Julia Programming for Operations Research 2/e

Changhyun Kwon

Copyrighted material

Julia Programming for Operations Research

https://www.chkwon.net/julia

Second Edition

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

Copyright © 2019 by Changhyun Kwon
All Rights Reserved.

version 2021/03/06 13:58:00

Copyrighted material

Contents

1 Introduction and Installation **1**

1.1 What is Julia and Why Julia? 2

1.2 Installing Julia 4

1.2.1 Installing Julia in Windows 4

1.2.2 Installing Julia in macOS 10

1.2.3 Running Julia Scripts 15

1.2.4 Installing Gurobi 15

1.2.5 Installing CPLEX 16

1.3 Installing IJulia 18

1.4 Package Management 21

1.5 Help 25

2 Simple Linear Optimization **29**

2.1 Linear Programming (LP) Problems 30

2.2 Alternative Ways of Writing LP Problems 34

2.3 Yet Another Way of Writing LP Problems 37

2.4 Mixed Integer Linear Programming (MILP) Problems 38

Copyrighted material

3 Basics of the Julia Language 41
3.1 Vector, Matrix, and Array 41
3.2 Tuple 47
3.3 Indices and Ranges 48
3.4 Printing Messages 51
3.5 Collection, Dictionary, and For-Loop 54
3.6 Function 57
3.7 Scope of Variables 59
3.8 Random Number Generation 63
3.9 File Input/Output 67
3.10 Plotting 72
3.10.1 The PyPlot Package 72
3.10.2 Avoiding Type-3 Fonts in PyPlot 77

4 Selected Topics in Numerical Methods 79
4.1 Curve Fitting 79
4.2 Numerical Differentiation 84
4.3 Numerical Integration 87
4.4 Automatic Differentiation 91

5 The Simplex Method 95
5.1 A Brief Description of the Simplex Method 95
5.2 Searching All Basic Feasible Solutions 98
5.3 Using the JuMP Package 104
5.4 Pivoting in Tableau Form 105
5.5 Implementing the Simplex Method 107
5.5.1 initialize(c, A, b) 109
5.5.2 is_optimal(tableau) 111
5.5.3 pivoting!(tableau) 112
5.5.4 Creating a Module 116
5.6 Next Steps 122

6 Network Optimization Problems 123
6.1 The Minimal-Cost Network-Flow Problem 123
6.2 The Transportation Problem 133
6.3 The Shortest Path Problem 139

Copyrighted material

6.4 Implementing Dijkstra's Algorithm 144

7 Interior Point Methods **151**
7.1 The Affine Scaling Algorithm 151
7.2 The Primal Path Following Algorithm 157
7.3 Remarks 162

8 Nonlinear Optimization Problems **165**
8.1 Unconstrained Optimization 165
8.1.1 Line Search 165
8.1.2 Unconstrained Optimization 167
8.1.3 Box-constrained Optimization 168
8.2 Nonlinear Optimization 169
8.3 Other Solvers 170
8.4 Mixed Integer Nonlinear Programming 175

9 Monte Carlo Methods **177**
9.1 Probability Distributions 177
9.2 Randomized Linear Program 179
9.3 Estimating the Number of Simple Paths 186

10 Lagrangian Relaxation **197**
10.1 Introduction 197
10.1.1 Lower and Upper Bounds 198
10.1.2 Subgradient Optimization 200
10.1.3 Summary 200
10.2 The p-Median Problem 201
10.2.1 Reading the Data File 202
10.2.2 Solving the p-Median Problem Optimally 204
10.2.3 Lagrangian Relaxation 205
10.2.4 Finding Lower Bounds 206
10.2.5 Finding Upper Bounds 210
10.2.6 Updating the Lagrangian Multiplier 212

Copyrighted material

11 Complementarity Problems 225
11.1 Linear Complementarity Problems (LCP) 225
11.2 Nonlinear Complementarity Problems (NCP) 233
11.3 Mixed Complementarity Problems (MCP) 237

12 Parameters in Optimization Solvers 239
12.1 Setting CPU Time Limit 239
12.2 Setting the Optimality Gap Tolerance 240
12.3 Warmstart 241
12.4 Big-M and Integrality Tolerance 242
12.5 Turning off the Solver Output 244
12.6 Other Solver Parameters 244

Index 247

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

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 a 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 and other related packages. After JuMP, I no longer look for better modeling languages. I am also grateful to Joo Yeon Woo for the cover design and Bomin Kwon for the cat drawing.

Tampa, Florida
Changhyun Kwon

Copyrighted material

Copyrighted material

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 v1.3.0`
- `JuMP v0.21.2`
- `Optim v0.20.6`

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

Copyrighted material

1.1 What is Julia and Why Julia?

The Julia Language is a young emerging language, whose primary target 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 was 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

Copyrighted material

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 a free and open source language. 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 still 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 fell 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:

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

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


Copyrighted material

```
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

optimize!(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

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

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 several years of using Julia, I still love it—I even wrote a book.

1.2 Installing Julia

Graduate students and researchers are strongly recommended to install Julia in their local computers. In this guide, we will first install Julia and then install two optimization packages, JuMP and GLPK. JuMP stands for 'Julia for Mathematical Programming', which is a modeling language for optimization problems. GLPK is an open-source linear optimization solver that can solve both continuous and discrete linear programs. Windows users go to Section 1.2.1, and Mac users go to Section 1.2.2.

Copyrighted material

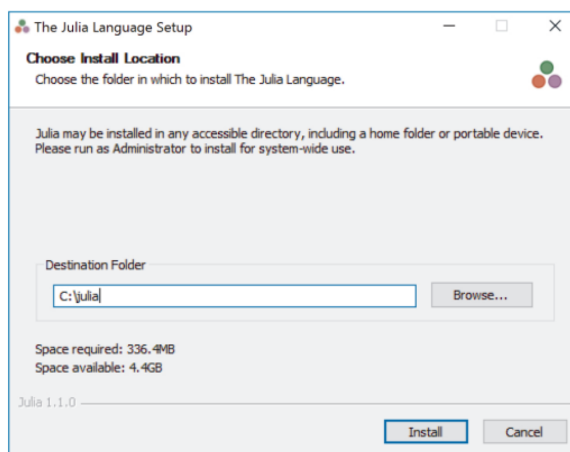
1.2.1 Installing Julia in Windows

- **Step 1.** Download Julia from the official website.[3] (Select an appropriate version: 32-bit or 64-bit. 64-bit recommended whenever possible.)

Current stable release (v1.1.0)

Windows Self-Extracting Archive (.exe) [help]	32-bit	64-bit	
	Windows 7/Windows Server 2012 users also require Windows Management Framework 3.0 or later		
macOS 10.8+ Package (.dmg) [help]		64-bit	
Generic Linux Binaries for x86 [help]	32-bit (GPG)	64-bit (GPG)	
Generic FreeBSD Binaries for x86 [help]		64-bit (GPG)	
Source	Tarball (GPG)	Tarball with dependencies (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.)

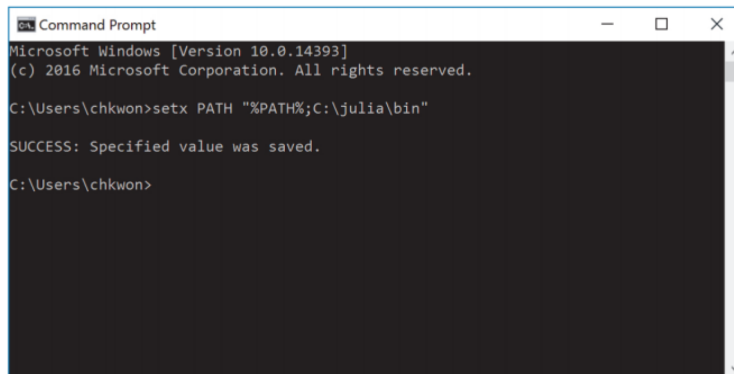


- **Step 3.** Open a Command Prompt and enter the following command:

[3] http://julialang.org/downloads/

Copyrighted material

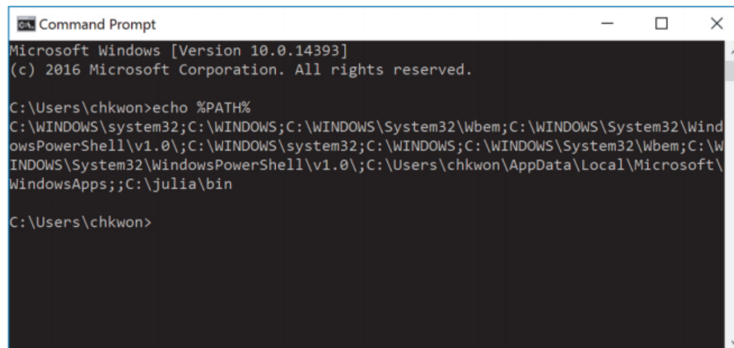
```
setx PATH "%PATH%;C:\julia\bin"
```



If you do not know how to open a Command Prompt, just google 'how to open command prompt windows.'

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

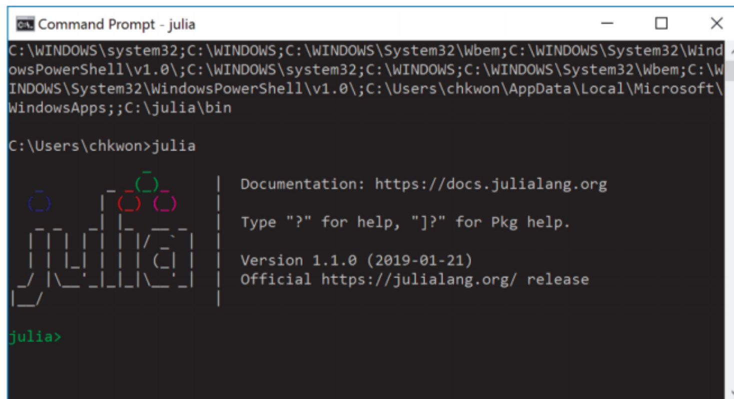
```
echo %PATH%
```



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

Copyrighted material

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

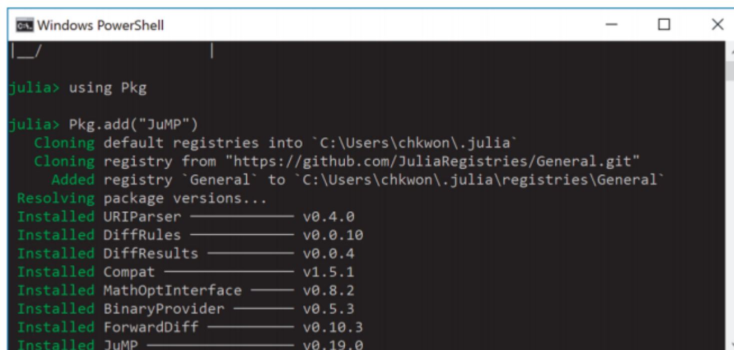


You have successfully installed the Julia Language on your Windows computer. Now it is time to install additional packages for mathematical optimization.

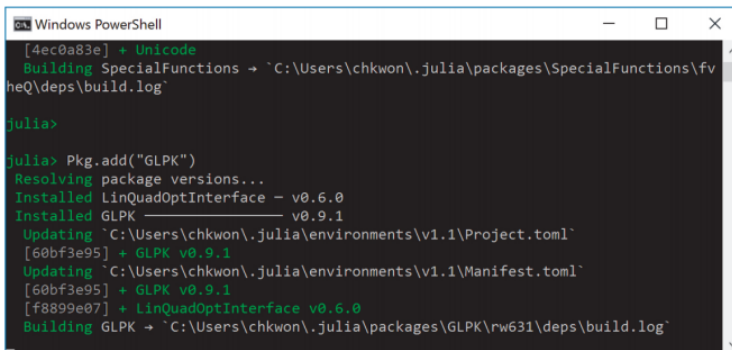
- **Step 6.** In your Julia prompt, type

```
julia> using Pkg
julia> Pkg.add("JuMP")
julia> Pkg.add("GLPK")
```

Installing the first package can take long time, because it initializes your Julia package folder and synchronizes with the entire package list.



Copyrighted material



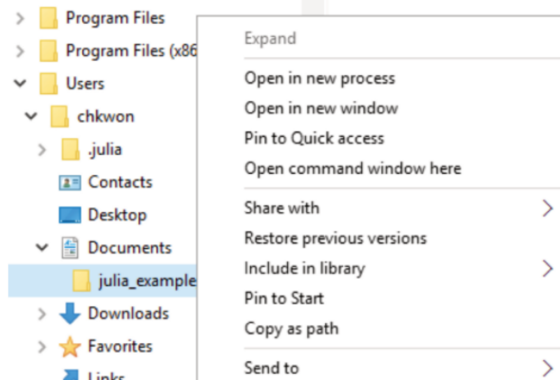
- **Step 7.** Open Notepad (or any other text editor such as Visual Studio Code[4]) and type the following, and save the file as `script.jl` in some folder of your choice.

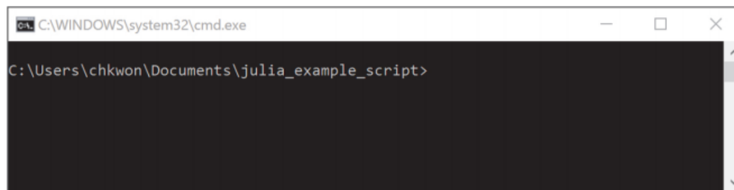
```
using JuMP, GLPK
m = Model(GLPK.Optimizer)

