Appendix A
Julia Primer

The purpose of this appendix is to give the reader a basic introduction to the Julia programming language. Julia's style and syntax are similar to MATLAB, R, and Python. As such, Julia provides the same ease of use and flexibility of these *interpreted* languages. On the other hand, Julia is a *compiled* language, making it almost as fast as C and Fortran.

A.1 Getting Started

Julia can be installed from

https://julialang.org/.

Here you will find full documentation, examples, tools, and more.

Julia comes with an interactive command-line executable, called the REPL (read-eval-print-loop), which allows for a line-by-line evaluation of Julia statements. Simply click the Julia executable or type `julia` from a system command line. For example, entering the following statement in the REPL:

```
print("Hello World!")
```

produces the output:

```
Hello World!
```

Or we can use the REPL as a calculator (note that # is used to comment the code):

J. C. C. Chan, D. P. Kroese, *Statistical Modeling and Computation*, Springer Texts in Statistics, https://doi.org/10.1007/978-1-0716-4132-3

```
x = 1.234;  # the semicolon suppresses output
y = sin(x)*sqrt(x^2)/x
```

```
0.9438182093746337
```

Julia uses the modern software paradigm where a *base* (i.e., built-in) library of code can be supplemented by loading additional *packages*. For example, the sine function `sin` is part of the in-built library of functions.

To use a package, two steps need to be taken. First, the package needs to be installed. Second, to use an installed package in a Julia program, the package needs to be loaded. The first step only has to be performed once, as Julia will remember which packages have been installed at any time. The second step needs to be repeated for every program that wants to use the particular package. Installing packages can be carried out via Julia's package manager, as in

```
import Pkg
Pkg.add("NameOfPackage")
```

Table A.1 lists a number of useful Julia packages, some of which are already built into the base library.

Table A.1 A few useful Julia packages.

`Plots`	Main plotting library
`LinearAlgebra`	Built-in library for linear algebra
`IJulia`	Package to interface with Jupyter notebooks
`Random`	Built-in library for random number generation
`Distributions`	Collection of probability distributions
`Statistics`	Built-in statistics library
`StatsBase`	Basic functionality for statistics
`DelimitedFiles`	Reading and writing delimited files
`Downloads`	Provides download functionality
`NaNStatistics`	Fast statistic with missing data
`FFTW`	Fast Fourier transforms
`Optim`	Optimization package
`SparseArrays`	Functionality for sparse arrays

You can check which additional packages have been installed with

```
import Pkg
Pkg.status()
```

Once a package has been installed, it can be included in the code by preceding the package name with `using` or `import`. For example, the following code uses the built-in random generator to generate a billion random numbers

and stores them in a vector `x`. The macro `@time` reports the run time as well as the memory storage.

```
using Random
@time x = rand(10^9);
```

```
1.603207 seconds (2 allocations: 7.451 GiB, 15.27% gc time)
```

A Julia program or *script* is a collection of statements that can be run by the julia executable. Its file extension is `.jl`. A Julia script `mycode.jl`, say, can be executed in various ways. One way is to run the Julia executable in a system shell, as in

```
julia mycode.jl
```

It is then important that either Julia is started in the correct working directory or that the path to the file is specified completely. Another way is to execute the file from within the REPL, via

```
include("mycode.jl")
```

But the most convenient way to develop and execute Julia programs is to use an integrated development environment such as Visual Studio Code (VSCode), which can be downloaded from

https://code.visualstudio.com/

After installing the Julia Language Support extension, VSCode will be able to read and execute Julia programs. The extension also comes with a debugger. Apart from the main window for the code, the IDE displays the workspace directory, the REPL window, the system shell, and a plot window. To execute a region with one or more lines in the code, one can highlight the region with the mouse and then press Shift-Enter.

Try out the statements in the following Julia file, either by executing them in the REPL or in VSCode. Running the program via `julia first.jl` in a system shell will provide no output, other than the output from the `println` and `replace` functions.

first.jl

```
i = 1       # assignment
println("i has type ", typeof(i))  # type of i
i, j = 2, 8 # assignment via a tuple
k = j/i     # division of (in this case) two integers
typeof(k)   # the result, k, is a float!
div(i,j)    # integer division
```

```
i % j       # remainder of integer division
s = "Hello. How are you"
typeof(s)   # String
replace(s, "e" => "a", count = 1)  # string replacement

u = [1, 2, 3]  # vector assignment
typeof(u)      # the vector has integer-type elements
w = u .* u     # elementwise multiplication
x = u + w      # adding two vectors of the same dimension
y = 100 .+ u   # constant plus vector. The . is essential!
sin.(u)        # elementwise computation of sine function
umat = [1 2 3] # 1x3 matrix is not a vector!
umatT = umat'  # transpose is a 3x1 matrix, not the same as u

A = [1 2 3; 4 5 6; 7 8 9]  # 3x3 matrix
u[2]           # second element of u
A[2,3]         # element of A in row 2, column 3
v = A*u        # matrix multiplication
w = u'*A       # premultiply the matrix A with the transpose
   of u
A^2            # square of matrix A
A.^2           # matrix of elementwise squares
```

A.2 Variables and Their Types

Each *variable* in Julia is a name associated with a *value*, and each value has a *type*. To find the type of a variable `x`, use `typeof(x)`. The statement `sizeof(x)` returns the *size* of the value of `x`; that is the size in bytes of the object in computer memory to which `x` refers.

For example, the statement `x = 1` creates a integer variable `x`, whose value is 1, with type `Int64`. Its size is 8 bytes in memory. Similarly, the statement `x = 1.0` creates a float variable `x`, whose value is 1.0, with type `Float64`. Its size is also 8 bytes in memory. These are the default numerical types (on a 64-bit = 8 bytes) computer.

Types can be *abstract* or *concrete* and form a hierarchy. You can find the supertypes of a type with `supertypes` and the subtypes with `subtypes`. At the top of all types is the abstract type `Any`. The number hierarchy is headed by the abstract type `Number` and lower down the hierarchy are concrete types such as `Int64` and `Float64`.

For each of the statements below, verify the type and size of the variables.

typex.jl

```
x = [1, 2, 3]      # same as x = Vector{Int64}([1,2,3])
typeof(x)          # 3-element Vector{Int64}
y = Vector{Float64}([1,2,3])
typeof(y)          # 3-element Vector{Float64}
A = [1 2 3]        # Note the absence of commas!
typeof(A)          # 1x3 Matrix{Int64}
b = Vector{Bool}([0,1,1,0])  # 4-element Vector{Bool}
sizeof(b)          # 4 bytes
notb = .~ b        # elementwise NOT operation
typeof(notb)       # 4-element BitVector
tobe = b .| notb   # elementwise OR operation
sizeof(tobe)       # 8 bytes
```

Julia is a *strongly-typed* language, meaning that there are firm restrictions on mixing different types within a statement. For example, a $1 \times n$ matrix is not the same as a vector of length n.

```
x = [1,2,3]      # a vector of Int64
A = [1 2 3]      # a 1x3 matrix of Int64
z = x + A        # gives an error
```

```
DimensionMismatch: dimensions must match
```

However, when applying mathematical operations such as $+$ or $*$, the operands are as a rule converted to a common type. For example, adding a `Int64` variable to a `Float64` results in a `Float64` variable. In the `typex.jl` program above, although `b` and `notb` have different types, we can still perform the elementwise OR operation, as the `VectorBool` object is converted to a `BitVector` object. We can convert the latter it into a `VectorBool` object via the `collect` function:

```
x = [0.2, 0.6, 0.3, 0.7] . < 0.5  # elementwise comparison
typeof(x)     # 4-element BitVector
y = collect(x)
typeof(x)     # 4-element Vector{Bool}
```

