R programming a course on theory and mechanics of statistical analysis

R is a statistical data manipulation scripting language for analysis. The R language was originally inspired by the S language developed by Bell Labs on behalf of AT&T.

The case of using R over other resources:

… Open sourced developed, supported, and extended
… Comparable and often superior to commercial enterprise products
… A high-level, general purpose language that is extensible and automatable
… Features of object-oriented, and functional programming languages
… Massive user community for support and growth

R leverages that features of object oriented programming. In general, this feature refers to the inputs and outputs of complex programmatic functions being assigned as objects for single uniform reference across the platform. R is also polymorphic, meaning a single function can be applied to different types of inputs (generic functions). An example of object-oriented programming features in R is illustrated below:

Consider the head( ) and format( ) functions in R:

```
head(x)   →   returns the first or last parts of a vector, matrix, table,
data frame or function.
x  →   is an object
format(x, . . .)   →   formats an R object for pretty printing
. . .   →   refers to additional arguments
head(format(x)) → returns the first part of the object class in the
assigned format
```

Not only can multiple independent functions be combined (polymorphic), but the objects can be applied to multiple functions without specific designation or restriction.

R leverages functional programming in ways like implicit iteration. Rather than being required to code loops, R's functional features allow expression of iterative behavior implicitly. This results in faster run times and lower computational costs. Code becomes more compact, executes faster, requires less debugging, and transitions to parallel programming in a simpler manner.

The Art of R programming A Tour of Statistical Software Design

Norman Matloff

The contents of this, and proceeding documentation is a comprehensive outline, or executive summary, written as cited from ***The Art of R Programming – A Tour of Statistical Software Design*** as written and published by Norman Matloff.

Norm Matloff's Biographical Sketch

Dr. Norm Matloff is a professor of computer science at the University of California at Davis, and was formerly a professor of statistics at that university. He is a former database software developer in Silicon Valley, and has been a statistical consultant for firms such as the Kaiser Permanente Health Plan. He was born and raised in the Los Angeles area, and has a PhD in pure mathematics from UCLA, specializing in probability/functional analysis and statistics.

Norm Matloff's Blog – Upon Closer Inspection

Norm Matloff's Blog – Mad (Data) Scientist

The Art of R Programming – A Tour of Statistical Software Design

R programming a course on theory and mechanics of statistical analysis

The following content listing outlines the topics covered in proceeding documentation:

functions 𝝀 in R

A function is a group of instructions that take inputs, performs computations, and returns a result.

```
functionName <- function(x) {
logical computations
}
```

The above illustrates the structure of a function with one argument, x. in formal terms, x function(x) refers to the formal argument (parameter), with the actual value input of x referred to as the actual argument.

Variables existing within functions are local variables, they exist only within the computations of the function and disappear after the result is computed. Variable assignments outside of functions are global variables and will retain their values until reassigned or removed from the environment.

Variables within functions can also be assigned default arguments that will use the defined function values if not specified by the user within the function parameters.

data structures in R

Vectors drive the core functionality of R programming. Vector elements in R must be consistent of the same data types (mode) throughout.

Scalars (individual numbers) technically do not exist in R. Scalars are treated as one-element vectors.

Character Strings are also one-element vectors with a character data type (or mode character).

Matrices in R correspond to the mathematical concept of the same nature. A Matrix is a rectangular array of numbers. Technically, a matrix is a vector with two additional attributes; 1) the number of rows; and 2) the number of columns.

Like a vector in R, a **list** in R is a container for values. The difference between vectors and lists are that lists can contain many different data types. List elements are accessed in R with the dollar sign $ syntax.

In application, a typical data set consists of multiple types of data that cannot be stored in a single matrix in R alone. Instead, **data frames** are created to store actual data sets. A data frame in R is a list, with each component of the list being a vector related to each column in the data set.

In R, objects are instances of classes. **Class instances** are R lists with the class name as an extra attribute. Class objects in R are often utilized as R's S3 classes. The S3 classes are from the S language, version 3.

vectors [ ] in R

Vectors are the fundamental data type in R; the following topics are discussed in detail:

… Recycling → automatic lengthening of vectors in certain settings
… Filtering → extracting subsets of vectors
… Vectorization → functions are applied element-wise to vectors

recycling in R

When performing an operation with two vectors that requires an equal vector length, R automatically recycles (repeats) the shorter vector until it reaches the same length as the longer vector.

common vector operations in R

With R being a functional programming language, all mathematical operators are actually functions. R executes operations element-wise on vectors and matrices. Importantly in relation to Linear Algebra, R performs the multiplicative, and other numeric operations, element-by-element.

Indexing vectors occurs when a subvector is formed by selecting elements of the given vector for specific indices. The format is **`vector1[vector2]`**; the result is a selection of elements from **`vector1`** whose indices are provided by **`vector2`**. As stated previously, R indexes vectors by 1, and not 0, like other languages.

The colon **`:`** operator creates a range of numbers that is useful for indexing vectors and functions. It is important to understand the operator precedence constraints in R (order of operations). For example, the colon **`:`** operator has a higher precedence than the subtraction **`–`** operator, but less precedence than the parenthesis **`()`** operator.

The sequence **`seq()`** function is a generalization of the colon **`:`** operator, which generates a sequence in arithmetic progression. Unlike the colon **`:`** operator, the sequence **`seq()`** function can specify units apart.

The sequence **`seq()`** function is additionally useful in loops, where the **`length()`** function falls short in loops where the argument is null, and thus should have zero iterations of the loop.

The repeat **`rep()`** function allows the placement of the same constant in vectors; creates a vector of size **`x`** times the copies of **`x`**. Note the use of the **`each =`** argument having nonexpected behavior in **`rep()`**. The **`any()`** and **`all()`** functions are convenient to examine the logical outputs of any given argument.

vectorized operations in R

Operations in R are vectorized when a function is applied individually to each element in a vector. Simple to complex operators can be vectorized in functions; inclusive of transcendental functions—square roots, logs, trig functions, etc.—are all vectorized. Formally speaking, scalars are one-element vectors in R and arithmetic operators (+) are functions within themselves.

```
> y <- c(3, 6, 9)
> y + 4                 #the numeric 4 will be added element-wise to vector 'y'
[1]  7 10 13

> '+'(y, 4)       #'+' as a function of vector 'y' with input 4 yields the same results
[1]  7 10 13
```

Note the recycling took part in the above function as ‘4’ was recycled into matching vector length (4,4,4). Vectorization should be accounted for when writing functions and code in R for ‘code ethics and safety’.

NA and NULL values Y in R

"No such Animal" values are accounted for specifically in R as either NA or NULL values. Missing data in statistical samples is denoted as NA; implying that the data is simply missing from the dataset. Conversely, the NULL assignment of a value in a dataset implies that the example value simply *does not exist*; rather than being existent but unknown.

```
> x <- c(3, NA, 9)        #3-element vector with the middle value set = NA
> x
[1]  3 NA  9
> mean(x)                 #R accounts for the NA value in the function mean()
[1] NA
> mean(x, na.rm = T)      #argument to remove NA values from the vector input
[1] 6
> x <- c(3, NULL, 9)      #3-element vector with the middle value set = NULL
> mean(x)                 #R does not account for NULL values in the function mean()
[1] 6
```

One use of NULL is to build vectors within loops; each iteration adding a new element(s) to the vector. Concretely, NULL values in R are treated as nonexistent; NULL is a special R object without a class (mode).

```
> x <- NULL             #one-element vector of value = NULL
> length(x)
[1] 0
> y <- NA               #one-element vector of value = NA
> length(y)
[1] 1
```

vector names and classes in R

Vector elements can be names with the names( ) function; vectors can also be referenced by their names as "strings". The concatenate c( ) function has unique properties in R:

… Arguments passed to c( ) are reduced to the type of lowest common denominator (mode/class)
… c( ) has a flattening effect for vectors

matrices and arrays in R

A matrix is a vector with two additional attributes: the number of rows and the number of columns. Matrices are also vectors; they have modes (classes). However, vectors are not one-rows/column matrices.

Matrices are special cases of R type object: arrays. Unlike matrices, arrays can be multi-dimensional; having rows, columns, and layers. Matric row and column subscripts (indexes) begin with 1. The internal storage of a matrix is column-major order, meaning all of column 1 is stored initially, then all of column 2, and so on.

However, it can be specified as an argument to build a matrix with row-major order, opposed to the default:

```
> y <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)     #build 2x2 matrix
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y <- matrix(1:4 nrow = 2)                          #only nrow or ncol needed
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y <- matrix(nrow = 2, ncol = 2)                    #individually specifying elements
> y
> y[1, 1] <- 1
> y[2, 1] <- 2
> y[3, 1] <- 3
> y[1, 2] <- 3
> y[2, 2] <- 4
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> m <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE)   #specify byrow argument
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Common Linear Algebra operations can be performed on matrices, such as matrix multiplication, matrix scalar multiplication, matrix addition, matrix subtraction, etc.

