

Chapter 2

An Introduction to R

Exploration is our mission; we and those who use our software want to find new paths to understand the data and the underlying processes.

John M. Chambers, Software for Data Analysis (2008, p. 3).

Abstract This chapter introduces R, a dialect of the S language, which was developed at Bell Laboratories. R's inventor Dr. John Chambers was awarded the 1998 Association of Computing Machinery Software award. In its citation, the ACM noted that *S will forever alter the way people analyze, visualize, and manipulate data*. R's mission is to enable the best and most thorough exploration of data possible. R is open-source software (GNU General Public License), and has statistical, data manipulation, and visualization libraries. R is a functional programming language, where software programs are organized into *functions* that can be invoked to transform data. This chapter describes key R elements, including vectors, lists, matrices, data frames and functions. It concludes by presenting a system dynamics model of customer growth, which is implemented using the **deSolve** open source package. Appendix A summarizes the installation process for R, and the reader is recommended to work through this chapter using the R Studio console, so that the short examples can be executed.

Keywords Vectors • Functions • Matrices • Data frames • deSolve

Vectors

The fundamental data type in R is the vector, which is a variable that contains a sequence of elements that have the same data type (Matloff 2009). A vector is defined by the ability to index its elements by position, in order to extract or replace a subset of data (Chamber 2008). The vector object is similar to a one-dimensional array structure in a programming language such as C or Java. Vectors can be created in the following manner.

```
v1<-c(1,2,3,4,5)
```

This creates a vector variable **v1** and assigns it an initial value using the function **c**, which is the combine function in R. By typing **v1** at the console, the vector's values can be inspected.

```
> v1  
[1] 1 2 3 4 5
```

The printed value **[1]** at the beginning of the output is a useful piece of information that displays the starting index for that particular printed row of vector data. The concept of an index is important in R, as it allows access to individual elements of a vector, using the square brackets notation. In R the index for a vector starts at 1. This command displays the third element of the vector **v1**.

```
> v1[3]  
[1] 3
```

In R, variable types can include integer, numeric, character, and logical types. The mode of a variable can be examined using the **typeof(x)** function call. In a vector, the mode of each element is the same.

```
> typeof(v1)  
[1] "double"
```

Functions can operate on vectors, for example, to find the length, maximum value, and minimum value, the functions **length(x)**, **max(x)** and **min(x)** are used. Each of these R functions returns a vector of length 1.

```
> length(v1)  
[1] 5  
  
> max(v1)  
[1] 5  
  
> min(v1)  
[1] 1
```

A powerful feature of R is that it supports *vectorization*, where functions can operate on every element of a vector, and return the results of each individual operation in a new vector. Many in-built R functions support vectorization, including the square root function **sqrt(x)**.

```

> v1
[1] 1 2 3 4 5

> r<-sqrt(v1)

> r
[1] 1.000000 1.414214 1.732051 2.000000 2.236068

```

A significant benefit of this feature is that the analyst does not have to write a loop to iterate through the vector. Vectorized functions have the general form of *vector in, vector out* (Matloff 2009), where the size of the output vector mirrors the size of the input vector.

Arithmetic operations can also be applied to vectors in an element-wise manner. For this example, the vector **v1** is multiplied by the constant 3, and the result (**v2**) is then added to **v1**, and finally stored in **v3**.

```

> v1
[1] 1 2 3 4 5
> v2<-3*v1
> v2
[1] 3 6 9 12 15
> v3<-v1+v2
> v3
[1] 4 8 12 16 20

```

When operations are applied to two vectors that requires them to be of equal length, R automatically recycles the shorter vector until it is of sufficient length to match the longer one.

```

> v4<-c(10,20)
> v1
[1] 1 2 3 4 5
> v5<-v1+v4
Warning message:
In v1 + v4 :
  longer object length is not a multiple of shorter object
length
> v5
[1] 11 22 13 24 15

```

Conditional expressions can also be applied to vectors, and these are used to filtering vector data. For example, by taking the original vector **v1** and applying a conditional expression to that vector, R will return a logical vector (e.g. a vector whose elements are either TRUE or FALSE) containing the results for each

conditional expression evaluation. In this case, the condition tests which vector elements are even, and R's modulus operator (`%%`) is used.

```
> v1
[1] 1 2 3 4 5
> test<-v1 %% 2 == 0
> test
[1] FALSE TRUE FALSE TRUE FALSE
```