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

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

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

JuMP.optimize!(m)
println("Objective value: ", JuMP.objective_value(m))
println("x = ", JuMP.value(x))
println("y = ", JuMP.value(y))
```

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

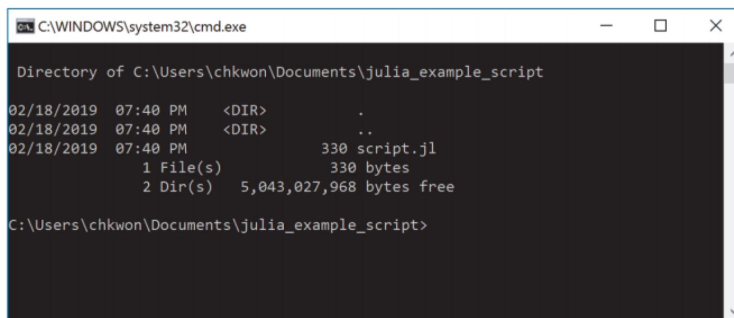
[4]https://code.visualstudio.com

Copyrighted material





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

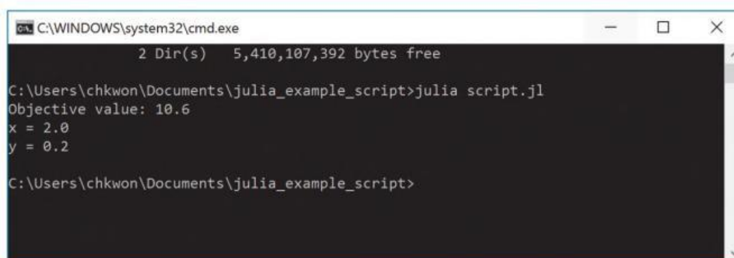


If you see a filename such as `script.jl.txt`, use the following command to rename:

Copyrighted material

```
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 1.2.3.

1.2.2 Installing Julia in macOS

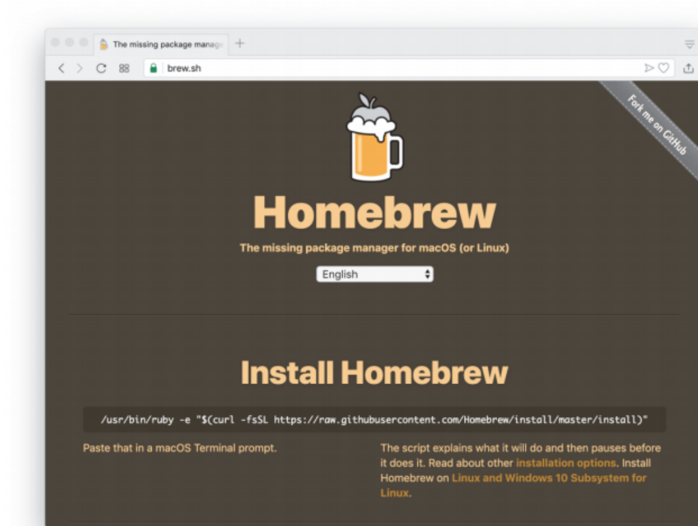
In macOS, we will use a package manager, called Homebrew. It provides a very convenient way of installing software in macOS.

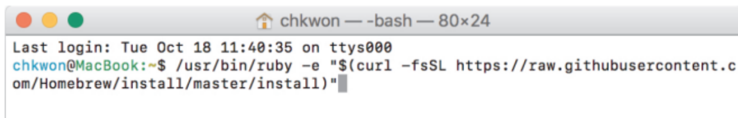
- **Step 1.** Open "Terminal.app" from your Applications folder. (If you do not know how to open it, see this video.[5] It is convenient to place "Terminal.app" in your dock.

- **Step 2.** Visit `http://brew.sh` and follow the instruction to install Homebrew. It may ask you to enter your password to install Xcode Command Line Tools.

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

Copyrighted material



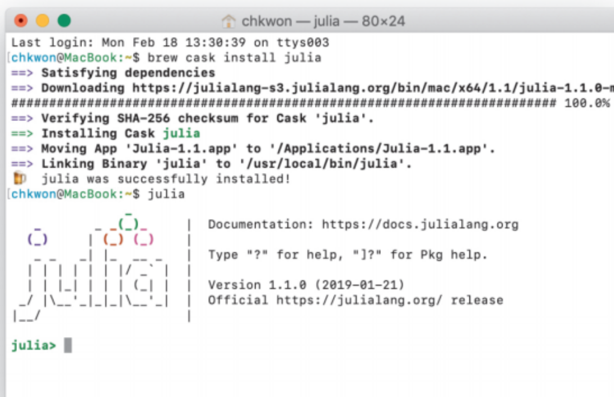


- **Step 3.** Installing Julia using Homebrew: In your terminal, enter the following command:

```
brew cask install julia
```

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

Copyrighted material



- **Step 6.** In your Julia prompt, type

```
julia> using Pkg
julia> Pkg.add("JuMP")
julia> Pkg.add("GLPK")
```

Installing the first package can take a long time, because it initializes your Julia package folder and synchronizes with the entire package list.

Copyrighted material

```
chkwon — julia — 80×24
Last login: Wed Feb 20 22:25:27 on ttys004
chkwon@MacBook:~$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.0 (2019-01-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> using Pkg

julia> Pkg.add("JuMP")
   Cloning default registries into `~/.julia`
   Cloning registry from "https://github.com/JuliaRegistries/General.git"
     Added registry `General` to `~/.julia/registries/General`
 Resolving package versions...
 Installed NaNMath ────────────── v0.3.2
 Installed DiffResults ────────── v0.0.4
 Installed CommonSubexpressions ─ v0.2.0
 Installed OrderedCollections ─── v1.0.2
 Installed BinaryProvider ─────── v0.5.3
 Installed BinDeps ────────────── v0.8.10
```

```
chkwon — julia • julia — 80×24
  [9a3f8284] + Random
  [ea8e919c] + SHA
  [9e88b42a] + Serialization
  [1a1011a3] + SharedArrays
  [6462fe0b] + Sockets
  [2f01184e] + SparseArrays
  [10745b16] + Statistics
  [8dfed614] + Test
  [cf7118a7] + UUIDs
  [4ec0a83e] + Unicode
  Building SpecialFunctions → `~/.julia/packages/SpecialFunctions/fvheQ/deps/bui
ld.log`

julia> Pkg.add("GLPK")
 Resolving package versions...
 Installed LinQuadOptInterface ─ v0.6.0
 Installed GLPK ──────────────── v0.9.1
  Updating `~/.julia/environments/v1.1/Project.toml`
  [60bf3e95] + GLPK v0.9.1
  Updating `~/.julia/environments/v1.1/Manifest.toml`
  [60bf3e95] + GLPK v0.9.1
  [f8899e07] + LinQuadOptInterface v0.6.0
  Building GLPK → `~/.julia/packages/GLPK/rw631/deps/build.log`
```

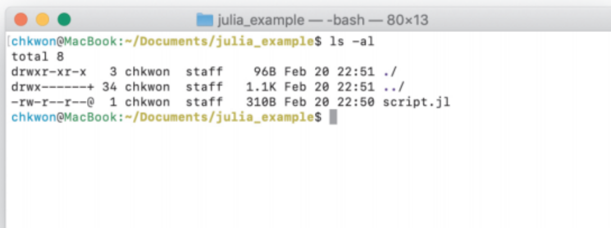
- **Step 7.** Open TextEdit (or any other text editor such as Visual Studio Code[6]) and type the following, and save the file as `script.jl` in some folder of your choice.