Direct *conversion* between two types can be effected by the function `convert`. Below is an example that converts a binary vector in `Int64` to a vector of `Bool`, thus reducing the size of binary vector by a factor of 8.

```
x = [1,0,0,1] # 4-element Vector{Int64}
sizeof(x)     # 32 bytes
```

```
bx = convert(Vector{Bool},x)
sizeof(bx)    # 4 bytes
```

Composite data types can be created via the Julia structures, consisting of a collection of field names with (optionally) their types. Here is an example of a `mutable struct` object:

```
mutable struct Person
  name :: String
  age :: Int
  height :: Float64
end
```

A default way to initialize a struct is to specify the values of the field names.

```
p1 = Person("Josh", 39, 1.76)
p2 = Person("Dirk", 60, 1.84)
```

Some functions will return a struct as their output, so it is important to understand how to access the field values. This is done via the dot notation, as is usual in Python and many other languages. For a mutable struct, the field values can not only be read but also be modified. Removing the `mutable` qualifier in the struct definition gives an immutable struct; attempting to change the field values will give an error message. The function `fieldnames` gives the names of the struct.

```
fieldnames(Person)
println(p1.name)    # print the name of person 1
p2.age = 61         # change the age of person 2
println(p2.age)
```

```
(:name, :age, :height)
Josh
61
```

A.3 Vectors, Matrices, and Arrays

Statistical computation often involves the manipulation of vectors and matrices. Julia's syntax for matrix computation is very similar to MATLAB's. In Julia vectors and matrix are special cases of arrays. A vector is a one-dimensional array and a matrix a two-dimensional array. For example, to create a vector $\boldsymbol{a}$, enter in the REPL or editor:

```
a = [1, 2, 3]
```

The REPL returns:

```
3-element Vector{Int64}:
 1
 2
 3
```

Similarly,

```
A = [1 2 5;  3 4 7; 6 7 9]     # no commas!
```

creates a 3×3 matrix $\mathbf{A}$. It is worth noting that Julia is case sensitive for variable names and built-in functions. That means Julia treats $\boldsymbol{a}$ and $\mathbf{A}$ as different objects. To display the i-th element in a vector $\boldsymbol{x}$, just type `x[i]`. For example,

```
a[2]
```

refers to the second element of $\boldsymbol{a}$. Similarly, one can access a particular element of $\mathbf{A}$ by specifying its row and column number (row first followed by column). For instance,

```
A[2,3]
```

displays the $(2, 3)$-entry of the matrix $\mathbf{A}$. To display multiple elements in the matrix, one can use expressions involving colons. For example,

```
A[1,1:2]
```

displays the first and second elements in the first row, whereas

```
A[:,2]
```

displays all the elements in the second column. The elements of a matrix can be stacked into a single vector as follows:

```
v = A[:]
print(v')
```

```
[1 3 6 2 4 7 5 7 9]
```

To perform numerical computation, one needs some basic matrix operations. In Julia, the following matrix operations, among several others, are available:

+ addition	/ right division
− subtraction	^ power
∗ multiplication	′ transpose
\ left division	

For example,

```
a'
```

```
1x3 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  3
```

returns the transpose of $\boldsymbol{a}$, whereas

```
a'*A
```

```
1x3 adjoint(::Vector{Int64}) with eltype Int64:
 30 36 42
```

gives the product of $\boldsymbol{a}'$ and $\mathbf{A}$.

Other operations are obvious, except for the matrix divisions \ and /. If $\mathbf{A}$ is an invertible square matrix and $\boldsymbol{a}$ is a compatible vector, then $\boldsymbol{x} = \mathbf{A}\backslash\boldsymbol{a}$ is the solution of $\mathbf{A}\,\boldsymbol{x} = \boldsymbol{a}$ and $\boldsymbol{x} = \boldsymbol{a}'/\mathbf{A}$ is the solution of $\boldsymbol{x}\,\mathbf{A} = \boldsymbol{a}'$. In other words, $\mathbf{A}\backslash\boldsymbol{a}$ gives the same result (in principle) as $\mathbf{A}^{-1}\,\boldsymbol{a}$, though they compute their results in different ways. Specifically, the former solves the linear system $\mathbf{A}\,\boldsymbol{x} = \boldsymbol{a}$ for $\boldsymbol{x}$ by Gaussian elimination, whereas the latter first computes the inverse $\mathbf{A}^{-1}$ and then multiplies it by $\boldsymbol{a}$. As such, the second method is in general slower as computing the inverse of a matrix is time-consuming (and inaccurate).

When using `LinearAlgebra`, Julia reserves the letter `I` for a "generic" identity matrix object of type `UniformScaling`. The advantage is that this object has negligible memory requirements. For example, the following computes the inverse of the above matrix $\mathbf{A}$.

```
I/A
```

```
 -6.5    8.5  -3.0
  7.5  -10.5   4.0
 -1.5    2.5  -1.0
```

It is important to note that although addition and subtraction are element-wise operations, the other operations listed above are not—they are matrix operations. For example, $\mathbf{A}^2$ gives the square of the matrix $\mathbf{A}$, not a matrix whose entries are the squares of those in $\mathbf{A}$. One can make the operations $*$, \, /, and ^ to operate element-wise by preceding them by a full stop. For example, the following returns the square of the matrix $\mathbf{A}$:

```
A^2
```

```
3x3 Matrix{Int64}:
 37   45   64
 57   71  106
 81  103  160
```

On the other hand:

```
A.^2
```

```
3x3 Matrix{Int64}:
  1   4  25
  9  16  49
 36  49  81
```

computes the squares element-wise.

Vectors and matrices are special cases of Julia *arrays*. The following creates an $3 \times 4 \times 2$ array $\mathbf{A}$ that is filled with zeros; these are by default of type `Float64`. The functions `typeof`, `eltype`, `ndims`, `size`, and `length` provide various properties of an array.

```
A = zeros(3,4,2)
typeof(A)    # type of the array
eltype(A)    # type of the elements in the array
ndims(A)     # number of dimensions
size(A)      # dimensions
length(A)    # number of elements
```

```
3x4x2 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

Array{Float64, 3}
Float64
```

```
3
(3, 4, 2)
24
```

Arrays can be accessed in the same way as vectors and matrices, e.g., `A[2,1,1]` is the (2,1,1)th element of $\mathbf{A}$, and *slice* operations such as `A[:,:,2]` can also be used. A vector is a one-dimensional array, and a matrix is a two-dimensional array.

