

G *Julia*

Julia is a scientific programming language that is free and open source.¹ It is a relatively new language that borrows inspiration from languages like Python, MATLAB, and R. It was selected for use in this book because it is sufficiently high level² so that the algorithms can be compactly expressed and readable while also being fast. This book is compatible with Julia version 1.5. This appendix introduces the concepts necessary for understanding the included code, omitting many of the advanced features of the language.

¹ Julia may be obtained from <http://julia-lang.org>.

² In contrast with languages like C++, Julia does not require programmers to worry about memory management and other lower-level details, yet it allows low-level control when needed.

G.1 *Types*

Julia has a variety of basic types that can represent data given as truth values, numbers, strings, arrays, tuples, and dictionaries. Users can also define their own types. This section explains how to use some of the basic types and how to define new types.

G.1.1 *Booleans*

The *Boolean* type in Julia, written `Bool`, includes the values `true` and `false`. We can assign these values to variables. Variable names can be any string of characters, including Unicode, with a few restrictions.

```
α = true  
done = false
```

The variable name appears on the left-hand side of the equal sign; the value that variable is to be assigned is on the right-hand side.

We can make assignments in the Julia console. The console, or REPL (for read, eval, print, loop), will return a response to the expression being evaluated. The # symbol indicates that the rest of the line is a comment.

```
julia> x = true
true
julia> y = false; # semicolon suppresses the console output
julia> typeof(x)
Bool
julia> x == y # test for equality
false
```

The standard Boolean operators are supported.

```
julia> !x # not
false
julia> x && y # and
false
julia> x || y # or
true
```

G.1.2 Numbers

Julia supports integer and floating point numbers as shown here:

```
julia> typeof(42)
Int64
julia> typeof(42.0)
Float64
```

Here, `Int64` denotes a 64-bit integer, and `Float64` denotes a 64-bit floating point value.³ We can perform the standard mathematical operations:

```
julia> x = 4
4
julia> y = 2
2
julia> x + y
6
julia> x - y
2
julia> x * y
8
julia> x / y
2.0
```

³On 32-bit machines, an integer literal like `42` is interpreted as an `Int32`.

```

julia> x ^ y # exponentiation
16
julia> x % y # remainder from division
0
julia> div(x, y) # truncated division returns an integer
2

```

Note that the result of x / y is a `Float64`, even when x and y are integers. We can also perform these operations at the same time as an assignment. For example, $x += 1$ is shorthand for $x = x + 1$.

We can also make comparisons:

```

julia> 3 > 4
false
julia> 3 >= 4
false
julia> 3 ≥ 4 # unicode also works, use \ge[tab] in console
false
julia> 3 < 4
true
julia> 3 <= 4
true
julia> 3 ≤ 4 # unicode also works, use \le[tab] in console
true
julia> 3 == 4
false
julia> 3 < 4 < 5
true

```

G.1.3 Strings

A *string* is an array of characters. Strings are not used very much in this textbook except for reporting certain errors. An object of type `String` can be constructed using `"` characters. For example:

```

julia> x = "optimal"
"optimal"
julia> typeof(x)
String

```

G.1.4 Symbols

A *symbol* represents an identifier. It can be written using the `:` operator or constructed from strings.

```

julia> :A
:A
julia> :Battery
:Battery
julia> Symbol("Failure")
:Failure

```

G.1.5 Vectors

A *vector* is a one-dimensional array that stores a sequence of values. We can construct a vector using square brackets, separating elements by commas.

```

julia> x = []; # empty vector
julia> x = trues(3); # Boolean vector containing three trues
julia> x = ones(3); # vector of three ones
julia> x = zeros(3); # vector of three zeros
julia> x = rand(3); # vector of three random numbers between 0 and 1
julia> x = [3, 1, 4]; # vector of integers
julia> x = [3.1415, 1.618, 2.7182]; # vector of floats

```

An *array comprehension* can be used to create vectors.

```

julia> [sin(x) for x = 1:5]
5-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
-0.7568024953079282
-0.9589242746631385