[6] https://code.visualstudio.com

Copyrighted material

```
using JuMP, GLPK
m = Model(GLPK.Optimizer)

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

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

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

JuMP.optimize!(m)
println("Objective value: ", JuMP.objective_value(m))
println("x = ", JuMP.value(x))
println("y = ", JuMP.value(y))
```

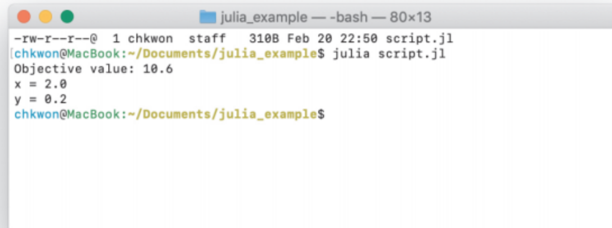
- **Step 8.** Open a terminal window[7] at the folder that contains your `script.jl`.
- **Step 9.** Type `ls -al` to check your script file.



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

[7] 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/

Copyrighted material



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

Please proceed to Section 1.2.3.

1.2.3 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")`.

1.2.4 Installing Gurobi

Instead of GLPK, one can use Gurobi, which is a commercial optimization solver package for solving LP, MILP, QP, MIQP, etc. Gurobi is free for students, teachers, professors, or anyone else related to educational organizations.

To install, follow these steps:

1. Download Gurobi Optimizer[8] and install in your computer. (You will need to register as an academic user.)
2. Request a free academic license[9] and follow their instructions to activate it.

[8] https://www.gurobi.com/downloads/gurobi-optimizer-eula/
[9] https://www.gurobi.com/academia/academic-program-and-licenses/

Copyrighted material

3. Run Julia and add the **Gurobi** package. You need to tell Julia where Gurobi is installed:

   On Windows:

```
julia> ENV["GUROBI_HOME"] =
       "C:\\Program Files\\gurobi910\\win64"
julia> using Pkg
julia> Pkg.add("Gurobi")
```

   On macOS:

```
julia> ENV["GUROBI_HOME"] =
       "/Library/gurobi910/mac64"
julia> using Pkg
julia> Pkg.add("Gurobi")
```

4. Ready. Test the following code:

```
using JuMP, Gurobi
m = Model(Gurobi.Optimizer)
@variable(m, x <= 5)
@variable(m, y <= 45)
@objective(m, Max, x + y)
@constraint(m, 50x + 24y <= 2400)
@constraint(m, 30x + 33y <= 2100)

JuMP.optimize!(m)
println("Objective value: ", JuMP.objective_value(m))
println("x = ", JuMP.value(x))
println("y = ", JuMP.value(y))
```

1.2.5 Installing CPLEX

Instead of Gurobi, you can install and connect the CPLEX solver, which is also free to academics.

You can follow this step by step guide to install:

Copyrighted material

1. Go to the IBM ILOG CPLEX Optimization Studio page[10].
2. Click 'Access free academic edition.'
3. Log in with your institution email and certify.
4. Download an appropriate version of IBM ILOG CPLEX Optimization Studio. It should be v12.10 or higher.
5. Run the downloaded file and install CPLEX. I recommend using the default installation folder.
6. Add the `CPLEX` package in Julia. You have to tell Julia where the CPLEX library is installed.

   On Windows:

```
julia> ENV["CPLEX_STUDIO_BINARIES"] =
       "C:\\Program Files\\CPLEX_Studio1210\\cplex\\bin\\x86-64_win\\"
julia> using Pkg
julia> Pkg.add("CPLEX")
julia> Pkg.build("CPLEX")
```

   On macOS:

```
julia> ENV["CPLEX_STUDIO_BINARIES"] =
       "/Applications/CPLEX_Studio1210/cplex/bin/x86-64_osx/"
julia> using Pkg
julia> Pkg.add("CPLEX")
julia> Pkg.build("CPLEX")
```

7. Ready. Test the following code:

```
using JuMP, CPLEX
m = Model(CPLEX.Optimizer)
@variable(m, x <= 5)
@variable(m, y <= 45)
```

[10] https://www.ibm.com/products/ilog-cplex-optimization-studio

Copyrighted material

```
@objective(m, Max, x + y)
@constraint(m, 50x + 24y <= 2400)
@constraint(m, 30x + 33y <= 2100)

JuMP.optimize!(m)
println("Objective value: ", JuMP.objective_value(m))
println("x = ", JuMP.value(x))
println("y = ", JuMP.value(y))
```

1.3 Installing IJulia

You can also use an interactive Julia environment in your local computer, called Jupyter Notebook[11]. 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.

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`:

[11] http://jupyter.org

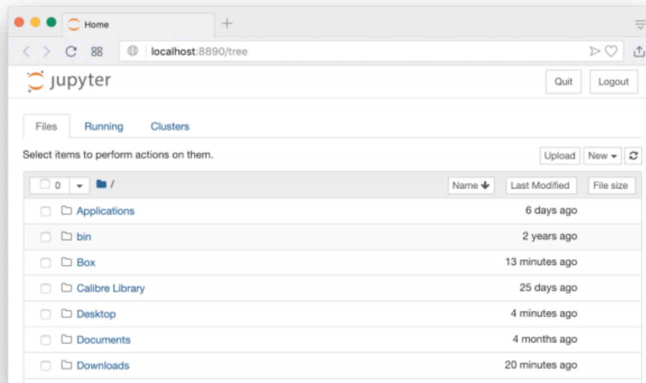
Copyrighted material

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

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

```
julia> using IJulia
julia> 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 1.1.0". See Figure 1.1.

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.

Copyrighted material

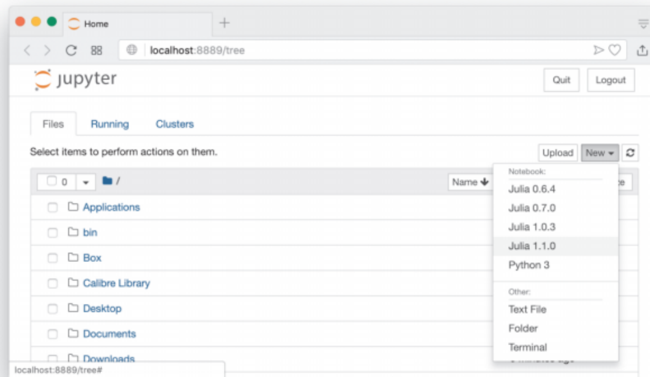


Figure 1.1: Creating a new notebook

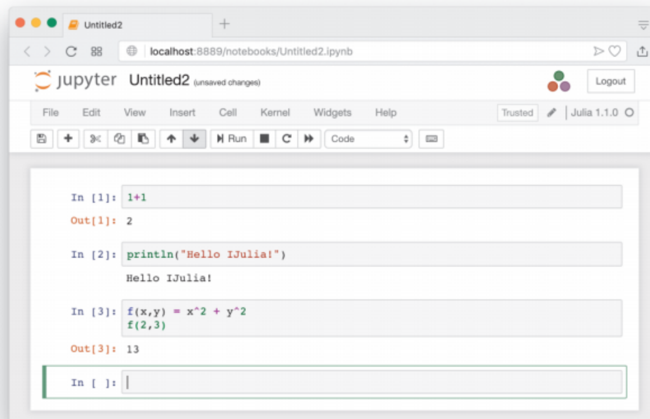


Figure 1.2: Some basic Julia codes.

Copyrighted material

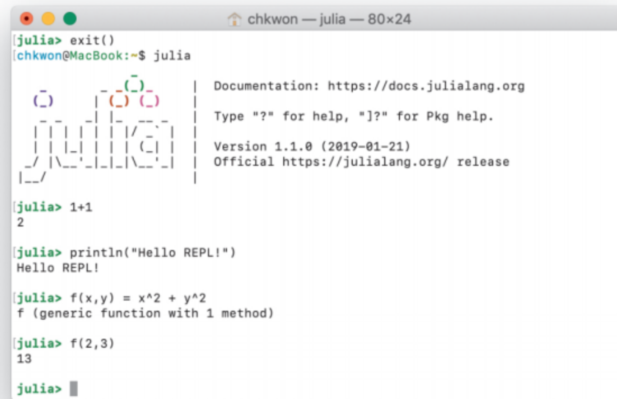


Figure 1.3: This is the REPL.

Personally, I prefer the REPL for most tasks, but I do 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 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.

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

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


Copyrighted material

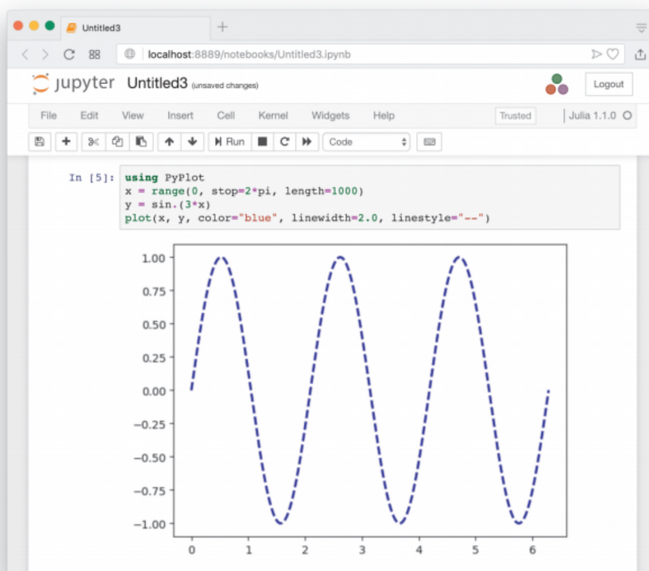


Figure 1.4: Plotting in IJulia

Copyrighted material

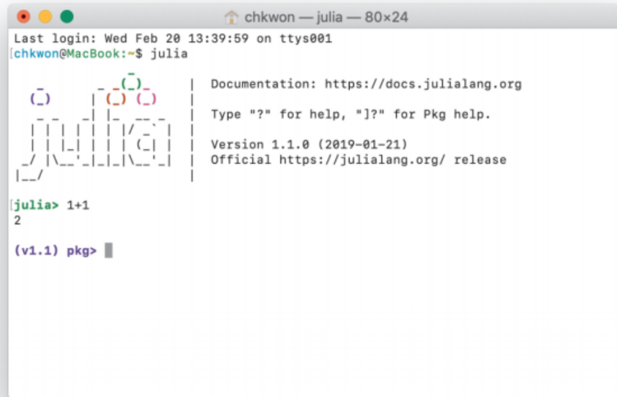


Figure 1.5: Package Mode in REPL

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`.

Using `Pkg.add` requires `using Pkg` first. In REPL, by pressing the ‘`]`’ key, you can enter the package management mode (Figure 1.5) and the prompt will change as follows:

```
(v1.3) pkg>
```

Then to install a package you can simply enter:

```
(v1.3) pkg> add PackageName
```

To install the `JuMP` package, you can do:

Copyrighted material

```
(v1.3) pkg> add JuMP
```

To come back to the julia prompt, press the backspace or delete key.

```
julia> Pkg.rm("PackageName")
(v1.3) pkg> rm PackageName
```

This removes the package.

```
julia> Pkg.update()
(v1.3) pkg> update
```

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

```
julia> Pkg.status()
(v1.3) 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.installed()["PackageName"]
```

```
julia> Pkg.build("PackageName")
(v1.3) pkg> build PackageName
```

Occasionally, 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.

Copyrighted material

1.5 Help

In REPL, you can use the Help mode. By pressing the `?` key in REPL, you can enter the help mode. The prompt will change as follows:

```
help?>
```

Then type in any function name, for example, `println`, which results in:

```
help?> println
search: println printstyled print sprint isprint

  println([io::IO], xs...)

  Print (using print) xs followed by a newline. If io is not supplied, prints to
  stdout.