Vectors, matrices, and arrays can be added only if they have the same dimensions. However, it possible to add a smaller array to a larger one by the process of *broadcasting*, which involves elementwise duplication of the array elements across the smaller dimension to match the larger dimension. The `.+` operator indicates that addition is carried out via broadcasting. The function **`reshape`** can be used to reshape an array into an array with different dimensions. Finally, vectors and matrices can be horizontally and vertically concatenated via the **`hcat`** and **`vcat`** functions. Here are a few examples.

```
A = [1 2; 3 4];    # matrix (suppress output)
v = [10,20];       # vector (suppress output)
B = v .+ A         # adding the vector to the columns of A
C = 1000 .+ A      # adding a constant to all elements of A
D = hcat(C,B)
E = vcat(B,C,D')
F = reshape(E,4,4)
```

```
2x2 Matrix{Int64}:
 11  12
 23  24

2x2 Matrix{Int64}:
 1001  1002
 1003  1004

2x4 Matrix{Int64}:
 1001  1002  11  12
 1003  1004  23  24

2x8 adjoint(::Matrix{Int64}) with eltype Int64:
 11  23  1001  1003  1001  1002  11  12
 12  24  1002  1004  1003  1004  23  24

4x4 reshape(adjoint(::Matrix{Int64}), 4, 4) with eltype Int64:
 11  1001  1001  11
 12  1002  1003  23
 23  1003  1002  12
 24  1004  1004  24
```

An array can have elements of different types. For example, the following vector has three types of elements.

```
x = ["string",  1, 1.0]
typeof(x[1])
typeof(x[2])
typeof(x[3])
```

```
3-element Vector{Any}:
  "string"
 1
 1.0

 String
 Int64
 Float64
```

The common type of these elements is the abstract type `Any`. Finally, note that vectors are different to *tuples* (indicated by round brackets) in that tuples are *immutable*; that is, they cannot be changed.

```
x = ("string", 1, 1.0)    # same as x = "string", 1, 1.0
typeof(x)
x[1] = "hello"
```

```
("string", 1, 1.0)
Tuple{String, Int64, Float64}
MethodError: no method matching setindex!(::Tuple{Int64, ... , ::Int64)
```

A.4 Functions

Functions make it easier to divide a complex program into simpler parts. To create a function Julia, the following syntax can be used:

```
function <function name>(<parameter_list>)
  <statements>
  return <value>  # this may be omitted
end
```

A shorter way is:

```
<function name>(<parameter_list>) = <expression>
```

An expression is any statement that gives a value when executed, such as in `sin(x) + x^2`. Thus,

```
f(x) = x^2 + 5*x - 10
```

creates the function with the name `f`, whose value at x is $f(x) = x^2 + 5x - 10$. An alternative way to define `f` is

```
f = x -> x^2 + 5*x - 10
```

The function name gives a means of invoking the function. The following evaluates the function for an integer, float, and integer vector argument.

```
f(10)
f(10.)
f([1,2,3])
```

```
140
140.0
MethodError: no method matching ^(::Vector{Int64}, ::Int64)
```

Note that the function does not know how to evaluate the square of a vector. We could remedy this by defining the function as

```
f(x) = x.^2 .+ 5*x .- 10   # note the three dots!
```

This will take vector and matrix arguments. However, a more elegant approach is to enforce elementwise operations on the function, i.e., broadcasting, by using a dot (.) after the function name:

```
f.([1,2,3])      # vector argument
f.([1 2; 3 4])   # matrix argument
```

```
-4
 4
14
-4  4
14 26
```

If the type of function arguments is important, this can be specified in the function definition, using the `::T` syntax, where `T` is the type. For example:

```
f(x::Integer) = x^2 + 5*x - 10
f(10)   # 140
f(10.)  # MethodError: no method matching f(::Float64)
```

In fact, one of Julia's strengths is the *multiple dispatch* mechanism, which allows many different versions of the same function to be defined for different types of arguments, similar to function overloading in Python.

Function names can be passed to other functions as inputs. To create a function that takes more than one input is just as easy. For example, the following code takes a column vector of data and computes its mean and standard deviation:

```
function stat(x)
  n = length(x);
  meanx = sum(x)/n;
  stdevx = sqrt(sum(x.^2)/n - meanx.^2);
  return meanx, stdevx
end

meanx,  stdx = stat(randn(100))
```

```
(0.038727242782939215, 1.119568688040557)
```

A function does not make a copy of the values of the names in the parameter list, but only assigns (binds) new names to these values. This means that functions can change the value of input arguments! In most cases, we do *not* want to change the input argument(s) to a function. When a function makes changes to the input, it is common to use an exclamation mark (!) at the end of the function name, as a warning sign. Here is an example:

```
function change2ndto100!(x)
   x[2] = 100
end

y = [1 2 3];
change2ndto100!(y)
print(y)
```

```
[1 100 3]
```

Julia has many in-built functions, and by *using* packages many more functions become available. Because of the multiple dispatch mechanism, there may be many different versions, or *methods*, of the same function. For example, without loading any additional packages, the **rand** function has (currently) 81 different methods.

```
rand
```

```
rand (generic function with 81 methods)
```

One can learn more about a specific function, say, **rand**, by typing `? rand` in the REPL. Here are some useful matrix-building functions. The functions **diag** and **diagm** are part of the `LinearAlgebra` package.

`zeros`	create a matrix of zeros
`ones`	create a matrix of ones
`diagm`	create a diagonal matrix from a vector
`diag`	extract the diagonal vector from a matrix
`rand`	generate $\mathcal{U}(0,1)$ random variables
`randn`	generate $\mathcal{N}(0,1)$ random variables

Some other useful vector and matrix functions are given below. Note that `exp`, `sqrt`, `sin`, `cos`, and `log` applied to a *matrix* will yield the corresponding matrix function, not the element-wise function. For example, `exp(A)` returns the matrix

$$\mathrm{e}^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

To obtain the elementwise operations for these functions, remember to use broadcasting, e.g., `exp.(A)`. The functions `det` and `cholesky` require the `LinearAlgebra` package.

`exp`	exponential	`log`	natural log
`sqrt`	square root	`abs`	absolute value
`sin`	sine	`cos`	cosine
`sum`	sum	`prod`	product
`maximum`	maximum	`minimum`	minimum
`cholesky`	Cholesky factorization	`inv`	inverse
`det`	determinant	`size`	dimensions

If $\boldsymbol{x}$ is a vector, `sum(x)` returns the sum of the elements in $\boldsymbol{x}$. Likewise, for a matrix $\mathbf{X}$, `sum(X)` returns the sum of all elements. For an $m \times n$ matrix $\mathbf{X}$, to obtain the $1 \times n$ matrix consisting of sums of each column, use `sum(X, dims=1)` while `sum(X,dims=2)` returns the $m \times 1$ matrix of sums of each row. For example:

```
A = [1 2 5; 3 4 7]
sum(A)
sum(A, dims=1)
sum(A, dims=2)
```

```
2x3 Matrix{Int64}:
 1  2  5
 3  4  7

22

1x3 Matrix{Int64}:
 4  6  12

2x1 Matrix{Int64}:
  8
 14
```

The function `sum` is an example of a function that has a *keyword argument* (in this case, `dims`). Many plotting functions have such keyword arguments. In general, keyword arguments can be defined via a semicolon in the argument list. For example, the following function `square` has a keyword argument `elw` which is set to `true` by default. The function returns the square of a matrix, unless the `elw` argument is set to `false`, in which case the matrix of elementwise squared values is returned. The function also illustrates the use of a *conditional expression*, as in the C language:

```
<condition> ? <expression1> : <expression2>
```

```
function square(A; elw = true)
  elw ? A^2 : A.^2      # conditional expression
end
X = [1 2; 3 4]
square(X)
square(X, elw = false)  # ; instead of , is allowed
```

For a positive definite matrix $\mathbf{A}$, `cholesky(A).L` returns the (lower) Cholesky matrix $\mathbf{B}$ such that $\mathbf{B}\mathbf{B}^\top = \mathbf{A}$. Note that `cholesky` returns a `struct` object, which has to be accessed via the dot notation. For example,

```
using LinearAlgebra
B = [2 0 0; 3 4 0; 5 1 2]
A = B*B';
cholesky(A).L    # the L field contains the Cholesky matrix
```

returns the lower Cholesky factor of $\mathbf{B}\mathbf{B}^\top$, which is, of course, $\mathbf{B}$. For some statistical applications, the current Cholesky implementation gives an error message for matrices that are ill-conditioned but are nevertheless positive definite, e.g., covariance matrices with some very small diagonal elements. For such matrices, the Hermitian nature of the matrix can be enforced via the function `Hermitian`, as in `cholesky(Hermitian(A)).L`. Of course, this stopgap solution should be changed in newer implementations of the `cholesky` function. Examples are given throughout the book, as in Chaps. 3, 8, 10, 12, and 13.