An interesting feature of R is that this logical vector can now be used as an *index* to the original vector, and those values that match to TRUE in the logical vector will be returned by the operation. Using the NOT logical operator (`!`), all the FALSE values can be returned.

```
> evens<-v1[test]
> evens
[1] 2 4
> odd<-v1[!test]
> odd
[1] 1 3 5
```

As R is a functional programming language, many operations can be cascaded together to provide a concise set of operations. Therefore, the statement for obtaining the even numbers from the vector `v1` can be written in a single line of code.

```
> evens<-v1[v1 %% 2 == 0]
> evens
[1] 2 4
```

R's **which()** function is used to find the location index of vector values. For example, to create a new vector of even numbers it is possible to first find the location the even numbers in the vector, and then use these indices to create the new vector.

```
> v1
[1] 1 2 3 4 5
> ind<-which(v1 %% 2 == 0)
> ind
[1] 2 4
> evens<-v1[ind]
> evens
[1] 2 4
```

This process can be written in a single line of code.

```
> evens<-v1[which(v1 %% 2 == 0)]
> evens
[1] 2 4
```

Indexing can also be used to extract elements from a vector, using the colon operator (:), which generates regular sequences within a specified range. These sequences can be applied to filter the original vector. A minus sign can be used to exclude a range of indices from the calculation.

```
> 2:4
[1] 2 3 4
> v1[2:4]
[1] 2 3 4
> v1[-(2:4)]
[1] 1 5
> v1[-1]
[1] 2 3 4 5
```

The function **seq()** is used to generate a sequence vector in arithmetic progression, and this will be used in the R system dynamics models to setup the simulation time. For example, the vector **times** is a sequence from 0 to 5 (inclusive).

```
> times<-seq(from=0,to=5)
> times
[1] 0 1 2 3 4 5
```

What is convenient about the **seq()** function is that it can accept an additional parameter (**by**) which can vary the distance between the different elements.

```
> times<-seq(from=0,to=5,by=.5)
> times
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Vectors can also be processed using the vectorized **ifelse(b,u,v)** function, which accepts a boolean vector **b** and allocates the element-wise results to be either **u** or **v**. For example, a new character vector can be formed with elements classified as “EVEN” or “ODD” depending on the input vector’s value.

```
> ans<-ifelse(v1%%2==0, "EVEN", "ODD")
> ans
[1] "ODD" "EVEN" "ODD" "EVEN" "ODD"
```

Two additional vectorized functions are useful. These are **all()** and **any()** which process the entire vector and report an overall single condition. It is an efficient form of carrying out a sequence of logical AND (all) or logical OR (any) tests on the vector elements.

```

> v1
[1] 1 2 3 4 5
> any(v1==1)
[1] TRUE
> any(v1<0)
[1] FALSE
> all(v1>=0)
[1] TRUE

```

The elements of a vector can also be allocated names, and in later chapters parameters in a simulation model will be identified this way. Here names are added to the original vector **v1**, and these are then displayed at the console.

```

> v1
[1] 1 2 3 4 5
> names(v1)<-c("a", "b", "c", "d", "e")
> v1
a b c d e
1 2 3 4 5

```

A useful feature of naming vector elements is that the name also provides an index to access the value.

```

> v1
a b c d e
1 2 3 4 5
> v1["c"]
c
3

```

Vectors can be increased with new elements. At an implementation level, a new variable is created in memory when a vector is added to, so some computational overhead is involved. This example shows how elements can be added to the end of a vector, using the concatenate (**c**) function.

```

> v1
[1] 1 2 3 4 5
> v1<-c(v1,c(6,7))
> v1
[1] 1 2 3 4 5 6 7

```

Elements can also be added to the start of a vector.

```

> v1
[1] 1 2 3 4 5
> v1<-c(c(-1,0),v1)
> v1
[1] -1 0 1 2 3 4 5

```

Lists

R's list structure can combine objects of different types. For example, using the **list()** function, a variable is created that can represent information on a student.