```

We can inspect the type of vectors:

```

julia> typeof([3, 1, 4]) # 1-dimensional array of Int64s
Array{Int64,1}
julia> typeof([3.1415, 1.618, 2.7182]) # 1-dimensional array of Float64s
Array{Float64,1}
julia> Vector{Float64} # alias for a 1-dimensional array
Array{Float64,1}

```

We index into vectors using square brackets.

```

julia> x[1]      # first element is indexed by 1
3.1415
julia> x[3]      # third element
2.7182
julia> x[end]    # use end to reference the end of the array
2.7182
julia> x[end-1] # this returns the second to last element
1.618

```

We can pull out a range of elements from an array. Ranges are specified using a colon notation.

```

julia> x = [1, 2, 5, 3, 1]
5-element Array{Int64,1}:
 1
 2
 5
 3
 1
julia> x[1:3]      # pull out the first three elements
3-element Array{Int64,1}:
 1
 2
 5
julia> x[1:2:end] # pull out every other element
3-element Array{Int64,1}:
 1
 5
 1
julia> x[end:-1:1] # pull out all the elements in reverse order
5-element Array{Int64,1}:
 1
 3
 5
 2
 1

```

We can perform a variety of different operations on arrays. The exclamation mark at the end of function names is used to indicate that the function *mutates* (i.e., changes) the input.

```

julia> length(x)
5
julia> [x, x]      # concatenation
2-element Array{Array{Int64,1},1}:

```

```

[1, 2, 5, 3, 1]
[1, 2, 5, 3, 1]
julia> push!(x, -1)      # add an element to the end
6-element Array{Int64,1}:
 1
 2
 5
 3
 1
-1
julia> pop!(x)          # remove an element from the end
-1
julia> append!(x, [2, 3]) # append [2, 3] to the end of x
7-element Array{Int64,1}:
 1
 2
 5
 3
 1
 2
 3
julia> sort!(x)         # sort the elements, altering the same vector
7-element Array{Int64,1}:
 1
 1
 2
 2
 3
 3
 5
julia> sort(x);        # sort the elements as a new vector
julia> x[1] = 2; print(x) # change the first element to 2
[2, 1, 2, 2, 3, 3, 5]
julia> x = [1, 2];
julia> y = [3, 4];
julia> x + y           # add vectors
2-element Array{Int64,1}:
 4
 6
julia> 3x - [1, 2]     # multiply by a scalar and subtract
2-element Array{Int64,1}:
 2
 4
julia> using LinearAlgebra

```

```

 julia> dot(x, y)           # dot product available after using LinearAlgebra
11
 julia> x⋅y                 # dot product using unicode character
11
 julia> prod(y)             # product of all the elements in y
12

```

It is often useful to apply various functions elementwise to vectors. This is a form of *broadcasting*. With infix operators (e.g., `+`, `*`, and `^`), a dot is prefixed to indicate elementwise broadcasting. With functions like `sqrt` and `sin`, the dot is postfixed.

```

 julia> x .* y             # elementwise multiplication
2-element Array{Int64,1}:
 3
 8
 julia> x .^ 2            # elementwise squaring
2-element Array{Int64,1}:
 1
 4
 julia> sin.(x)          # elementwise application of sin
2-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 julia> sqrt.(x)         # elementwise application of sqrt
2-element Array{Float64,1}:
 1.0
 1.4142135623730951

```

G.1.6 Matrices

A *matrix* is a two-dimensional array. Like a vector, it is constructed using square brackets. We use spaces to delimit elements in the same row and semicolons to delimit rows. We can also index into the matrix and output submatrices using ranges.

```

 julia> X = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
 julia> typeof(X)        # a 2-dimensional array of Int64s
Array{Int64,2}
 julia> X[2]             # second element using column-major ordering
4
 julia> X[3,2]          # element in third row and second column
8

```

```

Julia> X[1,:] # extract the first row
3-element Array{Int64,1}:
 1
 2
 3
Julia> X[:,2] # extract the second column
4-element Array{Int64,1}:
 2
 5
 8
11
Julia> X[:,1:2] # extract the first two columns
4x2 Array{Int64,2}:
 1  2
 4  5
 7  8
10 11
Julia> X[1:2,1:2] # extract a 2x2 submatrix from the top left of x
2x2 Array{Int64,2}:
 1  2
 4  5
Julia> Matrix{Float64} # alias for a 2-dimensional array
Array{Float64,2}