```
> y %*% y                #mathematical matrix multiplication
     [,1] [,2]
[1,]    7   15
[2,]   10   22

> 3*y                    #mathematical multiplication of matrix by scalar
     [,1] [,2]
[1,]    3    9
[2,]    6   12

> y+y                    #mathematical matrix addition
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

matrix subsetting [ ] in R

Matrices can be subset like vectors, with the addition of column arguments.

```
> z <- matrix(c(1,2,3,4,1,1,0,0,1,0,1,0), ncol = 3)     #build 3x4 matrix z
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0

> z[,2:3]                                              #subset columns 2-3 of z
     [,1] [,2]
[1,]    1    1
[2,]    1    0
[3,]    0    1
[4,]    0    0

> z[1:2,1]                                         #subset rows 1-2 and column 1 of z
[1] 1 2

> z[,-1]                                        #subset the complement of z, column 1
     [,1] [,2]
[1,]    1    1
[2,]    1    0
[3,]    0    1
[4,]    0    0

> z[1,] <- matrix(c(7,7,7), nrow = 3)                  #assign 1st row with vector
> z
     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    2    1    0
[3,]    1    0    1
[4,]    4    0    0

> x <- matrix(nrow = 5, ncol = 4)    #assign matrix z to rows 2-5 and columns 2-4 of x
> x[2:5,2:4] <- y
> x
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA    7    7    7
[3,]   NA    2    1    0
[4,]   NA    1    0    1
[5,]   NA    4    0    0
```

matrix filtering in R

Matrices can be filtered similar to vectors; however, it is important to remain mindful of the applied syntax.

```
> x <- matrix(c(1,2,2,3,3,4), nrow = 3, byrow = TRUE)
> x
     [,1] [,2]
[1,]    1    2
[2,]    2    3
[3,]    3    4

> x[x[,2] >= 3,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4

> j <- x[,2] >= 3
> j
[1] FALSE  TRUE  TRUE

> x[j,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4
```

The assignment of variable **j** to filter matrix **x** is a vectorized operation; all of the below hold true:

… The object x[, 2] is a vector
… The operator >= compares two vectors
… The number 3 was recycled to a vector of 3's

Similarly, a variable can be used outside of the variable that the filtering is applied:

```
> m <- matrix(1:6, nrow = 3)    #build a 3x2 matrix m
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> m[m[,1] >1 & m[,2] > 5,]    #filter matrix m to the row(s) where the first column
[1] 3 6                        values > 1 and the second column values > 5
```

Note that the above operation should have returned a 1x2 matrix, as opposed to a two-element vector. This is caused by the incorrect data type being assigned. The solution to avoiding this outcome is to apply the drop argument in the function; this will tell R to retain the two-dimensional nature of the data.

Additionally, since matrices are vectors, vector operations can be applied appropriately:

```
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> which(m > 2)         #returns a 4-element vector of all matrix elements of m > 2
[1] 3 4 5 6
```

applying functions to matrix rows and columns $f(x)$ in R

The apply( ) functions are among the most used functions in R; apply( ), tapply( ), lapply( ). Each instructs R to call user-defined functions against each row or each column in a matrix.

A generalized formation of the apply( ) function:

```
> apply(X, margin, FUN, ...)
```

The arguments in the above apply( ) function form are as follows:

… **X** is the matrix
… **margin** is the dimension; **1** applies the function to **rows**, **2** applies the function to **columns**
… **FUN** is the applied function
… **...** are optional arguments to be supplied to function **f**

Below applies the mean( ) function matrix **z** as an example:

```
> z <- matrix(1:6, nrow = 3)     #build 3x2 matrix z
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> apply(z,2,mean)          #apply the mean function to matrix z by columns
[1] 2 5

> colMeans(z, 2L)          #apply the colMeans function to matrix z by columns
[1] 2 5
```

Note that the colMeans( ) function performs the same operation and could be used instead; however, the above example is for illustrative purposes of the apply( ) function.

```
> f <- function(x) x/c(2,8)     #build function f that divides the elements
> y <- apply(z,1,f)              of x by the two-dimensional vector(2, 8)
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75

> t(y)                           #transpose matrix y to reflect the
     [,1]  [,2]                   dimensions of the original matrix z
[1,]  0.5 0.500
[2,]  1.0 0.625
[3,]  1.5 0.750
```

adding and deleting matrix rows and columns in R

Matrices are fixed in terms of rows and columns; thus, the latter cannot be technically *deleted*. Instead, matrices can be ***assigned*** and ***reassigned*** resulting in the same net effect.

Revisiting the reassignment of **vectors** to change size:

```
> x <- c(12, 5, 13, 16, 8)                        #build five-element vector
> x
[1] 12  5 13 16  8

> x <- c(x, 20)                                   #append value 20
> x
[1] 12  5 13 16  8 20

> x <- c(x[1:3], 20, x[4:6])                      #insert value 20
> x
[1] 12  5 13 20 16  8 20

> x <- x[-2:-4]                                   #delete elements 2 through 4
> x
[1] 12 16  8 20
```

Note: reassignment occurs often unseen. For example, even the assignment **x[2] <- 12** is a reassignment.

Analogous operations can be used to change the size of a **matrix**:

```
> one                                               #build four-element vector “one”
[1] 1 1 1 1

> z <- matrix(c(1:4,1,1,0,0,1,0,1,0), nrow = 4)     #build 4x3 matrix z
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0

> cbind(one,z)                                      #bind vector “one” to matrix z
     one
[1,]   1 1 1 1
[2,]   1 2 1 0
[3,]   1 3 0 1
[4,]   1 4 0 0

> cbind(1,z)                                 #bind values 1 to matrix z by recycling
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    1    0
[3,]    1    3    0    1
[4,]    1    4    0    0
```

```
> m <- matrix(1:6, nrow = 3)            #build 3x2 matrix m
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> m <- m[c(1,3),]                       #delete row 2 through reassignment
> m
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

matrix and vector distinction in R

Considering that matrices are also vectors:

```
z <- matrix(1:8, nrow = 4)              #build 4x2 matrix z
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

As a vector, the length of matrix z can be queried:

```
> length(z)
[1] 8
```

However, a matrix is in its own class as a “matrix” object (an S3 class).

```
> class(z)                                #print the class of matrix z
[1] "matrix"

> attributes(z)                           #print the attributes of matrix z
$dim
[1] 4 2

> dim(z)                                  #the dimensions can also be printed
[1] 4 2

> nrow(z)                                 #print the number of matrix z rows
[1] 4

> ncol(z)                                 #print the number of matrix z columns

[1] 2
```

Lastly, the objects themselves can be printed by calling their names:

```
> nrow
function (x)
dim(x)[1L]
<bytecode: 0x00000000095e36c8>
<environment: namespace:base>
```

The benefits of object-oriented programming are shown with functions whose arguments are a matrix, or matrices. The number of rows/columns are available rather than needing to be supplied in arguments.

avoiding dimension reduction in R

There are many cases in statistics where dimension reduction is the goal. However, the latter is only the case when the result of losing dimensions is the *intent*.

```
> z                          #build 4x2 matrix z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

> r <- z[2,]          #subset matrix z for row 2; the result is a vector, not a matrix
> r
[1] 2 6
```

Noting that the result of the above operation is a vector, the original matrix class is lost. Concretely, the result is a two-element vector, as opposed to a 1x2 matrix. The latter is proven as follows:

```
> attributes(z)                #matrix z is a 4x2 matrix
$dim
[1] 4 2

> attributes(r)                #vector r has no discernable attributes
NULL

> str(z)                       #matrix z has 2 indices for rows and columns
 int [1:4, 1:2] 1 2 3 4 5 6 7 8

> str(r)                       #vector r has a single index of range 1:2
 int [1:2] 2 6
```

The problems caused by unintentional dimension reductions within R code can result in general purpose code performing as expected in normal conditions; but failing in special cases. For example, assume a submatrix is extracted from a given matrix, followed by matrix operations on the submatrix. If the submatrix only has a single row, R coerces to a vector and fails in the following matrix operations.

Unintentional dimensionality reduction can be avoided through the use of the **drop** argument.

```
> r <- z[2,, drop = FALSE]       #maintain matrix integrity withy the drop() argument
> r
     [,1] [,2]
[1,]    2    6

> dim(r)                        #the dimensions of matrix r now consists of 2 columns
[1] 1 2
```

The consideration of **drop** as an argument is due to **"["** actually serving as a function, like "+" noted prior.

```
> z[3,2]                     #subset matrix z to return row 3 and column 2
[1] 7

> "[" (z,3,2)      #subset matrix with the "[" function z to return row 3 and column 2
[1] 7
```

Additionally, the **as.matrix( )** function can be used to treat an existing vector as a matrix.

```
> u <- c(1,2,3)              #construct three-element vector u
> u
[1] 1 2 3

> attributes(u)              #vector u contains no discernible attributes, as expected
NULL

> v <- as.matrix(u)          #construct matrix v by treating vector u as a matrix
> v
     [,1]
[1,]    1
[2,]    2
[3,]    3

> attributes(v)              #matrix v contains dimensions of 3 rows and 1 column
$dim
[1] 3 1
```

naming matrix rows and columns ▦ in R

Rows and columns in matrices are often referred to by their corresponding numbers. However, matrix rows and numbers can be assigned names with the **rownames( )** and **colnames( )** functions as follows:

```
> z                                #print 2x2 matrix z
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> colnames(z)                      #matrix z has no names assigned to columns
NULL

> colnames(z) <- c("a", "b")       #assign names to columns of matrix z
> z
     a b
[1,] 1 3
[2,] 2 4

> colnames(z)                      #matrix z has 2 columns names “a” and “b”
[1] "a" "b"

> z[,"a"]                          #column names can be used as references
[1] 1 2
```

higher-dimensional arrays in R

A typical matrix in R consists of rows and corresponding observations; a two-dimensional data structure. Assume the hypothetical where the same observations occur at different time frames amongst the sample (students taking tests in multiple periods during a school term). Time becomes the third dimension in R and these datasets are called ***arrays***. The following example illustrates a series of two tests students take during a given period:

```
> firsttest <- matrix(c(46,21,50,30,25,48), ncol = 2)      #construct fist test matrix
> secondtest <- matrix(c(46,41,50,43,35,49), nrow = 3)     #construct fist test matrix

> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

The **array( )** function is defined above to construct a two-layer array, each consisting of three rows and two columns. The number of layers are assigned with the second **'2'** in the **dim = c(3,2,2)** argument.