```
s<-list(id="1234567", fName="Jane", sName="Smith", age=21)
```

The list variable shows the components of the list (known as tags).

```
> s
$id
[1] "1234567"
$fName
[1] "Jane"
$sName
[1] "Smith"
$age
[1] 21
```

List elements can be accessed through the operator \$, for example.

```
> s$fName
[1] "Jane"
> s$age
[1] 21
```

Technically, a list is a vector, and elements it can also be accessed through its index, although double brackets are used instead of single ones to return a vector.

```
> s[[1]]
[1] "1234567"
> s[[2]]
[1] "Jane"
```

Also, elements can be returned using single brackets containing the name of the data type.

```
> s["fName"]
$fName
[1] "Jane"
> s["age"]
$age
[1] 21
```

New elements can be added to a list by simply adding a new element to the variable. The **str()** function can be used to view the structure of an R variable.

```
s$gender<-'F'

> str(s)
List of 5
 $ id      : chr "1234567"
 $ fName   : chr "Jane"
 $ sName    : chr "Smith"
 $ age      : num 21
 $ gender   : chr "F"
```

Elements can also be removed from a list, by setting the relevant element to `NULL`.

```
s$age<-NULL

> str(s)
List of 4
 $ id      : chr "1234567"
 $ fName    : chr "Jane"
 $ sName    : chr "Smith"
 $ gender   : chr "F"
```

The list elements can be accessed directly, using the `names()` function.

```
> names(s)
[1] "id"      "fName"   "sName"   "gender"
```

The data contained in a list can be returned as a single vector, using the `unlist()` function. Note that because the vector must contain elements of the same type, the age value is coerced into a character string.

```
> unlist(s)
      id      fName      sName      age      gender
"1234567"  "Jane"   "Smith"   "21"     "F"
```

Finally, interesting things can be done with lists. For instance, they can be recursive, which means a list can contain lists. The earlier example can be extended to do this, by adding an extra student.

```
s1<-list(id="1234567", fName="Jane", sName="Smith", age=21)
s2<-list(id="1234568", fName="Matt", sName="Johnson", age=25)
```

The two lists (representing each individual student) are added to a new list, and this list is then “a list of lists”.

```
l<-list(s1,s2)
```

The list output can be summarized as follows, which shows that each element contains a list of 4 elements.

```
> str(l)
List of 2
 $ :List of 4
  ..$ id   : chr "1234567"
  ..$ fName: chr "Jane"
  ..$ sName: chr "Smith"
  ..$ age  : num 21
 $ :List of 4
  ..$ id   : chr "1234568"
  ..$ fName: chr "Matt"
  ..$ sName: chr "Johnson"
  ..$ age  : num 25
```

Matrices

A matrix is a data structure that has a number of rows and columns, where each element has the same mode. Matrix subscripts, similar to vectors, commence at [1,1], and these are used to access row and column elements. A matrix can be initialized from a vector, where the numbers of rows and columns are specified as parameters. R stores matrices by column-major order, and by default matrices are filled in this manner. A matrix can be populated in row-major order by passing the parameter **byrow = TRUE** to the matrix function.

```
> m<-matrix(c(10,20,30,40,50,60),nrow=3,ncol=2)
> m
      [,1] [,2]
[1,]   10   40
[2,]   20   50
[3,]   30   60
```

Matrix elements can be accessed using their row and column numbers as indices.

```
> m[1,1]
[1] 10
> m[3,2]
[1] 60
```

Individual rows can be accessed in a convenient way, by removing the index for a specific column. For this, a vector of row elements is returned.

```
> m
      [,1] [,2]
[1,]   10   40
[2,]   20   50
[3,]   30   60
```

```
> m[1,]
[1] 10 40
```

Columns can be extracted by specifying the column index, and the column values are returned in a vector structure.

```
> m[,2]
[1] 40 50 60
```

The function **dim()** can be used to display the matrix dimension, and the functions **nrow()**, **ncol()** provide information on the number of rows and columns.

```
> dim(m)
[1] 3 2
> nrow(m)
[1] 3
> ncol(m)
[1] 2
```

A further useful set of matrix functions is **rowSums()** and **colSums()**, which sum all row and column elements respectively.