```

We can also construct a variety of special matrices and use array comprehensions:

```

Julia> Matrix(1.0I, 3, 3) # 3x3 identity matrix
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
Julia> Matrix(Diagonal([3, 2, 1])) # 3x3 diagonal matrix with 3, 2, 1 on diagonal
3x3 Array{Int64,2}:
 3  0  0
 0  2  0
 0  0  1
Julia> zeros(3,2) # 3x2 matrix of zeros
3x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
 0.0  0.0
Julia> rand(3,2) # 3x2 random matrix
3x2 Array{Float64,2}:
 0.026288  0.759272

```



```

0.887502  0.989538
0.0886743 0.91351
Julia> [sin(x + y) for x = 1:3, y = 1:2] # array comprehension
3×2 Array{Float64,2}:
 0.909297  0.14112
 0.14112   -0.756802
-0.756802 -0.958924

```

Matrix operations include the following:

```

Julia> X' # complex conjugate transpose
3×4 LinearAlgebra.Adjoint{Int64,Array{Int64,2}}:
 1  4  7 10
 2  5  8 11
 3  6  9 12
Julia> 3X .+ 2 # multiplying by scalar and adding scalar
4×3 Array{Int64,2}:
 5  8 11
14 17 20
23 26 29
32 35 38
Julia> X = [1 3; 3 1]; # create an invertible matrix
Julia> inv(X) # inversion
2×2 Array{Float64,2}:
-0.125  0.375
 0.375 -0.125
Julia> det(X) # determinant (requires LinearAlgebra)
-8.0
Julia> [X X] # horizontal concatenation, same as hcat(X, X)
2×4 Array{Int64,2}:
 1  3  1  3
 3  1  3  1
Julia> [X; X] # vertical concatenation, same as vcat(X, X)
4×2 Array{Int64,2}:
 1  3
 3  1
 1  3
 3  1
Julia> sin.(X) # elementwise application of sin
2×2 Array{Float64,2}:
 0.841471  0.14112
 0.14112   0.841471
Julia> map(sin, X) # elementwise application of sin
2×2 Array{Float64,2}:
 0.841471  0.14112

```

```

0.14112  0.841471
julia> vec(X)           # reshape an array as a vector
4-element Array{Int64,1}:
 1
 3
 3
 1

```

G.1.7 Tuples

A *tuple* is an ordered list of values, potentially of different types. They are constructed with parentheses. They are similar to vectors, but they cannot be mutated.

```

julia> x = ()          # the empty tuple
()
julia> isempty(x)
true
julia> x = (1,)       # tuples of one element need the trailing comma
(1,)
julia> typeof(x)
Tuple{Int64}
julia> x = (1, 0, [1, 2], 2.5029, 4.6692) # third element is a vector
(1, 0, [1, 2], 2.5029, 4.6692)
julia> typeof(x)
Tuple{Int64,Int64,Array{Int64,1},Float64,Float64}
julia> x[2]
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
julia> length(x)
5
julia> x = (1, 2)
(1, 2)
julia> a, b = x;
julia> a
1
julia> b
2

```

G.1.8 Named Tuples

A *named tuple* is like a tuple but where each entry has its own name.

```

julia> x = (a=1, b=-Inf)
(a = 1, b = -Inf)
julia> x isa NamedTuple
true
julia> x.a
1
julia> a, b = x;
julia> a
1
julia> (; :a⇒10)
(a = 10,)
julia> (; :a⇒10, :b⇒11)
(a = 10, b = 11)
julia> (; (:a, :b), (10, 11)) # [:a, :b] also works here
Error: syntax: invalid named tuple element "(:a, :b)"
julia> merge(x, (d=3, e=10)) # merge two named tuples
(a = 1, b = -Inf, d = 3, e = 10)

