for 32-bit systems
   - `cplex_studio126.winx8664.exe` for 64-bit systems.
4. Reboot.
5. Run the downloaded exe file You may need to rightclick the exe file and "Run as Administrator."
6. Run Julia and add the `CPLEX` package:

[11] https://www-304.ibm.com/ibm/university/academic/pub/jsps/assetredirector.jsp?asset_id=1070

```
julia> Pkg.add("CPLEX")
```

7. Ready. Test the following code:

```
using JuMP, CPLEX
m = Model(solver=CplexSolver())
@variable(m, x <= 5)
@variable(m, y <= 45)
@objective(m, Min, x + y)
@constraint(m, 50x + 24y <= 2400)
@constraint(m, 30x + 33y <= 2100)

status = solve(m)
println("Optimal objective: ",getobjectivevalue(m))
println("x = ", getvalue(x), " y = ", getvalue(y))
```

CPLEX in macOS

The instruction includes how to deal with .bin file on macOS:

1. Log in at the academic initiative page[12] at the IBM website.
2. Follow the instructions on the page.
3. Download an appropriate version to your system: `cplex_studio126.osx.bin`.
4. Place the file in your home directory: `/Users/[Your User Name]`. Copying and pasting from the Download directory should work here.
5. To install, open Terminal.
6. At the prompt, type in

   ```
   /bin/bash ~/cplex_studio126.osx.bin
   ```

   and hit enter. Follow all the prompts.

[12] https://www-304.ibm.com/ibm/university/academic/pub/jsps/assetredirector.jsp?asset_id=1070

To add the `CPLEX` package to Julia, follow:

1. Open your `~/.bash_profile` file to edit:

```
open -e ~/.bash_profile
```

2. Add the following line to your `~/.bash_profile` file: (change [USER NAME])

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:"/Users/[USER NAME]/Applications/
  IBM/ILOG/CPLEX_Studio126/cplex/bin/x86-64_osx/"
```

   Note that the above code needs to be a *single* line.

3. Reload your profile:

```
source ~/.bash_profile
```

4. Run Julia and add the `CPLEX` package:

```
julia> Pkg.add("CPLEX")
```

5. Ready. Test the following code:

```
using JuMP, CPLEX
m = Model(solver=CplexSolver())
@variable(m, x <= 5)
@variable(m, y <= 45)
@objective(m, Min, x + y)
@constraint(m, 50x + 24y <= 2400)
@constraint(m, 30x + 33y <= 2100)

status = solve(m)
println("Optimal objective: ",getobjectivevalue(m))
println("x = ", getvalue(x), " y = ", getvalue(y))
```

1.4 Installing IJulia

As you have seen in Section 1.2, JuliaBox provides a nice interactive programming environment. You can also use such an interactive environment in your local computer. JuliaBox is based on Jupyter Notebook[13]. Well, at first there was `IPython` notebook that was an interactive programming environment for the Python language. It has been popular, and now it is extended to cover many other languages such as R, Julia, Ruby, etc. The extension became the Jupyter Notebook project. For Julia, it is called `IJulia`, following the naming convention of `IPython`.

To use `IJulia`, we need a distribution of Python and Jupyter. Julia can automatically install a distribution for you, unless you want to install it by yourself. If you let Julia install Python and Jupyter, they will be private to Julia, i.e. you will not be able to use Python and Jupyter outside of Julia. If you decided to install Python/Jupyter by yourself, I recommend you to install Anaconda with Python 2.7 version and consult with the documentation of `IJulia`[14].

The following process will automatically install Python and Jupyter.

1. Open a new terminal window and run Julia. Initialize environment variables:

```
julia> ENV["PYTHON"] = ""
""

julia> ENV["JUPYTER"] = ""
""
```

2. Install `IJulia`:

```
julia> Pkg.add("IJulia")
```

3. To open the `IJulia` notebook in your web browser:

```
julia> using IJulia
julia> notebook()
```

[13] http://jupyter.org
[14] https://github.com/JuliaLang/IJulia.jl

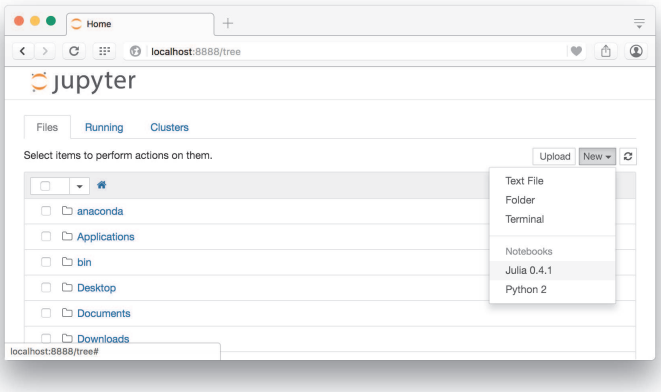
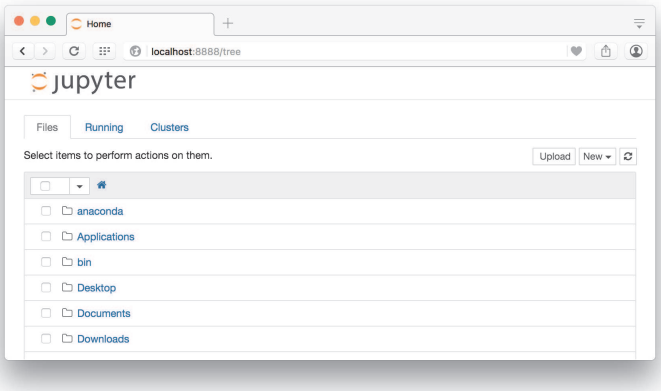


Figure 1.1: Creating a new notebook

It will open a webpage in your browser that looks like the following screenshot:



The current folder will be your home folder. You can move to another folder and also create a new folder by clicking the "New" button on the top-right corner of the screen. After locating a folder you want, you can now create a new `IJulia` notebook by clicking the "New" button again and select the julia version of yours, for example "Julia 0.4.1". See Figure 1.1.

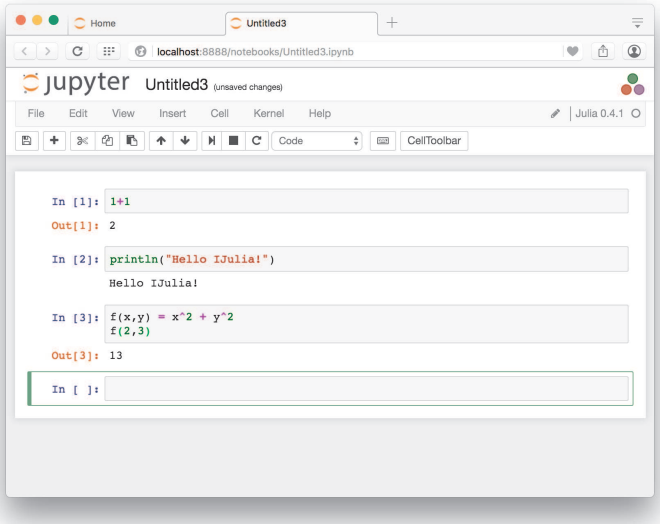


Figure 1.2: Some basic Julia codes.

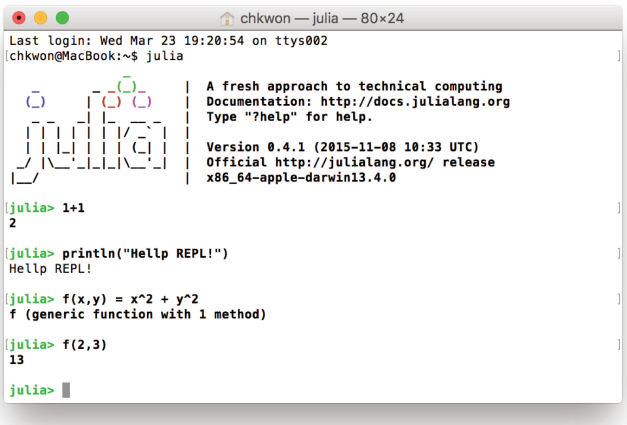


Figure 1.3: This is the REPL.

It will basically open an interactive session of the Julia Language. If you have used Mathematica or Maple, the interface will look familiar. You can test basic Julia commands. When you need to evaluate a block of codes, press `Shift`+`Enter`, or press the "play" button. See Figure 1.2.

If you properly install a plotting package like `PyPlot` (details in Section 3.10.1), you can also do plotting directly within the `IJulia` notebook as shown in Figure 1.4.

Personally, I prefer the REPL for most tasks, but I occasionally use `IJulia`, especially when I need to test some simple things and need to plot the result quickly, or when I need to share the result of Julia computation with someone else. (`IJulia` can export the notebook in various formats, including HTML and PDF.)

What is the REPL? It stands for read-eval-print loop. It is the Julia session that runs in your terminal; see Figure 1.3, which must look familiar to you already.

1.5 Package Management

There are many useful packages in Julia and we rely many parts of our computations on packages. If you have followed my instructions to install Julia, JuMP, Gurobi, and

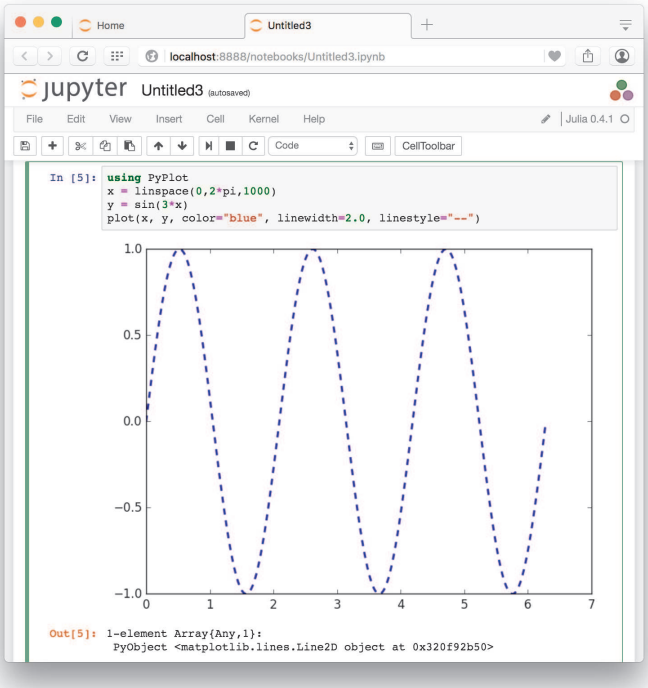


Figure 1.4: Plotting in IJulia

CPLEX, you have already installed a few packages. There are some more commands that are useful in managing packages.