```
> rowSums(m)
[1] 50 70 90
> colSums(m)
[1] 60 150
```

In a similar way, the functions **rowMeans()** and **colMeans()** calculate the means of rows and columns.

```
> rowMeans(m)
[1] 25 35 45
> colMeans(m)
[1] 20 50
```

Filtering can also be performed on matrices. For example, if a query is required to find all rows that have *column 1 values greater than 20*, the following code could be used. First a logical vector could be applied to the full column with the specified condition.

```
> test<-m[,1] > 20
> test
[1] FALSE FALSE  TRUE
```

Table 2.1 Useful matrix operations in R

Operator or function	Description
$A * B$	Element-wise multiplication
A/B	Element-wise division
$A \%*\% B$	Matrix multiplication
$t(A)$	Transpose of A
$e<-eigen(A)$	List of eigenvalues and eigenvectors for matrix A

This logical vector can then be applied to the row index for the matrix to filter out all FALSE values, and in this case, return the 3rd row, which matches the condition.

```
> m[test,]
[1] 30 60
```

R matrices support linear algebra operations, and this feature will be used in the epidemiology system dynamics model of Chap. 5. Table 2.1 summarizes these operations.

Rows and columns can be added to a matrix, using **rbind()** and **cbind()**, where a vector of appropriately sized values is included as an argument.

```
> rbind(m,c(40,70))
      [,1] [,2]
[1,]   10   40
[2,]   20   50
[3,]   30   60
[4,]   40   70

> cbind(m,c(70,80,90))
      [,1] [,2] [,3]
[1,]   10   40   70
[2,]   20   50   80
[3,]   30   60   90
```

Data Frames

A data frame is similar to a matrix, as it has a two-dimensional rows and columns structure, however it differs from a matrix in that each column can have a different mode (Matloff 2009). This is convenient for data processing, as many real-world data sets consist of tables with different data types, and these can be easily replicated in data frames. For example, the student example presented earlier can be represented in a data frame, by specifying each attribute as a vector, and then combining these into a data frame. The list items were:

```
s1<-list(id="1234567", fName="Jane", sName="Smith", age=21)
s2<-list(id="1234568", fName="Matt", sName="Johnson", age=25)
l<-list(s1,s2)
```

Based on this data, we can identify four different vectors as follows.

```
ids<-c("1234567", "1234568")
fNames<-c("Jane", "Matt")
sNames<-c("Smith", "Johnson")
ages<-c(21, 25)
```

These vectors can be combined into a data frame, which represents data similar to the manner in which it is stored in a convention spreadsheet. Attributes are lined up in columns, and each individual observation is stored in a row. The flag **stringsAsFactors** is set to FALSE, which means R will not convert strings to factors, which are used to represent categorical variables in R.

```
s<-data.frame(ID=ids, FirstName=fNames, Surname=sNames,
              Age=ages, stringsAsFactors=FALSE)
```

```
> s
```

	ID	FirstName	Surname	Age
1	1234567	Jane	Smith	21
2	1234568	Matt	Johnson	25

Technically, a data frame is a list, and so the list notation can be used to access information. For example, columns can be accessed using the double bracket notation `[[]]`, and individual elements can also be extracted from columns by applying a further index to locate the value.

```
> s[[1]]
[1] "1234567" "1234568"
> s[[1]][1]
[1] "1234567"
```

A data frame can also be accessed using matrix operators, where the structure is accessed via its rows and columns.

```
> s[1,]
      ID FirstName Surname Age
1 1234567      Jane   Smith  21
> s[,1]
[1] "1234567" "1234568"
> s[1,1]
[1] "1234567"
```

Finally, data frames elements can be accessed using the column names as follows.

```
> s$Surname
[1] "Smith"    "Johnson"
```

Filtering can be performed by applying conditional statements to the data frame, for example, finding all students whose age is greater than 21.

```
> s[s$Age > 21,]
      ID FirstName Surname Age
2 1234568      Matt Johnson  25
```

This query can also be applied using the **subset()** function, which takes a data frame and applies a filtering condition.