```
> attributes(tests)               #the attributes of the tests array are printed
$dim
[1] 3 2 2

> tests[3,2,1]                  #the score on the second portion of test 1 for student 3
[1] 48

>                               #merge firsttest and secondtest matrices into an array
>
> tests <- array(c(firsttest, secondtest), dim = c(3,2,2))
> tests
, , 1

     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

, , 2

     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

In additional to building a three-dimensional array by combining the two matrices above, four-dimensional matrices can be constructed by combining two, or more, three-dimensional arrays, and so on.

A common use of arrays is applied to calculating tables, discussed later.

lists in R

Vectors and matrices contain elements of the same class (mode). Conversely, Lists in R can contain structures of different object types. Lists form the basis for data frames in object-oriented programming.

A list is technically a vector. Ordinary vectors (atomic vectors) cannot be broken down into smaller components; therefore, lists are referred to as ***recursive*** vectors.

For example, the following list represents an employee database with three classes—character, numeric, and a logical. Lists can be constructed as lists of lists, or other types of lists like data frames (discussed later):

```
> j <- list( name = "joe", salary = 55000, union = TRUE)          #named components
> j
$name
[1] "joe"

$salary
[1] 55000

$union
[1] TRUE

> jalt <- list("Joe", 55000, TRUE)                          #numerically indexed components
> jalt
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE
```

As illustrated above, lists can be constructed with names assigned with the **name =** argument; alternatively, the lists can be indexed by numbers. It is best practice to use names for features to support referencing.

Considering the lists are vectors, lists can be created via the **vector( )** function:

```
> z <- vector(mode = "list")
> z[["abc"]] <- 3
> z
$abc
[1] 3
```

general list operations in R

List components can be accessed by treating the list as a vector with numerical indices, using double brackets **[[ ]]**.

```
> j$salary
[1] 55000

> j[["salary"]]
[1] 55000

> j[[2]]
[1] 55000
```

Concretely, the three ways to access a list component **c** of a list **lst** to return the data type of **c**.

… lst$c

… lst[["c"]]

… lst[[i]], where **I** is the index of **c** within **lst**

An important property in accessing components of a list lies in the **data being returned in type c**.
Alternatively, single brackets **[ ]** can be used to access list components, opposed to double brackets **[[ ]]**:
Both options access lists in a vector-index fashion; the difference exists in ***atomic*** (ordinary) vector indexing.

```
> j[1:2]           #subset the first two components of list j with single brackets [ ]
$name
[1] "joe"

$salary
[1] 55000

> j2 <- j[2]       #subset the second component of list j with single brackets [ ]
> j2
$salary
[1] 55000

> class(j2)        #print the class of j2, a subset of list j with single brackets [ ]
[1] "list"

> str(j2)          #print the structure of list j2, confirming the list assignment
List of 1
 $ salary: num 55000
```

The use of single brackets **[ ]** results in another list—a sublist of the original as illustrated above.
Conversely, the above illustration accessed with double brackets **[[ ]]** maintains the **type** of each component:

```
> j[[1:2]]
Error in j[[1:2]] : subscript out of bounds

> j2a <- j[[2]]           #subset of list j2 assigned to j2a returns the component
> j2a
[1] 55000

> class(j2a)              #the class of j2a retains the numeric type
[1] "numeric"

> str(j2a)
 num 55000
```

adding and deleting list elements ➕➖ in R

Adding and deleted elements of a list can be performed in many different contexts. For example...

New components can be added ***after*** a list is created:

```
> z <- list(a = "abc", b = 12)          #create a two-element list z
> z
$a
[1] "abc"
$b
[1] 12

> z$c <- "sailing"                      #add an additional component c
> z
$a
[1] "abc"
$b
[1] 12
$c
[1] "sailing"
```

Components of a list can also be added with a **vector index**:

```
> z[[4]] <- 28
> z[5:7] <- c(FALSE, TRUE, TRUE)
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"

[[4]]
[1] 28

[[5]]
[1] FALSE

[[6]]
[1] TRUE

[[7]]
[1] TRUE
```

Components of a list can be deleted by setting it to **NULL**:

```
> z$b <- NULL
> z
$a
[1] "abc"

$c
[1] "sailing"

[[3]]
[1] 28

[[4]]
[1] FALSE

[[5]]
[1] TRUE

[[6]]
[1] TRUE
```

Noting above that the deletion of **z$b** shifted the indices up by 1. Lists can also be **concatenated**:

```
> c(list("Joe", 55000, TRUE), list(5))
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE

[[4]]
[1] 5
```

Considering a list is a vector, the number of components in a list can be returned with the **length( )** function:

```
> length(j)
[1] 3
```

accessing list components and values in R

Assuming list components have tags (assigned names), they can be returned with the **names( )** function. Additionally, the values of the list are returned with the **unlist( )** function:

```
> names(j)
[1] "name"   "salary" "union"

> ulj <- unlist(j)
   name  salary   union
  "joe" "55000"  "TRUE"

> class(ulj)
[1] "character"
```

The **unlist( )** function returns a vector of character strings in the illustration above, with the names originating from the original list **j**.

The same assumption applies if the list is created as numbers, returning a vector of numbers:

```
> z <- list(a=5, b=12, c=13)
> y <- unlist(z)
> class(y)
[1] "numeric"

> y
 a  b  c
 5 12 13
```

Applying the above behavior when returning functions, note the output of **mixed classes**:

```
> w <- list(a=5, b="xyz")
> wu <- unlist(w)
> class(wu)
[1] "character"

> wu
    a     b
  "5" "xyz"
```

Applying the common denominator coercion to the output of the **unlist( )** function, R applies the highest type of components to result vectors in the hierarchy :

NULL>raw>logical>integer>real>complex>character>list>expression: pairlists are treated as lists.

Although **wu** is a vector and not a list, vector **wu** was still assigned names; of which can be removed.

```
> names(wu) <- NULL    #remove the names of vector wu by setting names to NULL
> wu
[1] "5"   "xyz"

> wun <- unname(wu)    #remove names of vector wu directly with the unname( ) function
> wun
[1] "5"   "xyz"
```

The above preserves the names in vector **wu** for later use; otherwise **wu** could be assigned instead of **wun**.

applying functions to lists *f*( ) in R

the **lapply( )** and **sapply( )** functions are useful for applying functions to lists in R. Similar to the matrix **apply( )** function, the **lapply( )** calls a specific function on each component of a list (or vector coerced to a list) and returns an additional list.

```
> lapply(list(1:3,25:29),median)  #using lapply to call the median function
[[1]]
[1] 2

[[2]]
[1] 27
```

As shown above, the **lapply( )** function calls the median function on all components of the list. In some cases, the list returned above could be simplified to a vector or matrix through the **sapply( )** function.

```
> sapply(list(1:3,25:29),median)
[1]  2 27
```

recursive lists in R

Lists in R can be recursive; in the sense that lists can exists within lists.

```
> b <- list(u = 5, v = 12)                        #a consists of a two-component list
> c <- list(w = 13)                               #with each component also being
> a <- list(b,c)                                  #its own separate list
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12

[[2]]
[[2]]$w
[1] 13

> length(a)
[1] 2
```

The concatenate function **c( )** has an optional argument of ***recursive***, controlling whether ***flattening*** occurs when recursive lists are combined. The second example results in a single list; opposed to a recursive list.

```
> c(list(a=1, b=2, c=list(d=5, e=9)))#concatenate lists, default recursive arg = FALSE
$a
[1] 1

$b
[1] 2

$c
$c$d
[1] 5

$c$e
[1] 9

> c(list(a=1, b=2, c=list(d=5, e=9)), recursive = TRUE) #concatenate lists, arg = TRUE

  a   b c.d c.e
  1   2   5   9
```

data frames in R

Intuitively, data frames are similar to matrices; having two-dimensional rows-and-columns structure. The difference exists where each column in a list can be of different classes (modes). For example, one column can be numerical values, while another contains character strings. Therefore, just as lists are heterogeneous analogs of vectors in one-dimension, data frames are heterogeneous analogs of matrices for two-dimensional data.