  Examples


  julia> println("Hello, world")
  Hello, world

  julia> io = IOBuffer();

  julia> println(io, "Hello, world")

  julia> String(take!(io))
  "Hello, world\n"
```

See also Figure 1.6.

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

Copyrighted material

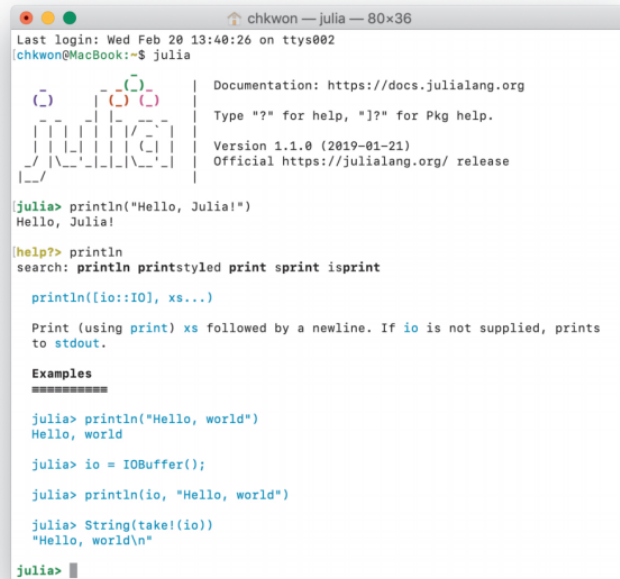


Figure 1.6: Help Mode in REPL

Copyrighted material

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 JuMP is worth visiting for information about the JuMP.jl package.

http://jump.dev

Copyrighted material

Copyrighted material

2

Simple Linear Optimization

This chapter provides a quick guide for solving simple linear optimization problems. For modeling, we use the JuMP package, and for computing, we use one of the following solvers.

- Clp: an open-source solver for linear programming (LP) problems from COIN-OR.
- Cbc: an open-source solver for mixed integer linear programming (MILP) problems from COIN-OR.
- GLPK: an open-source solver for mixed integer linear programming problem (MILP) problems from GNU.
- Gurobi: a commercial solver for both LP and MILP, free for academic users
- CPLEX: a commercial solver for both LP and MILP, free for academic users

Open-source solvers Clp, Cbc, and GLPK can be obtained by simply installing the corresponding Julia packages:

```
julia> using Pkg
julia> Pkg.add("Clp")
julia> Pkg.add("Cbc")
julia> Pkg.add("GLPK")
```


Copyrighted material

In fact, the `Clp` package automatically installs the `Cbc` package. COIN-OR is an open-source initiative, titled "Computational Infrastructure for Operations Research."

For commercial solvers Gurobi and CPLEX, one must first install the solver software, and then install the corresponding Julia packages:

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

There are a couple of things to do before you add Julia packages. See Sections 1.2.4 and 1.2.5 for the details.

There are some alternatives available, both open-source and commercial solvers. See the list of available solvers via JuMP[1]. Nonlinear optimization solvers will be discussed in Chapter 8.

2.1 Linear Programming (LP) Problems

Once you have installed the JuMP package and an optimization solver mentioned above, we can have Julia solve linear programming (LP) and mixed integer linear programming (MILP) problems easily. For example, consider the following LP problem:

$$\max \quad x_1 + 2x_2 + 5x_3$$

subject to

$$\begin{aligned} -x_1 + x_2 + 3x_3 &\leq -5 \\ x_1 + 3x_2 - 7x_3 &\leq 10 \\ 0 \leq x_1 &\leq 10 \\ x_2 &\geq 0 \\ x_3 &\geq 0. \end{aligned}$$

Using Julia and JuMP, we can write the following code:

[1] http://jump.dev/JuMP.jl/stable/installation/

Copyrighted material

Listing 2.1: LP Example 1
code/chap2/LP1.jl

```
using JuMP, GLPK

# Preparing an optimization model
m = Model(GLPK.Optimizer)

# Declaring variables
@variable(m, 0<= x1 <=10)
@variable(m, x2 >=0)
@variable(m, x3 >=0)

# Setting the objective
@objective(m, Max, x1 + 2x2 + 5x3)

# Adding constraints
@constraint(m, constraint1, -x1 +  x2 + 3x3 <= -5)
@constraint(m, constraint2,  x1 + 3x2 - 7x3 <= 10)

# Printing the prepared optimization model
print(m)

# Solving the optimization problem
JuMP.optimize!(m)

# Printing the optimal solutions obtained
println("Optimal Solutions:")
println("x1 = ", JuMP.value(x1))
println("x2 = ", JuMP.value(x2))
println("x3 = ", JuMP.value(x3))

# Printing the optimal dual variables
println("Dual Variables:")
println("dual1 = ", JuMP.shadow_price(constraint1))
println("dual2 = ", JuMP.shadow_price(constraint2))
```

The above code is pretty much self-explanatory, but here are some explanations. We first declare a placeholder for an optimization model:

Copyrighted material

```
m = Model(GLPK.Optimizer)
```

where we also indicated that we want to use the GLPK optimization solver. We call the model `m`.

We declare three variables:

```
@variable(m, 0<= x1 <=10)
@variable(m, x2 >= 0)
@variable(m, x3 >= 0)
```

where we used 'macros' from the `JuMP` package, `@variable`. In Julia, macros do repeated jobs for you. It is somewhat similar to 'functions' with some important differences. Refer to the official documentation[2].

Using another macro `@objective`, we set the objective:

```
@objective(m, Max, x1 + 2x2 + 5x3)
```

Two constraints are added by the `@constraint` macro:

```
@constraint(m, constraint1, -x1 +  x2 + 3x3 <= -5)
@constraint(m, constraint2,  x1 + 3x2 - 7x3 <= 10)
```

Note that `constraint1` and `constraint2` are the names of those constraints. These names will be useful for obtaining the corresponding dual variable values.

We are now ready with the optimization problem. If you like you can print the optimization model and check how it is written, the code is as simple as:

```
print(m)
```

We solve the optimization problem:

[2] http://docs.julialang.org/en/v1/manual/metaprogramming/#macros

Copyrighted material

```
JuMP.optimize!(m)
```

After solving the optimization problem, we can obtain the values of variables at the optimality by using the `JuMP.value()` function:

```
println("Optimal Solutions:")
println("x1 = ", JuMP.value(x1))
println("x2 = ", JuMP.value(x2))
println("x3 = ", JuMP.value(x3))
```

where `println()` is a function that puts some text in a line on the screen. If you don't want to change the line after you print the text, use the `print()` function instead.

To obtain the values of optimal dual variables, call `JuMP.shadow_price()` with the corresponding constraint names as follows:

```
println("Dual Variables:")
println("dual1 = ", JuMP.shadow_price(constraint1))
println("dual2 = ", JuMP.shadow_price(constraint2))
```

IMPORTANT: There is also the `JuMP.dual()` function defined. However, the sign of `JuMP.dual()` results might not be as you would expect, since it follows the convention of conic duality. For linear optimization problems, `JuMP.shadow_price()` provides dual variable values as defined in most standard textbooks. Please refer to the relevant discussion in the JuMP documentation[3].

In my machine, the output by Gurobi looks like:

```
julia> include("LP1.jl")
Max x1 + 2 x2 + 5 x3
Subject to
 x1  0.0
 x2  0.0
 x3  0.0
```

[3] http://jump.dev/JuMP.jl/stable/constraints/#constraint_duality-1

Copyrighted material