```
> sb<-subset(s,s$Age>21)
> sb
      ID FirstName Surname Age
2 1234568      Matt Johnson  25
```

Additional columns can be conveniently added to a data frame. For example, if all students under the age of 21 were eligible for a discount, the following command would add this information as a new column in the data set.

```
> s$Discount<-ifelse(s$Age<=21,"YES","NO")
> s
      ID FirstName Surname Age Discount
1 1234567      Jane   Smith  21      YES
2 1234568      Matt Johnson  25      NO
```

For data analysis, opportunities often arise by merging different data sets, and the **merge()** function facilitates this. In the student example, a second data frame could store examination results for each student.

```
ids<-c("1234567","1234568")
subjects<-c("CT111","CT111")
grade<-c(80,80)

r<-data.frame(ID=ids,Subject=subjects,Grade=grade,
              stringsAsFactors=FALSE)
> r
      ID Subject Grade
1 1234567   CT111    80
2 1234568   CT111    80
```

As this data frame shares a common attribute with the student information (i.e. the ID value), the two data frames can be merged based on this column (passed as an argument to the merge function).

```
> new<-merge(s,r,by="ID")
> new
      ID FirstName Surname Age Subject Grade
1 1234567      Jane   Smith  21    CT111    80
2 1234568      Matt Johnson  25    CT111    80
```

The merged data frame could then be used to support statistical analysis of a large data set, for example, to test whether there is a link between factors such as age, and examination performance.

Functions

A function is a group of instructions that takes input, uses the input to compute values, and returns a result (Matloff 2009). Users of R should adopt the habit of creating simple functions which will make their work more effective and also more trustworthy (Chambers 2008). Functions are declared using the **function** reserved word. They contain a list of parameters (some of which may have default values), and execute a set of instructions between an opening brace (`{`) and a closing brace (`}`).

```
convC2F<-function(celsius)
{
  fahr<-celsius*9/5 + 32.0
  return(fahr)
}

> convC2F(100)
[1] 212
```

This initial function converts temperature in Celsius to its corresponding value in Fahrenheit. The formal parameter **celsius**, and the variable **fahr** are both local to the function, which means they are no longer available after the function completes its task. This is important, as it enables information hiding within the function, and ensures that all direct communication between functions is done through its arguments and return value. Variables declared outside of functions are global, and are visible within the functions. The second function shows how a loop structure can be used within the function in order to calculate the result—in this case to count the frequency of even numbers in a vector. Notice that the return statement is omitted, as the last evaluated expression is by default returned by functions in R, and avoiding a return statement can improve code performance.

```

evenCount<-function(v)
{
  ans<-0
  for(x in v)
  {
    if(x%%2==0)
      ans<-ans+1
  }
  ans # more efficient method for returning values
}

```

The function is tested by passing in an arbitrary vector, and observing the result.

```

> evenCount(c(2,2,1,2))
[1] 3

```

Apply Functions

Another use of user-defined functions in R is as a parameter to the *apply* family of functions, which are one of the most famous and used features of R (Matloff 2009). The general form of the **sapply(x,f,fargs)** function is as follows:

- **x** is the target vector or list
- **f** is the function to be called
- **fargs** are the optional set of arguments that can be applied to the function **f**.

The **sapply()** function takes as input a target vector and a function. The function specifies the logic that is executed on each vector element, and **sapply()** then returns a vector with the processed data. For example, *if there was a requirement to calculate the difference between each value in a vector and the overall vector mean, the following code could be used.*

First, the sample **data** is generated, with 10 random values between 1 and 10, using the function **sample()**, where replacement is enabled. The mean is calculated using the **mean()** function.

```

> data<-sample(1:10,replace=T)
> data
[1] 9 2 8 10 9 1 8 2 1 6
> mean(data)
[1] 5.6

```

This **sapply()** call to perform this task, shown below, takes three parameters:

- The vector to be iterated over, which is the vector **data**.
- The function to process each element. This function is declared within the **sapply** call itself, and takes two parameters, **e** and **m**. The parameter **e** is the current vector element being processed, and the parameter **m** is the vector mean. The function then evaluates the difference between the two values, and this is processed by **sapply** and a vector returned after all the elements have been processed.
- The third parameter maps onto the second argument (**m**) to be passed to the function, which is the mean of the vector.