Concretely, a data frame is a list, with the components of that list being equal0length vectors. R does not allow components to be other types of objects, including other data frames. Therefore, the result is a heterogeneous-data analogs of arrays, not common in practice. It will thus be assumed that all components of a data frame are vectors.

creating data frames in R

When building data frames in R, the default selection sets **stringsAsFactors = TRUE**, converting character vectors into ***factors*** (discussed later).

```
> kids <- c("Jack", "Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages, stringsAsFactors = FALSE)
> d
  kids ages
1 Jack   12
2 Jill   10
```

accessing data frames in R

Considering that **d** is a list, it can be accessed via component index values or component names:

```
> d[[1]]                    #accessing dataframe d with components indices
[1] "Jack" "Jill"

> d$kids                    #accessing dataframe d with component names
[1] "Jack" "Jill"
```

Additionally, data frame **d** can be treated as a matrix to view components (also illustrated in its structure):

```
> d[,1]                                    #print column one of dataframe d
[1] "Jack" "Jill"

> str(d)                                   #examine matrix-like structure of dataframe d
'data.frame':  2 obs. of  2 variables:
 $ kids: chr  "Jack"
```

The structure of dataframe **d** illustrates two observations—two rows—that stores the data on two variables—two columns. It is safer and more clear to extra data from matrices using name references of the components with the **$** operator. However, in the construction of general code (packages and functions), it is necessary to use matrix-like notation **[ ]** and is also useful for extracting sub dataframes.

extracting sub data frames in R

Matrix operations also apply to data frames where appropriate. Most noted is often **filtering** of dataframes to extract various sub dataframes relevant to analysis.

```
> exams                          #print dataframe exams
  Exam 1 Exam 2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0

> exams[2:5,]                    #subset rows 2 through 5 of dataframe exams
  Exam 1 Exam 2 Quiz
2    3.3      2  3.7
3    4.0      4  4.0
4    2.3      0  3.3
5    2.3      1  3.3

> exams[2:5,2]                 #subset rows 2-5 of column 2 in exams (returned as vector)
[1] 2 4 0 1

> class(exams[2:5,2])          #print class of dataframe exams subset above
[1] "numeric"

> exams[2:5,2,drop = FALSE]    #specify the drop argument to maintain matrix integrity
  Exam 2
2      2
3      4
4      0
5      1

> class(exams[2:5,2,drop = FALSE])    #print the class of the subset with drop = FALSE
[1] "data.frame"
```

Additionally, dataframes can be filtered through sub dataframe extraction and specification of constraints.

```
> exams[exams$`Exam 1` >= 2.5,]
  Exam 1 Exam 2 Quiz
2    3.3    2.0  3.7
3    4.0    4.0  4.0
6    3.3    3.7  4.0
```

data frames with NA values Y in R

Assuming that NA values exist within the rows/columns of a dataframe, certain operations cannot be properly performed. In these cases, R should be specifically assigned to remove NA values from operations.

```
x <- c(2, NA, 4)
> mean(x)
[1] NA

> mean(x, na.rm = TRUE)
[1] 3
```

When dealing with NA values in larger datasets, it might be necessary to remove all observations where NA values are present. This is achieved through the **complete.cases( )** function as illustrated on the proceeding page:

```
> exams                 #print the exams dataframe with NA value in last row
  Exam 1 Exam 2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0
7    2.0     NA  4.0

> mean(exams)          #attempt to return the mean of the exam dataframe
[1] NA
Warning message:
In mean.default(exams) : argument is not numeric or logical: returning NA

> complete.cases(exams)       #print the result of rows that exclude NA values
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

> cmplExams <- exams [complete.cases(exams),]  #subset the dataframe from non-NA obs.
> cmplExams
  Exam 1 Exam 2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0
```

rbind( ) and cbind( ) with data frames in R

The **rbind( )** and **cbind( )** functions apply to dataframes equally, given compatible dataframe sizes; additional columns and rows are required to have the same length as the target dataframe. It is typical where an added row is in the form of another dataframe or list:

```
> d
  kids ages
1 Jack   12
2 Jill   10

> rbind(d, list("laura", 19))
   kids ages
1  Jack   12
2  Jill   10
3 laura   19
```

Additionally, new columns can be created from existing columns:

```
> eq <- cbind(exams, exams[,2] - exams[,1])
> eq
  Exam 1 Exam 2 Quiz exams[, 2] - exams[, 1]
1    2.0    3.3  4.0                     1.3
2    3.3    2.0  3.7                    -1.3
3    4.0    4.0  4.0                     0.0
4    2.3    0.0  3.3                    -2.3
5    2.3    1.0  3.3                    -1.3
6    3.3    3.7  4.0                     0.4
```

To avoid extraneous names being assigned to calculated columns, variables can be specified with the **names( )** function. Alternatively, the properties of lists (equal-length vectors) can be applied and calculated in a vectorized faction; the names can be assigned in the midst of this optimization (proceeding page):

```
> exams$ExamDiff <- exams$`Exam 2` - exams$`Exam 1`
> exams
  Exam 1 Exam 2 Quiz ExamDiff
1    2.0    3.3  4.0      1.3
2    3.3    2.0  3.7     -1.3
3    4.0    4.0  4.0      0.0
4    2.3    0.0  3.3     -2.3
5    2.3    1.0  3.3     -1.3
6    3.3    3.7  4.0      0.4
```

The above illustrates the addition of components to an existing list. The property of **recycling** can equally be exploited in the creation of components in existing lists.

```
> d
  kids ages
1 Jack   12
2 Jill   10

> d$one <- 1
> d
  kids ages one
1 Jack   12   1
2 Jill   10   1
```

apply( ) with data frames in R

The **apply( )** function can be used against dataframes, given that the columns are of the same type.

```
> apply(exams,1,max)              #finding the maximum grade each student received
[1] 4.0 3.7 4.0 3.3 3.3 4.0
```

merging data frames in R

In relational database theory, joining tables according to common variable values is an essential task. In R, the latter is accomplished through the **merge( )** function.

```
> merge(x, y)                             #the simplest form of a join in R
```

The above illustration assumes that the two data frames have one or more columns with names in common:

```
> d1
     kids states
1    Jack     CA
2    Jill     MA
3 Jillian     MA
4    John     HI

> d2
  ages    kids
1   10    Jill
2    7 Lillian
3   12    Jack

> d <- merge(d1, d2)  #merging d1 and d2 only returns records with common properties
> d
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

Utilized the **by.x** and **by.y** arguments in the **merge( )** function handles cases where variables contain the same information for joining, but lack consistency across naming conventions (illustrated on proceeding page):

```
> d3
  ages    pals
1   12    Jack
2   10    Jill
3    7 Lillian

> merge(d1, d3, by.x = "kids", by.y = "pals")
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

It is possible that duplicate matches will appear with the full results of a **merge( )** application.

```
> d1
     kids states
1    Jack     CA
2    Jill     MA
3 Jillian     MA
4    John     HI

> d2a <- rbind(d2, list(15, "Jill"))
> d2a
  ages    kids
1   10    Jill
2    7 Lillian
3   12    Jack
4   15    Jill

> merge(d1, d2a)     #One of the Jill's has unknown residence; erroneously assigned MA
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
3 Jill     MA   15
```

The above illustrations displays the unintended consequences of applying certain **merge( )** functions to dataframes in R. It is imperative to be aware of the data and how it is being applied in R functions.

applying functions to data frames *f* (▦) in R

Similar to lists, the **lapply( )** and **sapply( )** functions can be applied to dataframes in R; remembering that dataframes are special cases of lists (list components consisting of the dataframe's columns). Therefore, calling **lapply( )** on a dataframe with specified function **f( )**, will be called on each of the frame's columns, with the returned valued populated in a corresponding list.

```
> d
  kids ages
1 Jack   12
2 Jill   10

> dl <- lapply(d, sort) #create list dl with two vectors, sorted kids and sorted ages
> dl
$kids
[1] Jack Jill
$ages
[1] 10 12
```

Noting that **dl** is a list; dataframe coercion to a is accomplished through the **as.data.frame( )** function:

```
> as.data.frame(dl) #coercion to a dataframe has lost column references (Jack ≠ 10)
  kids ages
1 Jack   10
2 Jill   12
```

factors and tables in R

Factors are data type in R that resembles that of nominal (categorical) variables in statistics. Factors are nonnumeric in nature, even if coded to numerical values in practice. Factors are viewed in R as vectors with some additional information attached; the extra information being a record of distinct values in a vector, referred to as ***levels***. The illustration below displays the results of a factor coercion:

```
> x <- c(5,12,13,12)
> xf <- factor(x)
> xf
[1] 5  12 13 12
Levels: 5 12 13
```

The ***levels*** in the above illustration represent the distinct values in vector **xf—5, 12, 13**.

```
> str(xf)                                   #note the nonnumeric "5" representation
 Factor w/ 3 levels "5","12","13": 1 2 3 2

> unclass(xf)  #note that the core of xf is the level representations of 1, 2, and 3
[1] 1 2 3 2
attr(,"levels")
[1] "5"  "12" "13"
```

In additional to the behavior in the above illustration, the **length( )** of a factor is still represented as the length of the data, opposed to a count of levels, etc.

```
> length(xf)
[1] 4
```

Anticipated levels can be added into the factor's construction. For example, the current data represents 3 separate levels (categories, labels, classes, etc.) and a fourth is anticipated to exist in a new dataset:

```
> x                                         #four-element vector x
[1]  5 12 13 12

> xf                                        #factor xf with 3 levels (5, 12, 13)
[1] 5  12 13 12
Levels: 5 12 13

> xff <- factor(x, levels = c(5,12,13,88))    #factor xf with 4 levels (5, 12, 13, 88)
> xff
[1] 5  12 13 12
Levels: 5 12 13 88

> xff[2] <- 88                              #example assignment of additional factor
> xff
[1] 5  88 13 12
Levels: 5 12 13 88
```

Conversely, undefined levels within a factor cannot be added if the level within the factor does not exist:

```
> xff[2] <- 28
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = 28) :
  invalid factor level, NA generated
```

factors and functions $f()$ in R

In the context of factors, the **tapply( )** function is leveraged to apply functions against factor elements.

```
tapply(X, INDEX< FUN = NULL, ..., simplify = TRUE)
```

The call **tapply(x, f, g)** has vector **x**, factor (or list of multiple factors) **f**, and function **g**.