```
julia> Pkg.add("PackageName")
```

This installs a package, named `PackageName`. To find its online repository, you can just google the name `PackageName.jl`, and you will be directed to a repository hosted at `GitHub.com`.

```
julia> Pkg.rm("PackageName")
```

This removes the package.

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

This updates all packages that are already installed in your machine to the most recent versions.

```
julia> Pkg.status()
```

This displays what packages are installed and what their versions are. If you just want to know the version of a specific package, you can do:

```
julia> Pkg.status("PackageName")
```

```
julia> Pkg.build("PackageName")
```

Occassionally, installing a package will fail during the `Pkg.add("PackageName")` process, usually because some libraries are not installed or system path variables are not configured correctly. Try to install some required libraries again and check the system path variables first. Then you may need to reboot your system or restart your Julia session. Then `Pkg.build("PackageName")`. Since you have downloaded package files during `Pkg.build("PackageName")`, you don't need to download them again; you just build it again.

1.6 Helpful Resources

Readers can find codes and other helpful resources in the author's website at

http://www.chkwon.net/julia

which also includes a link to a Facebook page of this book for discussion and communication.

This book does *not* teach everything of the Julia Language—only a very small part of it. When you want to learn more about the language, the first place you need to visit is

http://julialang.org/learning/

where many helpful books, tutorials, videos, and articles are listed. Also, you will need to visit the official documentation of the Julia Language at

http://docs.julialang.org/

which I think serves as a good tutorial as well.

When you have a question, there will be many Julia enthusiasts ready for you. For questions and discussion, visit

https://discourse.julialang.org

and

http://julialang.org/community/

You can also ask questions at http://stackoverflow.com with tag `julia-lang`.

The webpage of JuliaOpt is worth visiting. JuliaOpt is a group of people who develop and use optimization related packages in the Julia Language. The website provides a nice overview of the available packages and well-tailored examples. The website is

http://www.juliaopt.org