```
> d<-sapply(data,function(e,m){e-m}, mean(data))
```

The resulting vector displays the difference between each element and the overall vector mean.

```
> d
[1] 3.4 -3.6 2.4 4.4 3.4 -4.6 2.4 -3.6 -4.6 0.4
```

The *apply* functions can also be used to process lists, as well as vectors. For example, consider the following list of students.

```
s1<-list(id="1234567",fName="Jane", sName="Smith", age=21)
s2<-list(id="1234567",fName="Matt", sName="Johnson", age=25)
l<-list(s1,s2)
```

The task here is to implement a simple query: find the list elements (in the list **l**) whose age is greater than 21. This can be done in two steps. First, **sapply()** is used to process the query and return a boolean vector indicating the list indices that match the conditional expression, and the result is stored in the vector **b**.

```
> b<-sapply(l,function(x)x$age>21)
> b
[1] FALSE TRUE
```

Next, the vector **b** can be used to filter the original list, and the answer is stored in the **ans**, which now contains all those elements that match the condition.

```
> ans<-l[b]
> str(ans)
List of 1
 $ :List of 4
  ..$ id   : chr "1234568"
  ..$ fName: chr "Matt"
  ..$ sName: chr "Johnson"
  ..$ age  : num 25
```

The **apply()** function can be used to process rows and columns for a matrix, and the general form of this function (Matloff 2009) is **apply(m, dimcode, f, fargs)**, where:

- **m** is the target matrix
- **dimcode** identifies whether it's a row or column target. The value 1 is used to process rows, whereas 2 applies to columns
- **f** is the function to be called
- **fargs** are the optional set of arguments that can be applied to the function **f**.

For example, **apply()** can be used to find the mean value in each row.

```
> m
      [,1] [,2]
[1,]    10    40
[2,]    20    50
[3,]    30    60
> apply(m,1,mean)
[1] 25 35 45
```

In a similar way, **apply()** can be used to find the mean value in each column.

```
> apply(m,2,mean)
[1] 20 50
```

deSolve Package

R's **deSolve** package solves initial value problems written as ordinary differential equations (ODE), differential algebraic equations (DAE), and partial differential equations (PDE) Soetaert et al. (2010). For system dynamics models, the ODE solver in **deSolve** is used. The key requirement is that system dynamics modelers implement the model equations in a function, and this function is called by **deSolve**. For this example the customer growth model from Chap. 1 is revisited, as shown in Fig. 2.1.

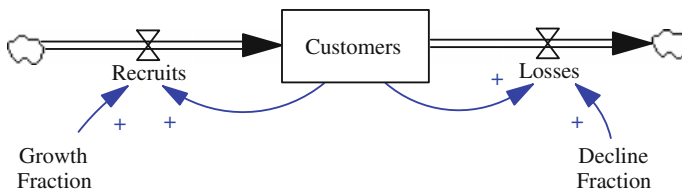


Fig. 2.1 A stock and flow model of customers (from Chap. 1)

The R implementation of this model is now described. To use the **deSolve** library, the package needs to be installed, and then it should be referenced in the model source file by calling the **library()** function.

```
library(deSolve)
```

In the R implementation, the first task is to define the simulation time constants, and then create the simulation time vector using the **seq()** function.

```
START<-2015; FINISH<-2030; STEP<-0.25
simtime <- seq(START, FINISH, by=STEP)
```

The vector **simtime** can be inspected, and it is useful to see how the **seq()** function creates the list of times from start to finish, with the appropriate steps in between. The **head()** and **tail()** function are used to display the first and final six elements of the vector.

```
> head(simtime)
[1] 2015.00 2015.25 2015.50 2015.75 2016.00 2016.25
> tail(simtime)
[1] 2028.75 2029.00 2029.25 2029.50 2029.75 2030.00
```

Next, two model vectors must be defined, as these are required as inputs to the system dynamics model function. The first vector is named **stocks** and contains the model stocks, along with their initial values. For this example, there is only a single stock, and its initial value is set to 10000. To improve model readability, a computer programming convention known as Hungarian notation is used to prefix a variable name with its system dynamics type, i.e. *s* for stock, *f* for flow and *a* for auxiliary).