The motivating example illustration is vector **x** of ages of voters and factor **f** of voter party affiliation. Each factor in **f** must consists of equal lengths to **x**. If a component of **f** is a vector, it is coerced into a factor by application of function **as.factor( )** to the vector in **f**.

The operation performed by **tapply( )** temporarily splits **x** into groups, each corresponding to a level (or combination of levels) in factor **f**, finally applying **g( )** to the resulting subvectors of **x** (illustrated below):

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages, affils, mean)
 D  R  U
41 31 21
```

The example illustration is expanded to break down groups by both gender and age. When 2 or more factors are applied, each produces a set of groups computed against each other under **AND** logic. The **mean( )** function will then be applied to return the each of the four subgroups inclusive of the operation:

1) Male < 25; **2)** Male > 25; **3)** Female < 25; **4)** Female >25

```
> d <- data.frame(list(gender = c("M","M","F","M","F","F"),
+                      age = c(47,59,21,32,33,24),
+                      income = c(55000,8800,32450,76500,123000,45650)))
> d
  gender age income
1      M  47  55000
2      M  59   8800
3      F  21  32450
4      M  32  76500
5      F  33 123000
6      F  24  45650

> d$over25 <- ifelse(d$age > 25, 1 ,0)
> d
  gender age income over25
1      M  47  55000      1
2      M  59   8800      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0

> tapply(d$income, list(d$gender, d$over25),mean)
      0         1
F 39050 123000.00
M    NA  46766.67
```

The two specified factors (Age < OR > 25 and Gender) each contain two levels. Therefore, **tapply( )** split (partitioned) the data into four groups; each group representative of each possible permutation of gender and age. Lastly, the **mean( )** function was applied to each group individually.

Using the **split( )** function simply forms the vector into groups, opposed to splitting a vector into groups and ***then*** apply a function against each group; seen in the **tapply( )** function.

```
split(x, f, drop = FALSE, ...)
```

The vector **x** and factor **f** serve similar roles in the **split( )** function as they did in the **tapply( )** function. Note that in the **split( )** function, **x** is allowed to be a dataframe; the latter is not true with **tapply( )**.

```
> d
  gender age income over25
1      M  47  55000      1
2      M  59   8800      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0

> split(d$income, list(d$gender, d$over25))  #list output denoted by $
$F.0
[1] 32450 45650

$M.0
numeric(0)

$F.1
[1] 123000

$M.1
[1] 55000  8800 76500
```

Note that the output is a list with components denoted by **$**; the vectors are represented by the combination of factors that make up the results (e.g. $F.0 = Female < 25, $M.1 = Male > 25).

The example below determines the indices of which vector elements correspond to Male, Female, or Infants:

```
> g <- c("M","F","F","I","M","M","F")
> split(1:7,g)
$F
[1] 2 3 7

$I
[1] 4

$M
[1] 1 5 6
```

The **by( )** function operates similar to that of **tapply( )**, which is internally called within **by( )**. The difference is how the **by( )** function is applied to **objects**, rather than vectors. This is useful for performing certain operations against dataframes and matrices.

```
split(x, f, drop = FALSE, ...)
```

Calls to the **by( )** function are similar to those of the **tapply( )** function. The first argument **x** specifies the data, the second argument **f** specifies the grouping factor(s), and the third argument specifies the function to be applied to each factor group individually. Just as **tapply( )** forms groups of indices of a vector corresponding to each factor level, **by( )** finds groups of row numbers of a dataframe, creating sub dataframes for each corresponding factor level. The specified function is then called against each of the identified factor groups defined in the problem.

working with tables in R

Given the following example:

```
> u <- c(22,8,33,6,8,29,-2)
> fl <- list(c(5,12,13,12,13,5,13),c("a","bc","a","a","bc","a","a"))
> tapply(u,fl,length)
   a bc
5  2 NA
12 1  1
13 2  1
```

Again, **tapply( )** partitions vector **u** into subvectors and applies the **length( )** function to each subvector. Note that the latter operation is independent of the contents of vector **u**; the focus is strictly on the factors. The resulting ***contingency table*** illustrates the frequency of factors occurring in the data. Note the occurrence of **NA** representing that **5** occurred with **"bc"** in no observations from the function. The problem with the above illustration is the misrepresentation of **0** occurrences as an **NA** value; use **table( )** instead:

```
> table(fl)    #applying the table( ) function to appropriately represent all cases
    fl.2
fl.1 a bc
  5  2  0
  12 1  1
  13 2  1
```

The first argument in the **table( )** function is either a factor or a list of factors (two factors in above).

Typically, a dataframe serves as the **table( )** argument. Consider the following example of polling-data:

```
> ct <- data.frame("Vote.for.X" = c("Yes", "Yes", "No", "Not Sure", "No"),
+                  "Voted.for.X.Last.Time" = c("Yes", "No", "No", "Yes", "No"))
> ct
  Vote.for.X Voted.for.X.Last.Time
1        Yes                   Yes
2        Yes                    No
3         No                    No
4   Not Sure                   Yes
5         No                    No
```

The **table( )** function computes the ***contingency table*** for the above dataset.

```
> cttab <- table(ct)
> cttab
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

One-dimensional counts are equally available; counts on a single factor:

```
> table(c(5,12,13,12,8,5))

 5  8 12 13
 2  1  2  1
```

matrix/array-like operations on tables in R

Considering that nonmathematical operators can be used on dataframes, the same applies to tables—*the cell counts component of a table object is an array*. Table cell counts can be accessed with matrix notation:

```
> class(cttab)
[1] "table"

> cttab[1,1]
[1] 2

> cttab[1,]
 No Yes
  2   0
```

Below illustrates multiplicative properties of table operations by converting cell counts to proportions:

```
> cttab/5
          Voted.for.X.Last.Time
Vote.for.X  No Yes
  No       0.4 0.0
  Not Sure 0.0 0.2
  Yes      0.2 0.2
```

The ***marginal values*** (statistical method are the values of a variable when the variable itself is held constant while other variables are summed) are computed with the **apply( )** function below:

```
> apply(cttab, 1, sum)
      No Not Sure      Yes
       2        1        2
```

The above illustration computes that marginal values of the polling data (1+1=2, 1+0=1, 1+1=2).

The ***marginal values*** both dimensions can be computed simultaneously with the **addmargins( )** function.

```
> addmargins(cttab)
          Voted.for.X.Last.Time
Vote.for.X No Yes Sum
  No        2   0   2
  Not Sure  0   1   1
  Yes       1   1   2
  Sum       3   2   5
```

The dimension names of a table can be accessed through the **dimnames( )** function as follows:

```
> dimnames(cttab)
$Vote.for.X
[1] "No"       "Not Sure" "Yes"

$Voted.for.X.Last.Time
[1] "No"  "Yes"
```

other factor- and table-related functions in R

the **aggregate( )** function calls **tapply( )** once for each variable in a group.

```
aggregate(x, ...)
```

The **cut( )** function is a common way to generate factors (particularly with tables) by providing data vector **x** and a set of bins defined by vector **b**. The functions determines the bins each element of **x** falls into.

```
cut(x, b, labels = FALSE, ...)
```

programming structures in R

R is a block-structured language in the context of the ALGOL-descendant family (C, C++, Python, Perl, etc.). Blocks are identified by braces **[ ]**; however, braces optional if a block contains a single statement. Statements are separated by newline characters or by semicolons (optional).

loops in R

Assume the following illustration that returns the squared value of each element **n** in vector **x**:

```
> x <- c(5,12,13)
> for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

Concretely, **for** every element **n** in vector **x**, the function **print(n^2)** returns the squared value of each element individually until the entire vector is passed through. The latter is achieved through looped iterations—the first iteration returns the function **print(n^2)** applied against **n = x[1]**, the second iteration returns the function **print(n^2)** applied against **n = x[2]**, and the third iterations returns the **n = [3]**.

Equally available for looping in R is with the **while** and **repeat** functions; completed with **break** (causing control to exit the loop). The follow illustrates the use of the latter three functions:

```
i <- 1
> while (i <= 10) i <- i+4    #i is assigned 1, 5, 9, and then 13 before breaking
> i
[1] 13

> i <- 1
> while(TRUE) {          #break plays a key role to break the loop after i > 10 = TRUE
+     i <- i+4
+     if (i > 10) break
+ }
> i
[1] 13

> i <- 1
> repeat {     #Note the lack of Boolean exit condition with repeat (break is required
+     i <- i+4
+     if (i > 10) break
+ }
> i
[1] 13
```

Additionally, the **next** function can be used to ***skip*** the remainder of the current iteration of a loop and proceed as follows. The advantage lies in avoiding complexity in nested **if-then-else** statements that can make code overly verbose.

The **for** constructs applies to ***any*** vector, regardless of class (mode). For example, filenames can be looped over to read their respective contents and perform a function or series of operations on.

looping over nonvector sets in R

R does not directly support iterative functions over nonvector sets. Some examples of indirect iterations to nonvector sets are as follows:

- … **lapply( )** is applicable, assuming the iterations of the loop are independent of each other; allowing the order of operation to be irrelevant.
- … **get( )** takes a character string as an argument representative of the name of some object and returns the object of the representative name.

Assuming two given matrices **u** and **v**, containing statistical data to become the object of R's linear regression function **lm( )** to each of the latter:

```
> u <- matrix(c(1,2,3,1,2,4), nrow = 3)
> v <- matrix(c(8,12,20,15,10,2), nrow =  3)
> u
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    4

> v
     [,1] [,2]
[1,]    8   15
[2,]   12   10
[3,]   20    2

> for(m in c("u", "v")) {
+     z <- get(m)
+     print(lm(z[,2] ~ z[,1]))
+ }

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
    -0.6667       1.5000

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
     23.286       -1.071
```

The above illustration initially sets **m** equal to **u**; the lines then assign matrix **u** to **z**, allowing **lm( )** call on **u**.

```
> z <- get(m)
> print(lm(z[,2] ~ z[,1]))

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
     23.286       -1.071
```

The loop is then iterated over on matrix **v**, performing the same set of operations; calling **lm( )** on **v**.

if-else statements in R

The **if-else** syntax can be found as follows:

```
> if (r == 4) {
+     x <- 1
+ } else {
+     x <- 3
+     y <- 4
+ }
```

Simplicity aside, it is noted that there is just a ***single statement*** within the **if** function (**x <- 1**). It is thus implied that the braces **{ }** around the initial **if** statement are unnecessary. However, the right brace **}** is used by the R parser to identify the statement as an **if-else**, rather than an **if** alone. An **if-else** statement works as a call to a function, returning the last value assigned:

```
> v <- if(cond) expression1 else expression2if (r == 4)
```

The above sets **v** to the result of **expression1** or **expression2** depending on whether **cond** is **TRUE**.

```
> x <- 2
> y <- if(x == 2) x else x+1
> y
[1] 2

> x <- 3
> y <- if(x == 2) x else x+1
> y
[1] 4
```

As applications become more complex, **expression1** and **expression2** are likely to take on calls to functions. It is noted to not allow compactness to take priority over clarity. When applying **if-else** statements to vectors, the **ifelse( )** function should be applied to produce faster running code.

arithmatic and boolean operators and values in R

a list of basic operators in R programming:

Operation	Description
x + y	Addition
x - y	Subtraction
x * y	Multiplication
x / y	Division
x ^ y	Exponentiation
x %% y	Modular arithmetic
x %/% y	Integer division
x == y	Test for equality
x <= y	Test for less than or equal to
x >= y	Test for greater than or equal to
x && y	Boolean AND for scalars
x || y	Boolean OR for scalars
x & y	Boolean AND for vectors (vector x, y, result)
x | y	Boolean OR for vectors (vector x, y, result)
!x	Boolean negation

R technically does not employ scalar data types; scalars are treated as one-element vectors. However, the above illustrates various Boolean operators for both scalar and vector cases. The example on the proceeding page illustrates the need for the latter distinction:

```
> x <- c(TRUE, FALSE, TRUE)        #create logical vector x
> y <- c(TRUE, TRUE, FALSE)        #create logical vector y
> x
[1]  TRUE FALSE  TRUE

> y
[1]  TRUE  TRUE FALSE

> x & y                            #test Boolean AND for vectors x AND y
[1]  TRUE FALSE FALSE

> x[1] && y[1]                     #test Boolean AND for scalars in x[1] AND y[1]
[1] TRUE

> x && y                           #looks at just the first elements in vector x AND y
[1] TRUE

> if( x[1] && y[1]) print("both TRUE")
[1] "both TRUE"

> if(x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
  the condition has length > 1 and only the first element will be used
> if(x & y) print("both TRUE")
```

The illustration above demonstrates the unique property of an **if** function ***requiring a single Boolean*** in evaluation, opposed to a vector of Booleans. The **warning** result in example where **&** is applied is proof.

The Boolean logicals **TRUE** and **FALSE** can be abbreviated as **T** and **F** (capitalized). When applied in arithmetic, the values of **T** and **F** are represented as **1** and **0**:

```
> 1 < 2                  #1 is less than 2
[1] TRUE

> (1 < 2) * (3 < 4)      #1 is less than 2 and 3 is less than 4 (T * T = 1 * 1 = 1)
[1] 1

> (1 < 2) * (3 < 4) * (5 < 1)       #(T * T * F = 1 * 1 * 0 = 0)
[1] 0

> (1 < 2) == TRUE                   #1 is less than 2 = TRUE, thus TRUE = TRUE
[1] TRUE

> (1 < 2) == 1                      #1 is less than 2 = TRUE = 1, thus 1 = TRUE
[1] TRUE
```

default values for arguments = in R

R employs the use of ***named arguments*** and ***lazy evaluation*** in context. ***Named arguments*** refer to the optional arguments that exist within a given function. ***Lazy evaluation*** refers to how R does not evaluate an argument until/unless the argument is necessary. Therefore, ***named arguments*** may, or may not, be used:

```
read.csv(file, header = TRUE, sep = ",", quote = "\",
dec = ".", fill = TRUE, comment.char = "", ...)
```

The above example of the **read.csv( )** function illustrates the number of **default values** assigned as arguments when the function is called, unless stated otherwise in the code.

return values in R

The return value of a function can be any R object. Values are printed directly to the caller with the **return( )** function. By default, the value of the last executed statement is returned regardless. It is common in practice to avoid the use of the **return( )** function call; there is possibility of lengthening execution time as such. However, for the purpose of tracking function control returning to the caller, the insertion of **return( )** can make the code clearer and maintainable.

Considering the return value can be any R object, **complex objects** are equally applicable:

```
> g                                #print the function g()
function() {
    t <- function(x) return(x^2)
    return(t)
}
> g()                              #print the environment of function g()
function(x) return(x^2)
<environment: 0x0000000005f36210>
```

Functions with **multiple return values** should be placed in a **list**, or other applicable container.

functions are objects in R

Functions in R are referred to as ***first-class objects*** (of class "function"); they can be used like most objects.

```
> g <- function(x) {        #standard function syntax in R
+    return(x+1)
+ }
```

Concretely, the **function( )** is a **function** that serves to create **functions**. The two arguments in the above function are **x**—the formal argument—and the body **return(x + 1)** of class "expression". The illustration above creates a function object which is then assigned to **g**. The brace **"{"** is even a function in itself:

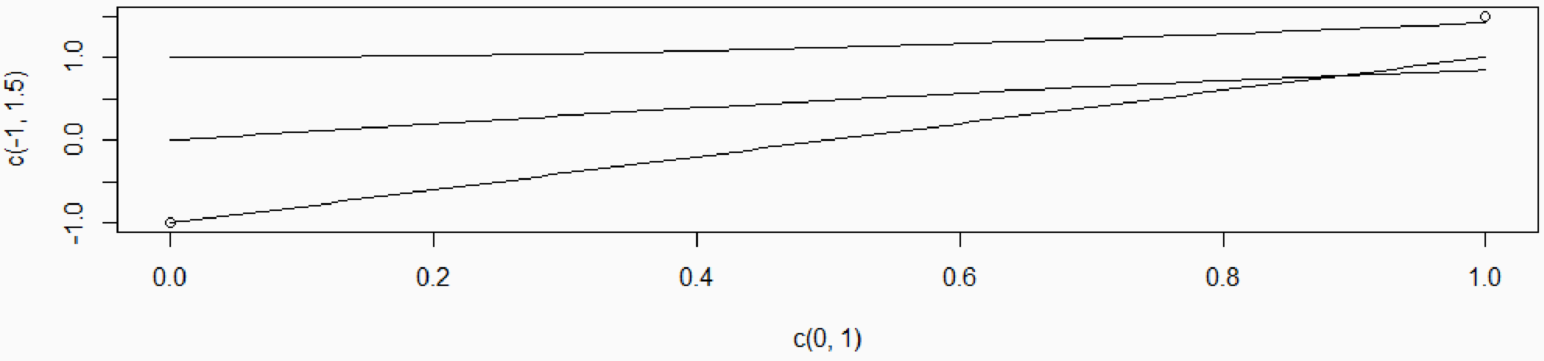
As defined by R – For {, the result of the last expression evaluated. This has the visibility of the last evaluation.

The arguments of the above example can be accessed as follows via the **formals( )** and **body( )** functions.

Because functions are R objects, they can be printed directly to the console by simply calling the function. For longer functions that need examination, the **page( )** function can be used to export the function to text.

Additionally, functions can be assigned to variables and used as arguments in other functions. Functions can be looped through a list of multiple functions equally. Looping through several functions to plot a graph:

```
> g1 <- function(x) return(sin(x))
> g2 <- function(x) return(sqrt(x^2 + 1))
> g3 <- function(x) return(2*x - 1)
> plot(c(0,1),c(-1,1.5))
> for(f in c(g1, g2, g3)) plot(f, 0, 1, add = TRUE)
```



The **formals( )** and **body( )** functions can equally be used to assign/replace functions (discussed more later).

environment and scope issues in R

Along with **functions** consisting of its arguments and body, the function's environment is also stored; made up of the collection of objects present at creation.

The Top-Level Environment

```
> w <- 12
> f <- function(y) {                        #function f() created at the Top-Level Environment
+     d <- 8                                #aka the interpreter command prompt
+     h <- function() {
+         return(d*(w+y))
+     }
+     return(h())
+ }
> environment(f)
<environment: R_GlobalEnv
```

The top-level environment in R is referred to as **R_GlobalEnv** in the output but **.GlobalEnv** in the code.