A.5 Flow Control

Julia has the usual control flow statements such as `if-then-else`, `while`, and `for`. For instance, the general form of a simple `if` statement is

```
if <condition1>
   <statements>
elseif <condition2>
   <statements>
else
   <statements>
end
```

Here, `<condition1>` and `<condition2>` are logical conditions that are either `true` or `false`; logical conditions often involve comparison operators (such as `==, >, <=, !=`). In general, there can be more than one `elseif` part, or it can be omitted. The `else` part can also be omitted. For example, the following code simulate rolling a four-sided die:

```
u = rand();
if u <= .25
    print('1');
elseif u <= .5
    print('2');
elseif u <= .75
    print('3');
else
    print('4');
end
```

The `while` loop has the following syntax.

```
while <condition>
   <statements>
end
```

To illustrate the `while` loop syntax, suppose we wish to generate a positive normal random variable (with pdf given in (2.26)). We can do that using the following `while` loop, wrapped in a function.

☞ 56

```
function posrand()
  u = randn();
  while u <= 0
    u = randn();
  end
  return u
end
```

Unlike a `while` loop, the `for` loop executes the statements for a fixed number of times. The `for` loop has the following syntax.

```
for <variable> in <collection>
   <statements>
end
```

Above, `<collection>` is any *iterable* object; that is, an object over which can be iterated. Typically this is a "range" object, such as `start:step:end`, specified by starting value, an optional step size, and an end value. One can also use the function `range` to create range objects. Vectors are natural iterable objects, but note that, in contrast to MATLAB, a range such as `1:10` is not equal to the vector $[1, \ldots, 10]$. For one thing, it takes up hardly any computer memory, as only the start and stop values need to be stored. The following shows three equivalent ways to create the same iterable object.

```
r1 = range(start = 0, stop = 1, length =101)
r2 = 0.0:0.01:1.0
r3 = range(start = 0, step = 0.01, stop = 1)
r1 == r2 == r3  # true
```

As an example, the following code generates five draws from the positive normal distribution.

```
x = zeros(5);    # create a storage vector
for i in 1:5     # can also write i=1:5
  x[i] = posrandn()
end
print(round.(x, digits=4)) # print x, rounded to 4 digits
```

```
[0.6404, 0.0033, 0.2557, 1.2712, 2.3754]
```

For further control in `for` and `while` loops, one can use a `break` statement to exit the current loop, and the `continue` statement to continue with the next iteration of the loop, while abandoning any remaining statements in the current iteration.

Similar to Python, Julia has a *list comprehension* syntax:

```
<expression> for <element> in <collection> if <condition>
```

This allows arrays to be constructed via embedded `for` loops. For example, the following produces the vector of squares of the odd numbers from 1 to 10.

```
x = [i^2 for i=1:10 if isodd(i)]
```

When performing loops, speed is important. Consider, for example, the simulation of a billion uniform random numbers, which we want to store in a vector $\boldsymbol{x}$. The following code is one way to fill the vector $\boldsymbol{x}$. Recall that

the macro `@time` can be used for timing. In this case we need to wrap the statements inside a begin-end block.

```
@time begin
  x = zeros(10^9)
  for i in 1:10^9
    x[i] = rand()
  end
end
```

```
28.560135 seconds (2.00 G allocations: 37.253 GiB, 1.75% gc time)
```

We see that the computation takes a long time. However, due to the way Julia's compiler works, it is better to put the loop inside a function. In this case, we refill the vector $\boldsymbol{x}$ by changing its entries one by one.

```
function fillrand!(x)
  for i in 1:10^9
     x[i] = rand()
  end
end

@time fillrand!(x)
```

```
1.410237 seconds (9.13 k allocations: 578.083 KiB, 0.89% compilation time)
```

Now it only takes a bit more than 1 second! Of course it is much cleaner (easier to read) if we create the vector in one go via the `rand` function, giving a similar performance:

```
@time x = rand(10^9)
```

```
1.324148 seconds (2 allocations: 7.451 GiB, 0.66% gc time)
```

A.6 Graphics

Julia has several "back-end" plot facilities. The default module is `GR`, which can be accessed via the `Plots` module. It allows users to create various graphical objects including two- and three-dimensional graphs. One can also have a title on a graph, add a legend, change the font and font size, label the axis, etc., by changing the corresponding attributes. A list of plot attributes may be obtained via `plotattr()`. See also

https://docs.juliaplots.org

In Julia the most basic function used to create 2D graphs is `plot`. For example, to make a graph of $y = \sin(x)$ on the interval from $x = 0$ to $x = 2\pi$, we use the following code, which also shows various plotting attributes, how to use LaTeX strings, and how to save a plot as a pdf file.

```
using Plots, LaTeXStrings
x = 0:0.01:2*pi
p1 = plot(x,sin.(x), # the . is important! Naming the plot p1
    tickfontsize = 15,              # axis font size
    guidefontsize = 20,             # label font size
    legend = false,                 # legend is on by default
    grid = false,                   # grid is on by default
    linewidth = 3,                 # linewidth is 1 by default
    tickfont = "Computer Modern",  # axis font
    color = "salmon"                # line color
  )
xlabel!(L"x")                       # using LaTeX font
ylabel!(L"\sin(x)")
savefig(p1,"sin.pdf")               # saving the figure
```

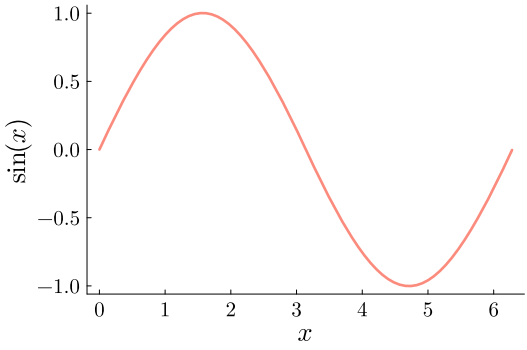

Fig. A.1 A plot of the graph $y = \sin(x)$ from 0 to 2π

The graph produced is given in Fig. A.1. Note that the command `x = 0:.01:2*pi;` creates a iterable grid object of type `StepRangeLen` that ranges from 0 to 2π in steps of 0.01. It is important to note that this is not a vector. As mentioned in the previous section, the function `range` can also be used to create range objects, as in `range(start=0, stop=2*pi, length=100)`.

Another useful function is `histogram`, which allows us to plot histograms. The following Julia script creates standard normal data of size 10000 and makes a histogram with 50 bins. Instead of a histogram, it is often more useful to have a density estimate. The code below uses fast and optimal **theta KDE**

of Botev et al. (2010). The corresponding Julia module `.ThetaKDE`, Plots which contains the function `kde`, can be downloaded from the book's website. Note that `plot!` is used to plot the kde and the histogram in the same figure (see Fig. A.2).

```
using .ThetaKDE, Plots
data = randn(10000)
histogram(data,bins=50, normalize = true, legend=false)
mindat = minimum(data); maxdat = maximum(data);
h,density,xmesh = kde(data,2^14,mindat,maxdat)
plot!(xmesh[1:2:end],density[1:2:end],linewidth=3)
```

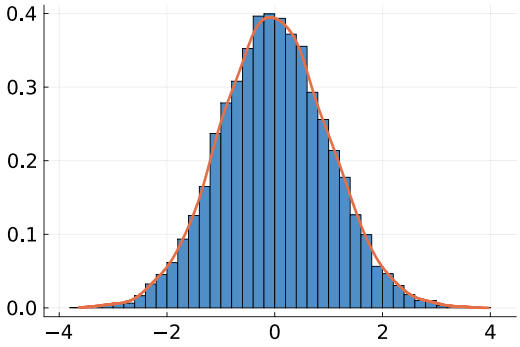

Fig. A.2 A histogram of 10000 standard normal draws and its kernel density estimate

It is often desirable to plot several graphs in the same figure window. For this purpose we can use the `layout` attribute of the `plot` function.