```
stocks <- c(sCustomers=10000)
```

The second vector is called **auxs** and this contains the exogenous parameters for the customer model.

```
auxs <- c(aGrowthFraction=0.08, aDeclineFraction=0.03)
```

When simulating with **deSolve**, the modeler must write a function to implement the model equations. The user-defined function, arbitrarily named **model()**, and called from the **deSolve** library, takes three parameters:

- The current simulation time (*time*),
- A vector of all current stock values (*stocks*).
- A vector of model parameters (*auxs*).

These vectors can be transformed to lists using **as.list()**, and embedded in the **with()** function, as this allows the variable names to be conveniently accessed.

```
model <- function(time, stocks, auxs){
  with(as.list(c(stocks, auxs)),{

    fRecruits<-sCustomers*aGrowthFraction

    fLosses<-sCustomers*aDeclineFraction

    dC_dt <- fRecruits - fLosses

    return (list(c(dC_dt),
                  Recruits=fRecruits, Losses=fLosses,
                  GF=aGrowthFraction,DF=aDeclineFraction))
  })
}
```

With these input values, all that remains is to specify the stock and flow equations in their correct solving sequence.

- The flow *fRecruits* is a product of the stock *sCustomers* and the growth fraction *aGrowthFraction*.
- The flow *fLosses* is a product of the stock *sCustomers* and the decline fraction *aDeclineFraction*.
- The net flow (derivative) for the stock is calculated as the difference in inflow and outflow, and stored in the variable *dC_dt*.

A list structure is then returned to the **deSolve** package. The first parameter is a vector of all the net flows, and this *must match the order in which the stocks are initialized* in the vector **stocks**. Following this, any other model variable can be added to the return list to ensure that appears as part of the final result set. In this case, the flows and auxiliaries are added, and user-friendly names provided.

Finally, the model is solved by calling the **ode()** function, which is part of the **deSolve** library. This function takes five arguments.

- The vector of stocks (*y=stocks*).
- The simulation time vector (*times=simtime*).
- The function name that contains the model equations (*func = model*).
- The auxiliary parameters (*parms=auxs*).

- The integration method (*method*="euler"). Other methods are available, including Runge-Kutta 4th order integration (*method*="rk4").

```
o<-data.frame(ode(y=stocks, times=simtime, func = model,
                  parms=auxs, method="euler"))
```

The full set of simulation results from **ode** are then converted into a data frame, and using R's **head()** function, the first six rows of results are displayed.

```
> head(o)
      time sCustomers Recruits   Losses   GF   DF
1 2015.00   10000.00  800.0000  300.0000  0.08  0.03
2 2015.25   10125.00  810.0000  303.7500  0.08  0.03
3 2015.50   10251.56  820.1250  307.5469  0.08  0.03
4 2015.75   10379.71  830.3766  311.3912  0.08  0.03
5 2016.00   10509.45  840.7563  315.2836  0.08  0.03
6 2016.25   10640.82  851.2657  319.2246  0.08  0.03
```

This data frame can be used as a basis to plot data and also to analyze results. For example, the **summary()** function can be applied to the stock and flows in the data frame, yielding useful summary statistics (columns 1, 5 and 6 are omitted).

```
> summary(o[, -c(1,5,6)])
      sCustomers      Recruits      Losses
Min.   :10000   Min.   : 800.0   Min.   :300.0
1st Qu.:12048   1st Qu.: 963.9   1st Qu.:361.4
Median :14516   Median :1161.3   Median :435.5
Mean   :14866   Mean   :1189.3   Mean   :446.0
3rd Qu.:17489   3rd Qu.:1399.2   3rd Qu.:524.7
Max.   :21072   Max.   :1685.7   Max.   :632.2
```

Visualization

R provides visualization libraries, and throughout this text, the R package **ggplot2** is used. The terminology used in **ggplot2** (Chang 2013) includes:

- The *data* to be visualized, which consists of variables stored in a data frame.
- The geometric objects, or *geoms*, that are drawn to represent the data, such as points and lines.
- Aesthetic attributes that are the visual properties of geoms, such as line color, point shape etc.

As the simulation results are already in a data frame, they are ready to be visualized by calls to **ggplot2**. The package **ggplot2** must be included, and following that a call to the function **ggplot()** is executed. A *layered approach* to building a plot is used, with where additional attributes are added in sequence, following the first call to **ggplot()**. With this example, a line is added specifying the data frame, and the function **aes()** specifies the *x* and *y* variables.