```
> ls()                    #lists the objects within the Top-Level environment
[1] "f" "w"

> ls.str()                #lists structure of Top-Level objects with more detail
f : function (y)
w :  num 12
```

Within the **scope hierarchy** of R, the environments to variables are defined similar to that of the C programming language; variable **w** is ***global*** to **f( )**, while variable **d** is ***local*** to **f( )**. The difference in R results from an expanded focus on the ***hierarchy***. Unlike C, R can have functions defined within functions—remembering that functions in R are objects and objects can be defined anywhere in R. Taking the above:

h( ) is ***local*** to **f( )**
d is ***local*** to **f( )**
d is ***global*** to **h( )**
y is ***local*** to **f( )** → arguments are considered ***locals*** in R

w( ) is ***global*** to **f( )**
w( ) is ***global*** to **h( )**
***h( )**'s environment consists of any arguments defined when **h( )** comes into existence; upon assignment

```
> h <- function() {
+     return(d*(w+y))
+ }
```

When **f( )** is called multiple times, **h( )** comes into existence multiple times and is then removed each time **f( )** returns. Therefore, **h( )**'s environment are the objects **d** and **y** created within **f( )**; ***plus*** **f( )**'s environment (**w**). Concretely, if one function is defined within another, the inner function's environment includes the environment of the outer function's, including any ***locals*** created thus far in the out function's environment.

With multiple nested functions, there is a nested sequence of larger and larger environments; the "root" environment consists of the top-level objects.

Applying the latter logic to calling the function **f( )** as follows:

```
> f(2)
[1] 112
```

```
> h
Error: object 'h' not found
```

Calling **f(2)** sets ***local*** **d** to **8** and then calls **h( )** → **d*(w+y)** = **8*(12+2)** resulting in **112**. Note that **w** does not explicitly exist within **h( )** and thus, R ascended the hierarchy (to the Top-Level) to find **w <- 12**.

As discussed above, **h( )** is ***local*** to **f( )** and thus not visible at the Top-Level Environment.

function side-effects in R

A property of functional programming is that functions do not change ***nonlocal*** variables (generally no side-effects). The code within a function has read access to ***nonlocal*** variables (not write access). Assumptions of reassignments are merely copies made during execution of a given function.

```
> w <- 12               #Top-Level Environment variable w remains unchanged as follows
> f
function(y) {
        d <- 8
        w <- w + 1
        y <- y - 2
        print(w)
        h <- function() {
            return(d*(w+y))
        }
        return(h())
}

> t <- 4                #Top-Level Environment variable t remains unchanged as above
> f(t)
[1] 13
[1] 120

> w
[1] 12

> t
[1] 4
```

Concretely as illustrated above, references to the ***local*** variable **w** are sent to the same memory location as the ***global*** variable **w** until the ***local*** variable **w** changes; in this case, a new memory location is used. An exception to the read-only nature of ***globals*** is with the **superassignment** operator (discussed later).

no pointers in R

R does not have ***pointers*** or ***references*** similar to those of programming languages like **C**. In Python, functions are available that directly change the arguments themselves:

```
>>> x = [13, 5, 12]
>>> x.sort()
>>> x
[5, 12, 13]
```

To produce the same result, assigning the sorted values of vector **x** to **x**, a ***reassignment*** is necessary:

```
> x <- c(13, 5, 12)
> sort(x)              #sorting vector x
[1]  5 12 13

> x                    #vector x is not assigned the arguments of sort()
[1] 13  5 12

> x <- sort(x)         #vector x must be reassigned to maintain the latter arguments
> x
[1]  5 12 13
```

The above logic can be applied to functions with multiple arguments equally; although with more complexity in the syntax as the arguments and nature of the function itself, expands.

writing upstairs in R

Although code within a certain level of environmental hierarchy has read access to the variables in levels above, direct write access to upper level variables is not possible with the standard assignment **<-** operator.

In order to write a variable to another in an environmental level higher than the variable of focus, the **superassignment <<-** operator, or the **assign( )** function, must be applied.

Writing to nonlocals with the superassignment operator

```
> two <- function(u) {
+     u <<- 2*u
+     z <- 2*z
+ }
> x <- 1
> z <- 3
> u                                          #u is not assigned in the global environment
Error: object 'u' not found

> two(x)                                     #execute function(u) as assigned to variable two
> x                                          #global variable x maintains original assignment
[1] 1

> z                                          #global variable z maintains original assignment
[1] 3

> u                                          #u is superassigned as a top-level variable
[1] 2
```

Although the **superassignment <<-** operator is used to write top-level variables, its function is more discrete. The **<<-** operator performs in upward search in the environment hierarchy, stopping at the first level where a variable of that name is identified; the selected level becomes global in nature.

In the above illustration, **inc( )** is defined within **f( )**. Because there is no assignment to **x** upward in the hierarchy, the **x** within **inc( )** is the one the value is then written to; not **x** at the top-level.

Writing to nonlocals with the assign( ) function

```
> two                              #variation of two( ) above; using assign( ) instead of <<-
function(u) {
    assign("u", 2*u, pos = .GlobalEnv)
    z <- 2*z
}

> two(x)
> x               #x is not within two( ) and maintains its global value
[1] 1

> u               #u is not within two( ) but is superassigned with assign( ) top-level
[1] 2
```

when to use global variables in R

The use of ***global variables*** within the discipline of programming is widely debated. Many experts advocate for the banishment of ***global variables*** under any and all circumstances possible. There are certain circumstances where ***global variables*** provide value in the context of R programming. In the following text, the term ***global variable***, or ***globals*** is used to refer to any variable located higher in the environment hierarchy than the level of the given code within context (focus).

Within the R compiled code and binary constructions, ***globals*** are used widely; both in C code and R routines. For example, the ***superassignment*** **<<-** operator can be found in many R libraries. ***Threaded code*** and ***GPU*** code used for writing fast programs, typically use ***globals*** aggressively in practice; this allows communication avenues between parallel actions/actors.

```
> f
function(lxxyy) {                     #lzzyy is a list containing x and y
    ...
    lxxyy$x <- ...
    lxxyy$y <- ...
    return(lxxyy)
}
> lxy$x <- ...                        #set global variable x
> lxy$y  <- ...                       #set global variable y
> lxy <- f(lxy)
> ... <- lxy$x                        #use new x
> ... <- lxy$y                        #use new y
```

The above code can become unreliable as the variables become more complex, say as list classes. An alternate method of applying ***globals*** within the function are as follows:

```
> f
function() {
    ...
    x <<- ...                         #set global variable x
    y <<- ...                         #set global variable y
}
> x <- ...
> y <- ...
> f()                     #variables x and y are changed within the function execution
> ... <- x                            #use new x
> ... <- y                            #use new y
```

The latter example results in more clear, concise, and manageable code for maintenance and debugging—choosing to use global variables opposed to returning lists. In this theory, the use of ***globals*** become acceptable of they are considered to be **truly global**; being used broadly within the program's environment.

The alter argument to using ***globals*** for simplified code occurs with the cost of simplicity gained; debugging code down the line will become difficult when trying to track the local of ***global*** variable assignments, or reassignments. It is noted, however, that the optimization of text editors allow 'find' functions to locate and identify text within a space (crtl + f); this make sense, considering the original publication calling for abandonment of ***globals*** was written in the 1970s.

An additional argument in the use of ***globals*** is found when certain functions **f( )** are called in multiple independent segments of a program. Each call might require its own value of variables **x** and **y.** The solution would be to set up vectors for **x** and **y** values for each value as a corresponding element in the vector. The ultimate solution, however, loses some of the simplicity gained from the use of ***globals***.

Specifically, in R, a concern with ***globals*** exists at the **Top-Level Environment**. Code using ***globals*** runs the risk of overwriting unrelated variables consisting of the same name, or naming convention. A solution to protecting the integrity of ***globals*** at the **Top-Level** would be to employ application-specific assignments:

The below (left) → **is replaced by the below (right)**

```
> sim <<- list()
```

```
> assign("simenv", new.env(), envir = .GlobalEnv)
```

The above creates a new environment to capture ***globals*** at the top-level; accessed with **get( )** or **assign( )**:

The below (left) → **is replaced by the below (right)**

```
> if(is.null(sim$evnts)) {
+     sim$evnts <<- newevnt
+ }
```

```
> if(is.null(get("evnts", envir = simenv))) {
+     assign("evnts", newevnt, envir = simenv)
+ }
```

The above illustrations loses simplicity for the sake of ***global*** variable integrity, but still remains more manageable than other circumstances (lists of lists of lists) and mitigates unintended reassignments.

closures {} in R

R ***closures*** consist of a function's arguments and body together with its **environment** at the time of a call. Including the **environment** is exploiting a type of programming with a feature **also known as** a ***closure***. A ***closure*** consists of a function that sets up a local variable and creates *another* function to access the variable.