Suppose we wish to make scatterplots of data from the two-dimensional standard normal and uniform distributions in the same figure window (see Fig. A.3). This is accomplished in the code below. The plot attribute `aspectratio` ensures that the x and y scaling is equal. The value of the aspect ratio variable is in this case `:equal`, which is a Julia *symbol*—a unique identifier. The attributes `ms, msw, ma` control the size, linewidth, and alpha value (i.e., transparency) of the marks, respectively. In this case the layout `(1,2)` indicates that the figures are to be plotted next to each other. Finally, `size` determines the size of the plot window.

```
x = randn(1000,2)   # 2D standard normal data
y = rand(1000,2)    # 2D standard uniform data
p3 = scatter(x[:,1],x[:,2], aspectratio = :equal, ms = 5,
     msw = 0, ma = 0.3,legend = false)
p4 = scatter(y[:,1],y[:,2], aspectratio = :equal, ms = 5,
```

```
    msw = 0, ma= 0.3, legend = false)
p34 = plot(p3,p4,layout = (1,2), size = (600,200))
```

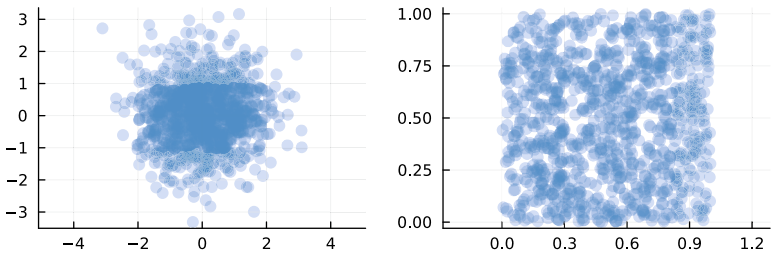

Fig. A.3 Scatterplots for the two-dimensional standard normal and standard uniform distributions

In addition, one can also easily produce 3D graphical objects in Julia. To illustrate various useful routines, suppose we want to plot the density function of the bivariate normal distribution (see Sect. 3.6) given by ☞ 83

$$f(x,y;\varrho) = \frac{1}{2\pi\sqrt{1-\varrho^2}}\, \mathrm{e}^{-\frac{1}{2(1-\varrho^2)}(x^2-2\varrho xy+y^2)} .$$

As in plotting a 2D graph, we first need to build a grid for x and y, which can be done with the function **range**. For example, we use the following code to plot the bivariate normal density function with $\varrho = 0.8$ in Fig. A.4.

```
using Plots
rho = 0.8
x = range(-3, stop=3, length=100)
y = range(-3, stop=3, length=100)
f(x,y) = 1/(2*pi*sqrt(1-rho^2))*exp(-(x^2 -2*rho*x*y + y^2)
   /(2*(1-rho^2)))
plt = surface(x, y, f,   # surface broadcasts f by default
legend=false,
camera = (75,40)    # azimuth and elevation angles
)
```

By adding the following code, we can even produce an animation that gradually changes the viewing angle

```
anim = @animate for i in 0:180
    plot!(plt, camera = (i, 40))
```

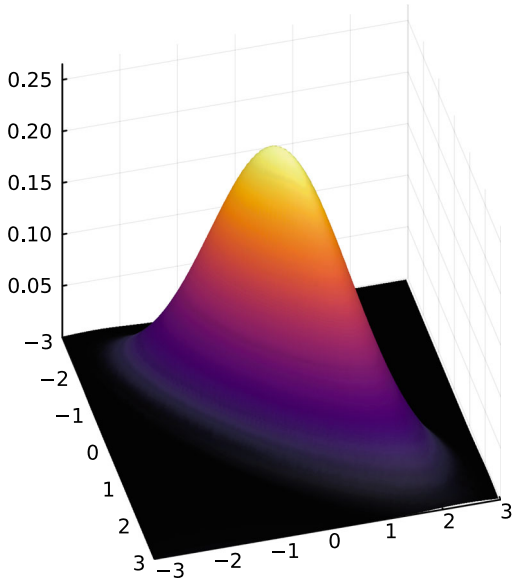

Fig. A.4 The density function of the bivariate normal distribution with $\varrho = 0.6$

```
end
gif(anim, "animsurf.gif", fps = 15)
```

Also, a contour plot can be obtained by using the function **contour**:

```
contour(x,y,f);
```

The result is shown in Fig. A.5.

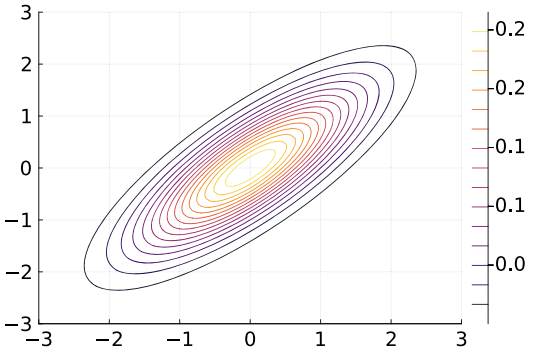

Fig. A.5 A contour plot of the bivariate normal density function with $\varrho = 0.6$

A.7 Optimization Routines

Julia provides various ways to optimize functions. In this section we discuss some of them that are used in the main text. Note that, typically, optimization routines are framed in terms of minimization. In order to perform maximization, some minor changes to the objective function are required. More precisely, suppose we want to maximize the function $f(\boldsymbol{x})$ and find a maximizer $\boldsymbol{x}_{\max} = \operatorname{argmax}_{\boldsymbol{x}} f(\boldsymbol{x})$. Instead of the original maximization problem, consider minimizing $-f(\boldsymbol{x})$ and noting that

$$\boldsymbol{x}_{\max} = \operatorname*{argmax}_{\boldsymbol{x}} f(\boldsymbol{x}) = \operatorname*{argmin}_{\boldsymbol{x}} -f(\boldsymbol{x}) \,.$$

Hence, without loss of generality, we will focus on minimization routines. One basic minimization function is `optimize` from the optimization package `Optim`. To illustrate its usage, suppose we wish to minimize the function $f(x) = \sin(x^2)$ over the interval $[0, 3]$ (see Fig. A.6).