```

G.1.9 Dictionaries

A *dictionary* is a collection of key-value pairs. Key-value pairs are indicated with a double arrow operator `=>`. We can index into a dictionary using square brackets just as with arrays and tuples.

```

julia> x = Dict(); # empty dictionary
julia> x[3] = 4 # associate key 3 with value 4
4
julia> x = Dict{3⇒4, 5⇒1} # create a dictionary with two key-value pairs
Dict{Int64,Int64} with 2 entries:
 3 ⇒ 4
 5 ⇒ 1
julia> x[5] # return value associated with key 5
1
julia> haskey(x, 3) # check whether dictionary has key 3
true
julia> haskey(x, 4) # check whether dictionary has key 4
false

```

G.1.10 Composite Types

A *composite type* is a collection of named fields. By default, an instance of a composite type is immutable (i.e., it cannot change). We use the `struct` keyword and then give the new type a name and list the names of the fields.

```

struct A
    a
    b
end

```

Adding the keyword `mutable` makes it so that an instance can change.

```

mutable struct B
    a
    b
end

```

Composite types are constructed using parentheses, between which we pass in values for each field. For example,

```
x = A(1.414, 1.732)
```

The double-colon operator can be used to specify the type for any field.

```

struct A
    a::Int64
    b::Float64
end

```

These type annotations require that we pass in an `Int64` for the first field and a `Float64` for the second field. For compactness, this text does not use type annotations, but it is at the expense of performance. Type annotations allow Julia to improve runtime performance because the compiler can optimize the underlying code for specific types.

G.1.11 Abstract Types

So far we have discussed *concrete types*, which are types that we can construct. However, concrete types are only part of the type hierarchy. There are also *abstract types*, which are supertypes of concrete types and other abstract types.

We can explore the type hierarchy of the `Float64` type shown in figure G.1 using the `supertype` and `subtypes` functions.

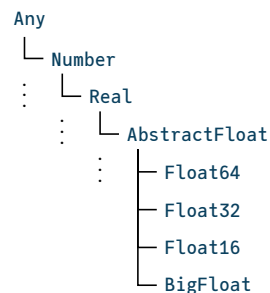


Figure G.1. The type hierarchy for the `Float64` type.

```

julia> supertype(Float64)
AbstractFloat
julia> supertype(AbstractFloat)
Real
julia> supertype(Real)
Number
julia> supertype(Number)
Any
julia> supertype(Any)           # Any is at the top of the hierarchy
Any
julia> using InteractiveUtils   # required for using subtypes in scripts
julia> subtypes(AbstractFloat) # different types of AbstractFloats
4-element Array{Any,1}:
  BigFloat
  Float16
  Float32
  Float64
julia> subtypes(Float64)       # Float64 does not have any subtypes
Type[]

```

We can define our own abstract types.

```

abstract type C end
abstract type D <: C end # D is an abstract subtype of C
struct E <: D # E is composite type that is a subtype of D
    a
end

```

G.1.12 Parametric Types

Julia supports *parametric types*, which are types that take parameters. The parameters to a parametric type are given within braces and delimited by commas. We have already seen a parametric type with our dictionary example.

```

julia> x = Dict{3⇒1.4, 1⇒5.9}
Dict{Int64,Float64} with 2 entries:
  3 ⇒ 1.4
  1 ⇒ 5.9

```

For dictionaries, the first parameter specifies the key type, and the second parameter specifies the value type. The example has `Int64` keys and `Float64` values, making the dictionary of type `Dict{Int64,Float64}`. Julia was able to infer these types based on the input, but we could have specified it explicitly.

```

julia> x = Dict{Int64,Float64}Dict(3⇒1.4, 1⇒5.9);
Error: MethodError: no method matching *(::Type{Dict{Int64,Float64}}, ::Dict{Int64,Float64})
Closest candidates are:
  *(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:538

```

While it is possible to define our own parametric types, we do not need to do so in this text.

G.2 Functions

A *function* maps its arguments, given as a tuple, to a result that is returned. This section discusses how to define and work with functions.