```
library(ggplot2)

ggplot()+
  geom_line(data=o,aes(time,o$sCustomers),colour="blue")+
  geom_point(data=o,aes(time,o$sCustomers),colour="blue")+
  scale_y_continuous(labels = comma)+
  ylab("Customers")+
  xlab("Year")
```

Figure 2.2 shows the variable of interest (*sCustomers*) changing over time. Additional variables can be added to the plot by adding further calls to **geom_line()**,

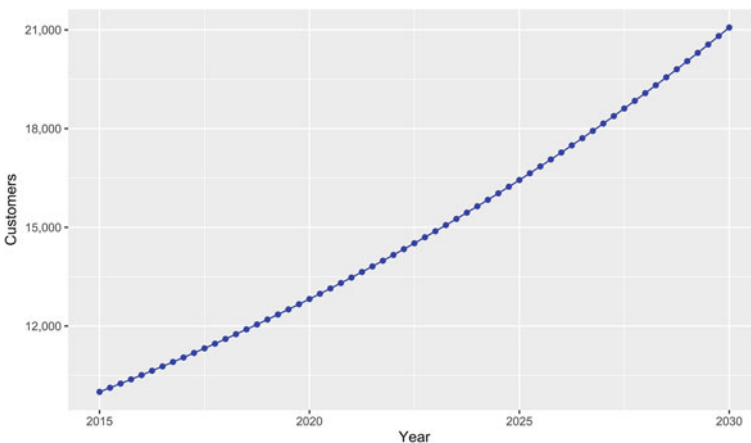


Fig. 2.2 Visualizing the output from *deSolve* for the customer model

and this data is also be presented in point format by using the function **geom_point()**. High resolution plots can also be created to support publication-quality presentations. Following the **ggplot()** call, the function **ggsave()** will save the image to a file on the disk, and this supports a range of formats.

```
ggsave("customers.png")
```

In summary, **ggplot2** is a powerful visualization framework. A more comprehensive listing of its features is outside the scope of this text, however Chang (2013) provides an excellent source of examples that can be built upon to maximize the visualization impact of simulation output.

Summary

In conclusion, R is a powerful data analytics platform that supports system dynamics modeling through the **deSolve** package. Further benefits from using R is the facility to vectorize simulation models, analyze data, and apply further statistical analysis to simulation output. For example, Chap. 5 will show how disaggregate system dynamics models can be created using R. Chapter 6 demonstrates how R's unit testing framework can be used to test models, and Chap. 7 provides examples of model calibration, sensitivity analysis, and statistical screening, that can all be used to enhance the model building process.

Exercises

1. Create a vector of 100 random numbers, in the range 1–10. From this vector, filter those variables that are divisible by 2. Finally, ensure that there are no duplicates in the resulting vector (the R function `deduplicated()` can be used to support this final operation).
2. A quadratic equation has the form $ax^2 + bx + c$. Use `sapply()` to transform an input vector in the range $[-100, +100]$ using a quadratic equation, where the parameters a , b and c are provided as additional inputs to the transformation.
3. For an input vector of 1000 uniform random numbers, find the difference of each element from the overall mean, and filter out all those resulting elements that are less than zero or equal to zero.

References

- Chambers J (2008) Software for data analysis: programming with R. Springer Science & Business Media, Chicago
- Chang W (2013) R graphics cookbook. O'Reilly Media Inc., Sebastapol, CA
- Matloff N (2009) The art of R programming. No Starch Press, San Francisco, CA
- Soetaert KER, Petzoldt T, Setzer RW (2010) Solving differential equations in R: package deSolve. J Stat Soft 33

<http://www.springer.com/978-3-319-34041-8>

System Dynamics Modeling with R

Duggan, J.

2016, XVIII, 176 p. 54 illus., 46 illus. in color., Hardcover

ISBN: 978-3-319-34041-8