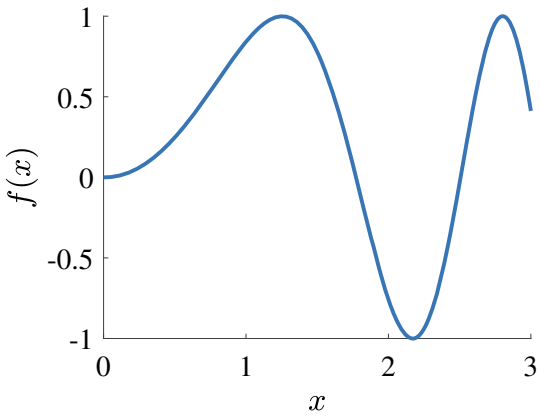

Fig. A.6 A plot of $f(x) = \sin(x^2)$ from 0 to 3

We can define the function in Julia as follows:

```
f(x) =  sin(x^2);
```

or also as

```
f =  x -> sin(x^2);
```

To ensure that we only consider arguments in $[0, 3]$, we could set any function value outside the interval to a very large value, via

```
f(x) = x <= 3 && x >= 0 ? sin(x^2) : 1E50
```

For scalar arguments, `optimize` takes three inputs: the function name and lower and upper bounds of the interval. The minimizer and minimum (i.e., minimum value of the function evaluated at the minimizer) can be found as attributes of the object returned by the function, as illustrated below.

```
using Optim
f(x) = x < 3 && x > 0 ? sin(x^2) : 1E50
res = optimize(f,0,3)
println("minimum = ", res.minimum, ";  minimizer = ",
res.minimizer)
```

```
minimum = -1.000000;  minimizer = 2.170804
```

The function `optimize` can also be used to minimize multivariate functions. However, care should be taken with the choice of the starting point for the algorithm. As an example, suppose we wish to minimize the *peaks* function (from MATLAB)

$$S(\boldsymbol{x}) = 3\,(1-x_1)^2\,\mathrm{e}^{-x_1^2-(x_2+1)^2} - 10\,(\frac{x_1}{5} - x_1^3 - x_2^5)\,\mathrm{e}^{-x_1^2-x_2^2} - \frac{1}{3}\,\mathrm{e}^{-(x_1+1)^2-x_2^2}, \quad [x_1, x_2] \in \mathbb{R}^2\,,$$

with respect to $\boldsymbol{x} = [x_1, x_2]$. A contour plot is given in Fig. A.7.

```
function S(x)
 3*(1-x[1])^2*exp(-x[1]^2 - (x[2]+1)^2) - 10*(x[1]/5-x[1]^3
 - x[2]^5)*exp(-x[1]^2-x[2]^2) - 1/3*exp(-(x[1]+1)^2-x[2]^2)
end

x0 = [0.0,0.0];  # starting point
res = optimize(S,x0);
println("minimum = ", res.minimum, ";  minimizer = ",
res.minimizer)
```

```
minimum = -0.064936;  minimizer = [0.296431, 0.320161]
```

This, however, turns out to yield a *local* minimum, rather than a *global* one. If we instead take the starting point `x0 = [0.1,-1]`, we obtain the global minimizer and minimum:

```
minimum = -6.551133;  minimizer = [0.228261, -1.625537]
```

A simple but powerful alternative is to use the *cross-entropy (CE)* method of Rubinstein and Kroese (2004). This is a global optimization function that uses repeated sampling combined with parameter updating, instead of gradient information. Below is a basic implementation. The function makes use of the packages `LinearAlgebra` and `Statistics`.

```
function CEmin(f, mu, sigma, N, Nel, tol)
# minimize function f via the CE method
n = length(mu)  # dimension
  while maximum(sigma) > tol
    ds = n == 1 ? sigma : diagm(sigma)  # scalars or vectors?
    X = randn(N,n)*ds .+ mu'  # N rows of n-dim normals
    fX = n==1 ? f.(X) : f.(eachrow(X));  # Function values
    # sort the samples by their function values
    sortfX = sortslices(hcat(X, fX), dims=1, by = x -> x[n
       +1])
    Elite = sortfX[1:Nel, 1:n]; # smallest (= elite) samples
    # update mu and sigma
    mu = n == 1 ? mean(Elite) : vec(mean(Elite,dims=1))
    sigma = n == 1 ? std(Elite) : vec(std(Elite,dims=1))
  end
  return f(mu), mu   # minimum,  minimizer
end
```

To use the function, specify the starting vector (or value for scalar arguments), a vector of standard deviations (initially chosen large enough to sample points from a wide region), the number of samples at each iteration, the number of elite (i.e., best) samples, and a tolerance for stopping.

```
using LinearAlgebra, Statistics
mu = [0,0]; sigma = 4.0*ones(2);
N = 1000; Nel = 100; tol = 1E-5;
minS, mu = CEmin(S,mu,sigma,N,Nel,tol);
dig = convert(Int64,-log10(tol))
println("minimum = ",minS, digits = dig),
"  minimizer = ", round.(mu,digits = dig))
```

```
minimum = -6.551130  minimizer = [0.228280, -1.625530]
```

Figure A.7 illustrates that the correct minimizer for this multimodal function is found in a few iterations.

Indeed, the same `CEmin` program can be used to minimize the function `f` above.

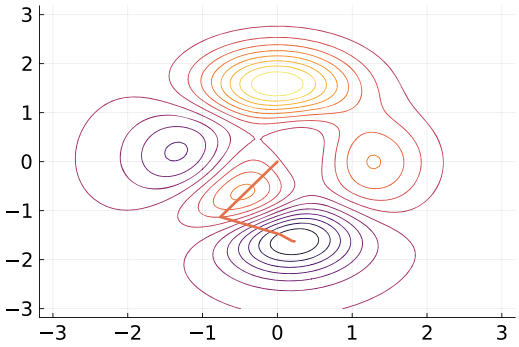

Fig. A.7 Contour plot with the CE minimization path, starting from the origin

```
f(x) = x < 3 && x > 0 ? sin(x^2) : 1E50
minf, mu = CEmin(f,1.0,1.0,100,10,1E-8);
dig = convert(Int64,-log10(tol))
println("minimum = ",round(minf,digits = dig),
"  minimizer = ", round.(mu,digits = dig))
```

```
minimum = -1.000000  minimizer = 2.170804
```

A.8 Handling Sparse Matrices

A **sparse matrix** is simply a matrix that contains a large proportion of zeros. Computation for sparse matrices can typically be done much faster than for full matrices. In addition, as most of the elements in a sparse matrix are zeros, the storage cost of a sparse matrix is also small. In statistics we often need to deal with large sparse matrices. Thus it is useful to learn how to employ them in Julia.

The package `SparseArrays` is necessary for sparse matrix and vector operations, and `LinearAlgebra` is usually also required. A basic function for creating sparse matrices is `sparse`. For example, suppose the matrix

$$\mathbf{W} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 1 \end{bmatrix}$$

is stored as a full matrix in Julia. The `sparse` function converts a full matrix to sparse form by squeezing out any zero elements.

```
using SparseArrays
W = [1 0 0 0 0 ;
     0 1 0 0 0
     0 0 2 0 0
     0 0 0 3 1]
S = sparse(W)
```

```
4x5 SparseMatrixCSC{Int64, Int64} with 5 stored entries:
 1  .  .  .  .
 .  1  .  .  .
 .  .  2  .  .
 .  .  .  3  1
```

Notice that only the non-zero elements in **W** are stored. To find the indices and values of the nonzero elements, the function **findnz** can be used, which returns a three-tuple of vectors, where the first two vectors identify the indices and the third vector the values. The function **nnz** returns the number of nonzeros of a sparse array.

```
a = findnz(S)
hcat(a[1], a[2], a[3])
nnz(S)
```

```
([1, 2, 3, 4, 4], [1, 2, 3, 4, 5], [1, 1, 2, 3, 1])

1  1  1
2  2  1
3  3  2
4  4  3
4  5  1

5
```

In general, we can create a sparse matrix **S** by the command

```
S = sparse(i,j,s,m,n)
```

This uses vectors $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{s}$ to generate an $m \times n$ sparse matrix such that $S(\boldsymbol{i}(k), \boldsymbol{j}(k)) = \boldsymbol{s}(k)$. For example, to create the matrix **W** above, we first need to build a vector $\boldsymbol{s}$ that stores all the non-zero elements:

```
s = [1, 1, 2, 3, 1];
```