G.2.1 Named Functions

One way to define a *named function* is to use the **function** keyword, followed by the name of the function and a tuple of names of arguments.

```

function f(x, y)
    return x + y
end

```

We can also define functions compactly using assignment form.

```

julia> f(x, y) = x + y;
julia> f(3, 0.1415)
3.1415

```

G.2.2 Anonymous Functions

An *anonymous function* is not given a name, though it can be assigned to a named variable. One way to define an anonymous function is to use the arrow operator.

```

julia> h = x -> x^2 + 1 # assign anonymous function with input x to a variable h
#1 (generic function with 1 method)
julia> h(3)
10
julia> g(f, a, b) = [f(a), f(b)]; # applies function f to a and b and returns array
julia> g(h, 5, 10)
2-element Array{Int64,1}:
 26
 101

```

```

julia> g(x->sin(x)+1, 10, 20)
2-element Array{Float64,1}:
 0.4559788891106302
 1.9129452507276277

```

G.2.3 Callable Objects

We can define a type and associate functions with it, allowing objects of that type to be *callable*.

```

julia> (x::A)() = x.a + x.b # adding a zero-argument function to the type A defined earlier
julia> (x::A)(y) = y*x.a + x.b # adding a single-argument function
julia> x = A(22, 8);
julia> x()
30
julia> x(2)
52

```

G.2.4 Optional Arguments

We can assign a default value to an argument, making the specification of that argument optional.

```

julia> f(x=10) = x^2;
julia> f()
100
julia> f(3)
9
julia> f(x, y, z=1) = x*y + z;
julia> f(1, 2, 3)
5
julia> f(1, 2)
3

```

G.2.5 Keyword Arguments

Functions may use keyword arguments, which are arguments that are named when the function is called. Keyword arguments are given after all the positional arguments. A semicolon is placed before any keywords, separating them from the other arguments.

```

julia> f(; x = 0) = x + 1;
julia> f()
1
julia> f(x = 10)
11
julia> f(x, y = 10; z = 2) = (x + y)*z;
julia> f(1)
22
julia> f(2, z = 3)
36
julia> f(2, 3)
10
julia> f(2, 3, z = 1)
5

```

G.2.6 Dispatch

The types of the arguments passed to a function can be specified using the double colon operator. If multiple methods of the same function are provided, Julia will execute the appropriate method. The mechanism for choosing which method to execute is called *dispatch*.

```

julia> f(x::Int64) = x + 10;
julia> f(x::Float64) = x + 3.1415;
julia> f(1)
11
julia> f(1.0)
4.1415000000000001
julia> f(1.3)
4.4415000000000004

```

The method with a type signature that best matches the types of the arguments given will be used.

```

julia> f(x) = 5;
julia> f(x::Float64) = 3.1415;
julia> f([3, 2, 1])
5
julia> f(0.00787499699)
3.1415

```


G.2.7 Splatting

It is often useful to *splat* the elements of a vector or a tuple into the arguments to a function using the `...` operator.

```

julia> f(x,y,z) = x + y - z;
julia> a = [3, 1, 2];
julia> f(a...)
2
julia> b = (2, 2, 0);
julia> f(b...)
4
julia> c = ([0,0],[1,1]);
julia> f([2,2], c...)
2-element Array{Int64,1}:
 1
 1

```

G.3 Control Flow

We can control the flow of our programs using conditional evaluation and loops. This section provides some of the syntax used in the book.

G.3.1 Conditional Evaluation

Conditional evaluation will check the value of a Boolean expression and then evaluate the appropriate block of code. One of the most common ways to do this is with an `if` statement.

```

if x < y
    # run this if x < y
elseif x > y
    # run this if x > y
else
    # run this if x == y
end

```

We can also use the *ternary operator* with its question mark and colon syntax. It checks the Boolean expression before the question mark. If the expression evaluates to true, then it returns what comes before the colon; otherwise it returns what comes after the colon.

```

julia> f(x) = x > 0 ? x : 0;
julia> f(-10)
0
julia> f(10)
10

```

G.3.2 Loops

A *loop* allows for repeated evaluation of expressions. One type of loop is the `while` loop. It repeatedly evaluates a block of expressions until the specified condition after the `while` keyword is met. The following example sums the values in the array `x`.

```

X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while !isempty(X)
    s += pop!(X)
end

```

Another type of loop is the `for` loop. It uses the `for` keyword. The following example will also sum over the values in the array `x` but will not modify `x`.

```

X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(X)
    s += X[i]
end

```

The `=` can be substituted with `in` or `∈`. The following code block is equivalent.

```

X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for y in X
    s += y
end