```
 x1  10.0
 -x1 + x2 + 3 x3  -5.0
 x1 + 3 x2 - 7 x3  10.0
Optimal Solutions:
x1 = 10.0
x2 = 2.1875
x3 = 0.9375
Dual Variables:
dual1 = 1.8125
dual2 = 0.06249999999999998
```

If you want to use the Gurobi optimization solver instead of GLPK, use the following inputs:

```
using JuMP, Gurobi
m = Model(Gurobi.Optimizer)
```

For CPLEX:

```
using JuMP, CPLEX
m = Model(CPLEX.Optimizer)
```

There are many other optimization solvers supported by the `JuMP` package. See the manual of JuMP for a list.[4]

2.2 Alternative Ways of Writing LP Problems

We can use arrays to define variables. For the same LP problem as in the previous section, we can write a Julia code alternatively as follows:

To define the variable $\mathbf{x}$ as a three-dimensional vector, we can write:

```
@variable(m, x[1:3] >= 0)
```

[4] http://jump.dev/JuMP.jl/stable/installation/

Copyrighted material

```
@constraint(m, bound, x[1] <= 10)
```

The final code is presented:

Listing 2.2: LP Example 2
code/chap2/LP2.jl

```
using JuMP, GLPK
m = Model(GLPK.Optimizer)

c = [ 1; 2; 5]
A = [-1  1  3;
      1  3 -7]
b = [-5; 10]

@variable(m, x[1:3] >= 0)
@objective(m, Max, sum( c[i]*x[i] for i in 1:3) )

@constraint(m, constraint[j in 1:2], sum( A[j,i]*x[i] for i in 1:3 ) <= b[j] )
@constraint(m, bound, x[1] <= 10)

JuMP.optimize!(m)


println("Optimal Solutions:")
for i in 1:3
  println("x[$i] = ", JuMP.value(x[i]))
end

println("Dual Variables:")
for j in 1:2
  println("dual[$j] = ", JuMP.shadow_price(constraint[j]))
end
```

Note that there have been changes in the code for printing. The result looks like:

```
julia> include("LP2.jl")
Optimal Solutions:
x[1] = 10.0
```


Copyrighted material

$$
\begin{aligned}
x_1 + 3x_2 - 7x_3 &\leq 10 \\
0 \leq x_1 &\leq 10 \\
x_2 &\geq 0 \text{ Integer} \\
x_3 &\in \{0, 1\}.
\end{aligned}
$$

Using JuMP, it is very simple to specify integer and binary variables. We can define variables as follows:

```
@variable(m, 0<= x1 <=10)
@variable(m, x2 >=0, Int)
@variable(m, x3, Bin)
```

The complete code would look like:

Listing 2.4: MILP Example 1
code/chap2/MILP1.jl

```
using JuMP, GLPK

# Preparing an optimization model
m = Model(GLPK.Optimizer)

# Declaring variables
@variable(m, 0<= x1 <=10)
@variable(m, x2 >=0, Int)
@variable(m, x3, Bin)

# Setting the objective
@objective(m, Max, x1 + 2x2 + 5x3)

# Adding constraints
@constraint(m, constraint1, -x1 +  x2 + 3x3 <= -5)
@constraint(m, constraint2,  x1 + 3x2 - 7x3 <= 10)

# Printing the prepared optimization model
print(m)

# Solving the optimization problem
JuMP.optimize!(m)
```


Copyrighted material

```
# Printing the optimal solutions obtained
println("Optimal Solutions:")
println("x1 = ", JuMP.value(x1))
println("x2 = ", JuMP.value(x2))
println("x3 = ", JuMP.value(x3))
```

The result looks like:

```
julia> include("MILP1.jl")
Max x1 + 2 x2 + 5 x3
Subject to
 x3 binary
 x2 integer
 x1  0.0
 x2  0.0
 x1  10.0
 -x1 + x2 + 3 x3  -5.0
 x1 + 3 x2 - 7 x3  10.0
Optimal Solutions:
x1 = 10.0
x2 = 2.0
x3 = 1.0
```


Copyrighted material

3 Basics of the Julia Language

In this chapter, I cover how we can do most common tasks for computing in operations research and management science with the Julia Language. While I will cover some part of the syntax of Julia, readers must consult with the official documentation[1] of Julia for other unexplained usages.

3.1 Vector, Matrix, and Array

Like MATLAB and many other computer languages for numerical computation, Julia provides easy and convenient, but strong, ways of handling vectors and matrices. For example, if you want to create vectors and matrices like

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}, \qquad \mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

then in Julia, you can simply type

```
a = [1; 2; 3]
b = [4 5 6]
A = [1 2 3; 4 5 6]
```

[1] http://docs.julialang.org/

Copyrighted material

where the semicolon (;) means a new row. Julia will return:

```
julia> a = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> b = [4 5 6]
1x3 Array{Int64,2}:
 4  5  6

julia> A = [1 2 3; 4 5 6]
2x3 Array{Int64,2}:
 1  2  3
 4  5  6
```

We can access the (i,j)-element of $\mathbf{A}$ by `A[i,j]`:

```
julia> A[1,3]
3

julia> A[2,1]
4
```

The transpose of vectors and matrices is easily obtained either of the following codes:

```
julia> transpose(A)
3x2 Array{Int64,2}:
 1  4
 2  5
 3  6

julia> A'
3x2 Array{Int64,2}:
 1  4
 2  5
 3  6
```


Copyrighted material

```
julia> B * inv(B)
3x3 Array{Float64,2}:
  1.0           0.0  0.0
  0.0           1.0  0.0
 -2.22045e-16   0.0  1.0
```

Note that the off-diagonal elements are not exactly zero. This is because the computation of the inverse matrix is not exact. For example, the (2,1)-element of the inverse matrix is not exactly 1, but:

```
julia> inv(B)[2,1]
1.0000000000000004
```

In the above, we have seen something like `Int64` and `Float64`. In 32-bit systems, it would have been `Int32` and `Float32`. These are data types. If the elements in your vectors and matrices are integers for sure, you can use `Int64`. On the other hand, if any element is non-integer values, such as 1.0000000000000004, you need to use `Float64`. These are usually done automatically:

```
julia> a = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> b = [1.0; 2; 3]
3-element Array{Float64,1}:
 1.0
 2.0
 3.0
```

In some cases, you will want to first create an array object of a certain type, then assign values. This can be done by calling `Array` with a keyword `undef`. For example, if we want an array of `Float64` data type and of size 3, then we can do:

Copyrighted material

Index

Symbols

.nl format 171

A

add_edge! 142
adjacency matrix 187
affine scaling algorithm 151
AMPL 171
AmplNLWriter 172
argmin 113, 211
Array 46, 109, 181, 212
array 41
  tuples 47, 128
@assert 99, 204
automatic differentiation 91

B

Bernoulli distribution 178
bi-level optimization 171
big-M 242
Bonmin 171
box-constrained optimization 168
break 55
Broyden-Fletcher-Goldfarb-Shanno, BFGS 168

C

Calculus 86
Clp 104
COIN-OR 30, 172
collect 50, 100, 110
collection 54
colon (:) 50
combinations 100, 110
Combinatorics 100
comma separated values, CSV 68, 125, 127, 135
commodity 123
Complementarity 227
  @complementarity 228
  @mapping 228

Copyrighted material

complementarity
  linear complementarity problems, LCP ... 225
  mixed complementarity problems, MCP ... 237
  nonlinear complementarity problems, NCP ... 233
complementarity problems ... 225
conjugate gradient ... 168
`convert` ... 203
`copy` ... 190
Couenne ... 171
CPLEX ... 3, 104, 239
  installation ... 16
  parameters ... 245
CPU time ... 239
curve fitting ... 79
`curve_fit` ... 82

D

DelimitedFiles ... 70
`derivative` ... 86, 93
`Diagonal` ... 154
`Dict` ... 55, 128, 136, 146
dictionary ... 54
Dijkstra's algorithm ... 140, 144
`dijkstra_shortest_paths` ... 142
Distributions ... 177
`dot` ... 43, 103

E

`end` ... 49
`enumerate_paths` ... 142
equilibrium
  traffic equilibrium ... 233
  Wardrop equilibrium ... 233

F

`fieldnames` ... 108
file input/output ... 67
`findall` ... 113, 115, 191
`findfirst` ... 113
finite difference ... 84
`Fminbox` ... 169
`for` ... 54
ForwardDiff ... 92
`function` ... 57

G

GAMS ... 3
Gauss-Kronrod integration method 90
Geometric distribution ... 178
global optimization ... 171
GLPK ... 104
Golden Section ... 167
`gradient` ... 87, 93
graph ... 123
Gurobi ... 4, 104, 239
  installation ... 15
  parameters ... 245

H

`hessian` ... 87, 93

I

identity matrix ... 43
IJulia ... 18
importance sampling ... 195
index ... 48
`Inf` ... 99, 126
inner product ... 43
integrality tolerance ... 242
`inv` ... 44, 101, 154

Copyrighted material

inverse matrix ... 44
Ipopt ... 169
!isempty ... 147

J

Julia
  installation ... 4
    macOS ... 10
    Windows ... 5
JuMP ... 3
  Bin ... 39
  @constraint ... 32, 35, 38
  getobjectivevalue ... 130
  Int ... 39
  @NLconstraint ... 170
  @NLobjective ... 170
  @objective ... 32, 35
  optimize ... 33
  print() ... 32
  set_start_value ... 241
  shadow_price() ... 33
  value ... 33
  @variable ... 32, 35

L

label ... 156
Lagrangian relaxation ... 197
least-squares fit ... 81
legend ... 156
length ... 53, 55, 127, 204
Levenberg-Marquardt algorithm ... 81
LightGraphs ... 141
line search ... 165
linear programming, LP ... 30, 95
linear regression ... 80
LinearAlgebra ... 43, 99, 154
linearization ... 242
link ... 123
loop ... 54
lower bound ... 199
LsqFit ... 81

M

margin_error ... 82
mathematical program with complementarity conditions, MPCC ... 172
MATLAB ... 3
matplotlib ... 72
matrix ... 41
max ... 127
maximum ... 127, 211
minimal-cost network-flow problem 123
mixed integer linear programming, MILP ... 38
mixed integer nonlinear programming, MINLP ... 171, 175
module ... 116
Monte Carlo ... 177

N

Nelder-Mead ... 168
network optimization ... 123
NLsolve ... 227
node ... 123
nonconvex nonlinear optimization 171
nonlinear optimization ... 169
nonlinear programming, NLP ... 169
norm ... 154
Normal distribution ... 64, 177
  multi-variate ... 179

Copyrighted material

normcdf 66
norminvcdf 66
normpdf 66
numerical differentiation 84
numerical integration 87
numerical methods 79

O

ones 44, 214
Optim 165, 168
optimality gap 199, 240
optimization 165
  bi-level 171
  box-constrained 168
  global 171
  nonconvex nonlinear 171
  nonlinear 169
  unconstrained 165
optimize 167, 168

P

p-median problem 201
package management 21
packages
  AmplNLWriter 172
  Calculus 86
  Combinatorics 99, 100
  Complementarity 227
  CPLEX 17
  DelimitedFiles 70
  Distributions 177
  ForwardDiff 92
  GLPK 7, 12
  Gurobi 16
  IJulia 19
  Ipopt 169
  LightGraphs 141
  LinearAlgebra 43, 99, 154
  LsqFit 81
  NLsolve 227
  Optim 165, 168
  PathDistribution 195
  PATHSolver 227
  Plots 72
  Printf 53
  PyPlot 72, 156, 218
  QuadGK 90
  StatsFuns 65
PathDistribution 195
PATHSolver 227
plot 156
Plots 72
plotting 72
primal path following algorithm 157
print 52
Printf 53
@printf 53
println 52
probability distribution
  Uniform distribution 63
probability distributions 177
  Bernoulli 178
  Binomial 66
  Gamma 66
  Geometric 178
  Normal 66, 177
    multi-variate 179
  Normal distribution 64
  Poisson 66
push! 214
PyPlot 72, 156, 218

Copyrighted material

Python 3

Q

QuadGK 90
quadgk 90

R

rand 63, 178, 192
randn 64
random number 63
randomized linear program 179
range 48
rank 99
readdlm 70
REPL 21
revenue management 179
Riemann sum 88
round 126

S

scope blocks 60
scope of variables 59
second_derivative 87
semicolon (;) 42
Set 146
setdiff 110, 115, 148
shortest path problem 139
simplex method 95
Simpson's rule 90
simulated annealing 168
sink node 123
size 99, 109, 204
solvers 29
  Bonmin 172
  Cbc 29
  Clp 29
  Couenne 172
  CPLEX 16, 29
  GLPK 7, 12, 29
  Gurobi 15, 29
  parameters 239, 245
sorting 207
sortperm 207
source node 123
@sprintf 54
StatsFuns 65
subgradient optimization 200
sum 182, 214

T

Taylor series 85
tolerance
  integrality 242
  optimality gap 240
traffic assignment 233
traffic equilibrium 233
transportation problem 133
transpose 43, 154, 214
trapezoidal rule 89
Tuple 128, 146
tuple 47
type 107
typeof 108

U

unconstrained optimization 165
undef 46
upper bound 199

V

vector 41

Copyrighted material