Next, we create a vector $\boldsymbol{i}$ that stores the row position for each element in $\boldsymbol{s}$. For example, the first element in $\boldsymbol{s}$ should be in the first row, the second element in second row, and so on. We then do the same thing for the column positions and store them in the vector $\boldsymbol{j}$:

```
i = [1, 2, 3, 4, 4]
j = [1, 2, 3, 4, 5]
```

To create the 4×5 matrix $\mathbf{W}$ above, write

```
W = sparse(i,j,s,4,5)
```

The function `Array` converts a sparse matrix back to dense form. When using the `LinearAlgebra` package, the command `sparse(I, 100, 100)` creates an `n`×`n` sparse identity matrix. We can accomplish the same goal via

```
sparse(1:100, 1:100, ones(n))
```

Another useful function is `spdiagm`, the sparse version of `diagm`, which can be used to create sparse diagonal matrices. To extract the (sparse) main diagonal from a sparse matrix, use `Diagonal`; this returns a sparse vector. Notice the syntax of creating diagonal elements below and above the main diagonal.

```
spdiagm(-1 => 1:99, 1 => 1:99) # 100x100 parse matrix with
  # entries on the principal sub and sup diagonals
s = Diagonal(S)  # yields a sparse vector
spdiagm(ones(10))  # 10x10 sparse identity matrix
```

As mentioned earlier, one main advantage of working with sparse rather than full matrices is that computations involving sparse matrices are usually much quicker. For instance, some methods to simulate a Gaussian random process on a grid of 400×400 pixels, as in Fig. A.8, require a Cholesky decomposition of a 160000×160000 matrix, which is impossible to store in CPU memory. However, if each row of the matrix only contains a few nonzero entries, then it is very feasible to compute the Cholesky decomposition quickly; see also Kroese et al. (2011, Section 5.1).

Finally, it should be noted that, currently, the sparse Cholesky method in Julia differs from the ordinary (full-matrix) Cholesky method, in that the sparse method first permutes the rows and columns of the original matrix for efficient storage and retrieval. In particular, if $\mathbf{A}$ is the sparse matrix of interest, Julia determines the Cholesky matrix $\mathbf{L}$ such that

$$\mathbf{L}\mathbf{L}^\top = \mathbf{P}\mathbf{A}\mathbf{P}^\top,$$

for some *permutation matrix* $\mathbf{P}$—a matrix of 0s and 1s with exactly one 1 in each column and row. Note that such a matrix is *orthogonal*; that is, $\mathbf{P}^\top = \mathbf{P}^{-1}$. Hence, defining $\mathbf{B} = \mathbf{P}^\top \mathbf{L}$, we have the matrix decomposition

$$\mathbf{B}\mathbf{B}^\top = \mathbf{A}.$$

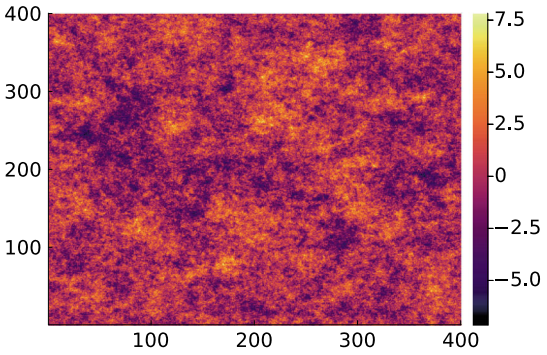

Fig. A.8 A Gaussian spatial process on a 400 × 400 grid

However, the matrix **B** is no longer lower-diagonal! Here is a worked example:

```
a = [1, 2, 3, 1, 2, 4, 1, 3, 4, 2, 3, 4]
b = [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]
c = [1.0, -0.25, -0.25, -0.25, 1.0, -0.25,
     -0.25, 1.0, -0.25, -0.25, -0.25, 1.0]
A = sparse(a,b,c)
R = cholesky(A); # calculate the (sparse) Cholesky matrix
P = sparse(1:4,R.p,ones(4)) # permutation matrix
B = P'*sparse(R.L)  # matrix with B*B' = A
isapprox(B*B', A)  # true
```

A.9 Distributions

We have already encountered the functions `rand` and `randn` from the base package to generate uniform and standard normal random variables. The packages `Distributions` and `Random` offer a wide of additional facilities for probability distributions and random variable simulation. Table A.2 lists the names of some common distributions available in `Distributions`. See Sects. 2.5 and 2.6 for various properties of these distributions.

The following illustrates how these distribution types can be used.

```
using Random, Distributions, Plots
Random.seed!(1234)   # set the random seed (optional)
dist = Poisson(5)    # Poisson distribution
mean(dist)           # the expectation for this distribution
var(dist)            # the variance for this distribution
x = rand(dist,10000) # an iid sample of size 10000
```

Table A.2 Names of common distributions in the `Distributions` package

Name	parameters
`Bernoulli`	p
`Binomial`	n, p
`DiscreteUniform`	a, b
`Geometric`	p
`Poisson`	λ

Name	parameters
`Beta`	α, β
`Chisq`	n
`Exponential`	$\theta = 1/\lambda$
`FDist`	m, n
`Gamma`	$\alpha, \theta = 1/\lambda$
`Normal`	μ, σ
`TDist`	n
`Uniform`	a, b

```
mean(x)              # the sample mean
var(x)               # the sample variance
plot(pdf.(dist,0:20), linetype = :scatter,
   line = :stem, marker= :circle) # a plot of the pdf

gammalist = [Gamma(i,4) for i in [0.5,1,2,4]]
                     # 4 different Gamma distributions
mean.(gammalist)     # lists of expectations

xmesh = 0:0.01:20
pdfs = [ pdf.(dist, xmesh) for dist in gammalist ];
plot(xmesh, pdfs, ylims=[0,0.5], linewidth=2)
```

Other useful functions are **`cdf`**, **`quantile`**, **`std`**, and **`median`**. Note that the latter three can be used to compute the *exact* quantile, standard deviation, and median of a distribution, as well as calculating *approximations* thereof via their sample equivalents.

```
dist = TDist(4);
cdf(dist,3.0)
quantile(dist,0.95)  # exact 95% quantile
std(dist)            # exact standard deviation
median(dist)         # exact median

x = rand(TDist(4),100);
quantile(x,0.95)     # sample 95% quantile
std(x)               # sample standard deviation
median(x)            # sample median
```

```
0.9800290159641406
2.1318467863266495
1.4142135623730951
0.0
```

```
2.4844351992270557
1.3846972965065583
-0.07485935730166166
```

A.10 Input/Output

Julia treats input and output as a *stream*: a sequence of data, with a program adding data to one end of the stream and a device taking data from the other end. The devices can either be input devices (e.g., a keyboard or a file) or output devices (e.g., a screen or a file).