```

G.3.3 Iterators

We can iterate over collections in contexts such as `for` loops and array comprehensions. To demonstrate various iterators, we will use the `collect` function, which returns an array of all items generated by an iterator:

```

 julia> X = ["feed", "sing", "ignore"];
 julia> collect(enumerate(X)) # return the count and the element
3-element Array{Tuple{Int64,String},1}:
 (1, "feed")
 (2, "sing")
 (3, "ignore")
 julia> collect(eachindex(X)) # equivalent to 1:length(X)
3-element Array{Int64,1}:
 1
 2
 3
 julia> Y = [-5, -0.5, 0];
 julia> collect(zip(X, Y)) # iterate over multiple iterators simultaneously
3-element Array{Tuple{String,Float64},1}:
 ("feed", -5.0)
 ("sing", -0.5)
 ("ignore", 0.0)
 julia> import IterTools: subsets
 julia> collect(subsets(X)) # iterate over all subsets
8-element Array{Array{String,1},1}:
 []
 ["feed"]
 ["sing"]
 ["feed", "sing"]
 ["ignore"]
 ["feed", "ignore"]
 ["sing", "ignore"]
 ["feed", "sing", "ignore"]
 julia> collect(eachindex(X)) # iterate over indices into a collection
3-element Array{Int64,1}:
 1
 2
 3
 julia> Z = [1 2; 3 4; 5 6];
 julia> import Base.Iterators: product
 julia> collect(product(X,Y)) # iterate over Cartesian product of multiple iterators
3×3 Array{Tuple{String,Float64},2}:
 ("feed", -5.0)  ("feed", -0.5)  ("feed", 0.0)
 ("sing", -5.0)  ("sing", -0.5)  ("sing", 0.0)
 ("ignore", -5.0) ("ignore", -0.5) ("ignore", 0.0)

```

G.4 Packages

A *package* is a collection of Julia code and possibly other external libraries that can be imported to provide additional functionality. This section briefly reviews a few of the key packages that we build upon. To add a registered package like `Distributions.jl`, we can run:

```
using Pkg
Pkg.add("Distributions")
```

To update packages, we use:

```
Pkg.update()
```

To use a package, we use the keyword `using`:

```
using Distributions
```

G.4.1 *LightGraphs.jl*

The `LightGraphs.jl` package allows us to represent and perform operations on graphs:

```

julia> using LightGraphs
julia> G = SimpleDiGraph(3); # create a directed graph with three nodes
julia> add_edge!(G, 1, 3); # add edge from node 1 to 3
julia> add_edge!(G, 1, 2); # add edge from node 1 to 2
julia> rem_edge!(G, 1, 3); # remove edge from node 1 to 3
julia> add_edge!(G, 2, 3); # add edge from node 2 to 3
julia> typeof(G)
LightGraphs.SimpleGraphs.SimpleDiGraph{Int64}
julia> nv(G) # number of nodes (also called vertices)
3
julia> outneighbors(G, 1) # list of outgoing neighbors for node 1
1-element Array{Int64,1}:
 2
julia> inneighbors(G, 1) # list of outgoing neighbors for node 1
Int64[]

```

G.4.2 *Distributions.jl*

The `Distributions.jl` package allows us to represent, fit, and sample from probability distributions:

```

julia> using Distributions
julia>  $\mu$ ,  $\sigma$  = 5.0, 2.5;
julia> dist = Normal( $\mu$ ,  $\sigma$ )           # create a normal distribution
Distributions.Normal{Float64}( $\mu$ =5.0,  $\sigma$ =2.5)
julia> rand(dist)                       # sample from the distribution
6.206977054108972
julia> data = rand(dist, 3)             # generate three samples
3-element Array{Float64,1}:
 3.424768821591547
 6.150186327254484
 5.914356539254751
julia> data = rand(dist, 1000);        # generate many samples
julia> Distributions.fit(Normal, data) # fit a normal distribution to the samples
Distributions.Normal{Float64}( $\mu$ =4.811767171070171,  $\sigma$ =2.5335815274297446)
julia>  $\mu$  = [1.0, 2.0];
julia>  $\Sigma$  = [1.0 0.5; 0.5 2.0];
julia> dist = MvNormal( $\mu$ ,  $\Sigma$ )      # create a multivariate normal distribution
FullNormal(
dim: 2
 $\mu$ : [1.0, 2.0]
 $\Sigma$ : [1.0 0.5; 0.5 2.0]
)
julia> rand(dist, 3)                   # generate three samples
2×3 Array{Float64,2}:
-0.224508  0.126245  2.18295
 0.671917  2.48189   3.09906
julia> dist = Dirichlet(ones(3))       # create a Dirichlet distribution Dir(1,1,1)
Distributions.Dirichlet{Float64}(alpha=[1.0, 1.0, 1.0])
julia> rand(dist)                       # sample from the distribution
3-element Array{Float64,1}:
 0.6098782851001804
 0.33255496811859186
 0.05756674678122777