The following program writes the prime numbers in the set $\{1, \ldots, 100\}$ into the file `primes.txt`. Note that this file is a *binary* file, as the numbers are written in binary form. To create a human-readable *text* file, the number needs to be written to the file as a string. You can try this out by uncommenting the corresponding lines below.

```
using Primes    # import if not already done so
io = open("primes.txt", "w"); # open the stream for writing
for i in 1:100
  if isprime(i) # is the number prime?
  write(io,i)   # write an Int64 object to the file
# write(io,string(i)*"\n") # write a string + newline
# println(io,i) # same as above
  end
end
close(io) # always remember to close the file
```

To read the file thus created, we basically just reverse the stream, making sure the correct data type is read.

```
io = open("primes.txt","r")  # open the file for reading
while !eof(io)  # while not the end of the file
   n = read(io,Int64)  # read an Int64 variable
#  n = read(io,String) # read a String variable
   println(n)
end
close(io)
```

To write and read CSV (comma separated values) file, one can use the package `DelimitedFiles`. Here is an example:

```
using DelimitedFiles
y = [0.1, 0.2]
X = [1 2 3; 4 5 6]
A = hcat(y,X)
io = open("mydata.csv","w")
writedlm(io,A, ',')  # write A, comma separated
close(io)

A1 = readdlm("mydata.csv",',')  # read A back
A == A1 # true
y1 = A[:,1]
X1 = A[:,2:end]
```

Finally, the following illustrates some dictionary and string operations on a large text file. A dictionary is a data structure that stores (key,value) pairs in an efficient way—via a hash-table, similar to an old-fashioned telephone book. The output of the script is a list of words consisting of at least five letters that appear at least 250 times in the text file.

```
io = open("ataleof2cities.txt")
d = Dict()  # create a new dictionary
for line in readlines(io)
   words = lowercase.(split(strip(line)))
   for w in words
      w = replace.(w, ['.', ',',';'] => "") # ignore punct.
      if !haskey(d,w) # is the word already in the dictionary
         ?
         d[w] = 1  # if not, add it
      else
         d[w] +=1
      end
   end
end
close(io)
sortd = sort(collect(d),by=last,rev=true)
for w in sortd
   if length(w[1]) >= 5 && w[2] >=250
      println(w[1],"    ",w[2])
   end
end
```

```
there    499
which    388
would    335
lorry    320
```

```
their    317
could    280
defarge   265
little   263
```

A.11 Other Aspects of the Language and Caveats

The previous sections cover most of the elements of the Julia language that are relevant for this book. However, there are many more language aspects to discover for the interested reader.

One thing we have not yet discussed is the *scoping* of variables, i.e., the way in which variable names are known or unknown within different regions of the code. Like in most other languages, functions have their own namespace. That is, any variable defined inside the function is not accessible outside the function. In general, code in Julia is organized in *modules*—regions of code that have their own namespace, and every time a module or package is loaded, a new namespace is created. The default modules are `Main, Core`, and `Base`. The function `varinfo` gives a summary of all the variables in a module. For example, to find the variables in the scope of the REPL, type `varinfo(Main)`. Likewise, the many variables and functions in the base module can be viewed with `varinfo(Base)`. As we have no need for user-defined modules in this book, we will say no more about this topic. Another feature of Julia that is out of the scope of this book is *metaprogramming*, i.e., writing a program that modifies a Julia program. The only encounter we will have with metaprogramming is via macros such as `@time` that measure the running time of a function or block of code.

We next list a number of caveats of which the reader should be aware, especially if they are familiar with MATLAB. Some of the issues have already been discussed in earlier sections, but it is prudent to emphasize them.

- Like most other computing languages, Julia uses square brackets `[ ]` to access arrays, in contrast to MATLAB, which uses parentheses `( )`.
- In Julia, the type of a variable matters, much more than is the case in MATLAB. Nevertheless, variables of different types can often be combined in a natural way. Consider, for example, the following code, where the variables `x`, `y`, and `z` refer to different types of objects.

```
x = 1:3             # range object 1:3
y = [1.0,2.0,3.0]   # 3-element vector of Float64
z = [1 2 3]'        # 3x1 matrix of Int64
x + z               # 3x1 matrix of Int64
x + 1               # ERROR
```

```
x + y              # 3-element vector of Float64
x .+ 1             # range object 2:4
x/x                # 3x3 matrix of Float64
x./x               # 3-element vector of 1.0s
A = [1 2 3 ; 4 5 6]# 2x3 matrix of Int64
A*x                # 2-element vector of Int64
x + y              # 3-element vector of Float64
A*z                # 2x1 matrix of Int64
```

- Elementwise operations on arrays in Julia are generally carried out via broadcasting, and this needs to be explicitly specified via the dot operator. For example:

```
using Plots
x = 1:0.1:3        # range object
y = sin.(x) .- 1   # 21-element Float64 vector
plot(x,y)          # plotting y against x
```

- It is important to realize that assigning a new name to an existing object does not create another instance of that object. Consider, for example,

```
x = [1 2 3 4];     # x refers to a matrix object
y = x;             # y refers to the SAME matrix object
z = copy(x);       # z refers to a NEW matrix object
y[2] = 0;          # same as x[2] = 0
z[2] = 0;          # now the new object is changed
println(x - y)     # x and y still refer to the same
println(x - z)
```

```
[0 0 0 0]
[0 2 0 0]
```

  In contrast, in MATLAB the assignment `y = x` will automatically create a new copy of the object to which `x` refers.
- A main difference with the scoping in MATLAB is that `for` and `while` loops introduce their own *local* scope. For example, the following common construction in MATLAB gives a warning and error message in Julia:

```
a = 0;
for i=1:10
   a = a + 1
end
```

```
Warning: Assignment to `a` in soft scope is ambiguous ...
ERROR: UndefVarError: `a` not defined
```

Instead we need to let Julia know that `a` is a global variable.

```
a = 0;
for i=1:10
  global a = a + 1
end
```

Another, rather bothersome, issue is that it is not possible to reset or clear various variables from the workspace. The easiest way to "clear" the workspace is to restart/delete the REPL.

- Punctuation in Julia is applied in many different ways. For example, a semicolon (`;`) at the end of a statement is used to suppress output, but a semicolon in an argument list of a function indicates a keyword argument. A colon (`:`) in front of a name indicates a *symbol*. For example if `f` refers to a function (i.e., a numerical recipe that maps input to output), `:f` refers to the symbol/letter `f` that represents this recipe. This is similar to the Lisp `quote` syntax which returns an expression without evaluating it. The different rules for punctuation are summarized in https://docs.julialang.org/en/v1/base/punctuation/
- Because Julia uses *just in time compilation*, first-time compilation or the "using" of a package or module can be slow. Subsequent running of the (now compiled) code will be much faster.
- As Julia is still in development (currently version 1.11.1), various scientific computing applications are not as well developed as in more mature computing platforms. For example, the plotting routines in Julia are still inferior to MATLAB's, especially for 3D plotting. Also the optimization packages have limited functionality.
- Although Julia has corrected various quirks of other languages (such as the use of parentheses to access matrices in MATLAB, and the strange R syntax for vector/matrix operations), it has itself introduced some idiosyncrasies, which perhaps may disappear in later versions. An example is the local scope within `for` loops and the required use of the **`global`** qualifier for certain global variables within the loop. We also mentioned the different results that the **`cholesky`** method yields when applied to sparse and dense matrices and use of the function **`Hermitian`** to force the method to accept certain positive definite matrices without throwing a (false) error message. Here are some more unexpected results:

```
"hello" * "hello"  # * for string concatenation,  not +
"hello"^2
10*10^6            # Int64
1e6                # Float64
60^20              # gives a negative integer
80^60              # results in 0
```

```
max(1,2,3)          # maximum of 3 arguments
x = [1,2,3]
max(x)              # ERROR
maximum(x)          # different function name required
```

```
"hellohello"
"hellohello"
10000000
1.0e7
-6450068360557232128
0
3
ERROR
3
```

A.12 Further Reading and References

Recent books that use Julia for statistics and decision-making include Nazarathy and Klok (2021), Chan (2021), and Kochenderfer et al. (2022). Another useful resource for learning Julia is

https://juliateachingctu.github.io/Julia-for-Optimization-and-Learning/stable/

Finally, all programs and (large) data files in this book may be downloaded from the homepage

https://people.smp.uq.edu.au/DirkKroese/statbook/

To accommodate the users of MATLAB and R, we have mirrored each Julia program with its equivalent in MATLAB and R.