```

G.4.3 JuMP.jl

The JuMP.jl package allows us to specify optimization problems and then apply a variety of different solvers, such as those included in GLPK.jl and Ipopt.jl:

```

julia> using JuMP
julia> using GLPK
julia> model = Model(GLPK.Optimizer)    # create model and use GLPK as solver
A JuMP Model
Feasibility problem with:

```

```

Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
julia> @variable(model, x[1:3])           # define variables x[1], x[2], and x[3]
3-element Array{JuMP.VariableRef,1}:
 x[1]
 x[2]
 x[3]
julia> @objective(model, Max, sum(x) - x[2]) # define maximization objective
x[1] + 0 x[2] + x[3]
julia> @constraint(model, x[1] + x[2] <= 3) # add constraint
x[1] + x[2] ≤ 3.0
julia> @constraint(model, x[2] + x[3] <= 2) # add another constraint
x[2] + x[3] ≤ 2.0
julia> @constraint(model, x[2] >= 0)       # add another constraint
x[2] ≥ 0.0
julia> optimize!(model)                   # solve
julia> value.(x)                           # extract optimal values for elements in x
3-element Array{Float64,1}:
 3.0
 0.0
 2.0

```

G.5 Convenience Functions

There are a few functions that allow us to more compactly specify the algorithms in the body of this book. Julia 1.7 will support a two-argument version of `findmax`, where we can pass in a function and a collection. It returns the maximum of the function when evaluated on the elements of the collection along with the first minimizing element. The `argmax` function is similar, but it only returns the first minimizing element. To support this in Julia 1.5, we manually extend these functions.

```

function Base.findmax(f::Function, xs)
    f_max = -Inf
    x_max = first(xs)
    for x in xs
        v = f(x)
        if v > f_max
            f_max, x_max = v, x
        end
    end
end

```

```

    end
    return f_max, x_max
end

```

```
Base.argmax(f::Function, xs) = findmax(f, xs)[2]
```

```
julia> findmax(x->x^2, [0, -10, 3])
(100, -10)
```

```
julia> argmax(abs, [0, -10, 3])
-10
```

These functions are useful when working with dictionaries and named tuples:

```
Base.Dict{Symbol,V}(a::NamedTuple) where V =
    Dict{Symbol,V}(n=>v for (n,v) in zip(keys(a), values(a)))
Base.convert(::Type{Dict{Symbol,V}}, a::NamedTuple) where V =
    Dict{Symbol,V}(a)
Base.isequal(a::Dict{Symbol,<:Any}, nt::NamedTuple) =
    length(a) == length(nt) &&
    all(a[n] == v for (n,v) in zip(keys(nt), values(nt)))

```

```
julia> a = Dict{Symbol,Integer}((a=1, b=2, c=3))
```

```
Dict{Symbol,Integer} with 3 entries:
```

```

:a => 1
:b => 2
:c => 3

```

```
julia> isequal(a, (a=1, b=2, c=3))
```

```
true
```

```
julia> isequal(a, (a=1, c=3, b=2))
```

```
true
```

```
julia> Dict{Dict{Symbol,Integer},Float64}((a=1, b=1)=>0.2, (a=1, b=2)=>0.8)
```

```
Dict{Dict{Symbol,Integer},Float64} with 2 entries:
```

```

Dict{Symbol,Integer}(:a=>1,:b=>1) => 0.2
Dict{Symbol,Integer}(:a=>1,:b=>2) => 0.8

```