```
> counter
function() {
  ctr <- 0
  f <- function() {
    ctr <<- ctr + 1
    cat("this count currently has value", ctr, "\n")
  }
  return(f)
}
```

Illustrating the operation of a single function in multiple programming environments as follows:

```
> c1 <- counter()                    #assigning counter() to variable c1
> c2 <- counter()                    #assigning counter() to variable c2
> c1                                 #c1 calls counter() in its own environment
function() {
    ctr <<- ctr + 1
    cat("this count currently has value", ctr, "\n")
  }
<environment: 0x0000000003cfbb30>
> c2                                 #c2 calls counter() in its own environment
function() {
    ctr <<- ctr + 1
    cat("this count currently has value", ctr, "\n")
  }
<environment: 0x00000000035ed130>
> c1()                               #call to counter with c1, registers “1”
this count currently has value 1
> c1()                               #call to counter with c1, registers “2”
this count currently has value 2
> c2()                               #call to counter with c2, registers “1”
this count currently has value 1
> c2()                               #call to counter with c2, registers “2”
this count currently has value 2
> c2()                               #call to counter with c2, registers “3”
this count currently has value 3
> c1()                          #note counter() operates in separate environments
this count currently has value 3
```

recursion in R

A ***recursive*** function in R is one that calls itself; the intuition behind ***recursion*** is relatively simple:

To solve a problem of type **X** by writing a **recursive** function **f( ):**

… Break the original problem of type **X** into one or more smaller problems of type **X**.
… Within function **f( )**, call function **f( )** on each smaller problem segment.
… Within function **f( )**, converge the results of each call to **f( )**; solving the original problem.

An famously illustrative example of ***recursion*** is found in the **Towers of Hanoi Problem**.

Illustrative recursion through a Quicksort implementation

Quicksort is an algorithm used to sort a vector of numbers from smallest to largest. The implementation in R can be explained as with the vector (5,4,12,13,3,8,88):

All elements (4,12,13,3,8,88) are compared to element 5 and two subvectors are formed

… Subvector1 → all elements < 5 → (4,3)
… Subvector2 → all elements >= 5 → (12,13,8,88)
… The function is then called upon the subvectors, returning (3,4) and (8,12,13,88)
… The returns are stringed together with element 5, resulting in (3,4,5,8,12,13,88)

The following example is for illustrative purposes, considering R's **sort( )** function is compiled in C (faster):

```
> qs
function(x) {
    if(length(x) <= 1) return(x)
    pivot <- x[1]
    therest <- x[-1]
    sv1 <- therest[therest < pivot]
    sv2 <- therest[therest >= pivot]
    sv1 <- qs(sv1)
    sv2 <- qs(sv2)
    return(c(sv1, pivot, sv2))
}
> x <- c(5,4,12,13,3,8,88)
> qs(x)
[1]  3  4  5  8 12 13 88
```

Noting the ***termination condition***:

```
    if(length(x) <= 1) return(x)
```

Without the above constraint, R would endlessly call upon itself with empty vectors (until R interpreter fails).

Two potential reservations about ***recursion*** in R:

… ***Recursion*** can be fairly abstract when implemented
… ***Recursion*** can be memory-intensive when operating on larger problems

replacement functions in R

```
1    > x <- c(1,2,4)
2    > names(x)
3    NULL
4
5    > names(x) <- c("a","b","ab")
6    > names(x)
7    [1] "a"   "b"   "ab"
8
9    > x
10    a   b  ab
11    1   2   4
```

Recalling an example introduced earlier:

Focusing on **line 5**, appears rather dormant. However, note the that a value is assigned to the result of a function call.

The availability of such an action in R is due to R's ***replacement function*** property. Below is the actual result of executing the script in **line 5**:

```
> x <- "names<-"(x, value = c("a","b","ab"))
```

Concretely, the call is function **names<-( )**.

Any assignment statement where the left side is ***not just*** an identifier (variable name) is considered a ***replacement function***. Concretely, when R is fed the following syntax, note the behavior:

The below (left) → **is computed by R as below (right)**

```
> g(u) <- v
```

```
> u <- "g<-"(u, value = v)
```

Note that function **g(u)** has to be defined in the current R environment prior to the call being successful.

Another example of ***replacement functions*** can be found through **subscripting** operations (also functions):

```
1    > x <- c(8,88,5,12,13)
2    > x
3    [1]  8 88  5 12 13
4
5    > x[3]
6    [1] 5
7
8    > "["(x,3)
9    [1] 5
10
11   > x <- "[<-"(x, 2:3, value = 99:100)
12   > x
13   [1]   8  99 100  12  13
```

As illustrated before, the call in **line 11** is performing the backend calculations in R when the code is executed: `> x[2:3] <- 99:100` which is ultimately verified as follows:

```
> x <- c(8,88,5,12,13)
> x[2:3]  <-  99:100
> x
[1]   8  99 100  12  13
```

tools for function composition in R

Functions can be written directly in the terminal session console (not advised for longer functions):

```
> g <- function() {
+     return(x+1)
+ }
```

Text editors (like Notepad) can be used and directly from the R Console through the **source( )** function:

```
>source("xyz.R")
```

Another option for quick edits to functions is through the **edit( )** function:

```
>f1 <- edit(f1)
```

The **edit( )** function can be used to edit **data structures** equally.

creating binary operations ⊕ in R

Similar to creating functions, ***binary operations*** can also be created in R:

```
> "%a2b%" <- function(a,b) return(a+2*b)
> 3 %a2b% 5
[1] 13
```

anonymous functions in R

In R, the purpose of the **function( )** function is to create functions. Considering the following code:

```
> inc <- function(x) return(x+1)
> inc
function(x) return(x+1)
```

The above illustration instructs R to create a function that adds 1 to its argument and then assigns that function to variable **inc**. However, the assignment is not always taken; it is available to use the function object created by the call to **function( )** without naming the object. Functions in the context are referred to as ***anonymous*** since they are unnamed. The proceeding example illustrates the latter:

```
> z <- matrix(1:6, nrow = 3)          #create matrix z
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> f <- function(x) x/c(2,8)           #create function(x) assigned to f
> y <- apply(z,1,f)                   #apply function f to matric z as variable y
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

The following bypasses the **assignment** to **f** by using an ***anonymous*** function without the call to **apply( )**:

```
> y <- apply(z,1,function(x) x/c(2,8))
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

The third **formal argument** to **apply( )** is required to be a function. This is represented in the above illustration considering that the return value of **function( )** is a function. Sometimes, it is more clear to write the code using ***anonymous functions*** opposed to defining functions ***externally***. However, more complex functions would be less like to benefit from the above application.

mathematics and simulations » in R

The following content applies the constructs and theory of linear algebra and multivariate calculus within R.

Basic mathematical functions in R

function	description
exp()	Exponential function, base e
log()	Natural logarithm
log10()	Logarithm base 10
sqrt()	Square root
abs()	Absolute value
sin(), cos(), etc.	Trigonometric functions
min(), max()	Minimum value and maximum value within a vector
which.min(), which.max()	Index of the minimal element and maximal element of a vector
pmin(), pmax()	Element-wise minima and maxima of several vectors
sum(), prod()	Sum and product of the elements of a vector
cumsum(), cumprod()	Cumulative sum and product of the elements of a vector
round(), floor(), ceiling()	Round to the closest integer, to the closest integer below, and to the closest integer above
factorial()	Factorial function

8.1.1 calculating a probability in R

Using the **prod( )** function, probabilities can be computed in R. Given the following example:

```
> # there are n independent events; the ith event has the probability of
> # pi of occurring. What is the probability of exactly one event occurring?
```

Assuming the first $n = 3$ and the events are named **A**, **B**, and **C**. The computation as follows:

$$\begin{array}{lcl} P(exactly\ one\ event\ occurs) = & \rightarrow & \boldsymbol{P(i) \quad =} \\ P(A\ and\ not\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{P(A \cap B' \cap C') +} \\ P(not\ A\ and\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{P((A \cap B)'\, and\ C') +} \\ P(not\ A\ and\ not\ B\ and\ C) & \rightarrow & \boldsymbol{P(A' \cap (B \cap C)')} \end{array}$$

The computation can also be represented in pseudocode as follows:

$$\begin{array}{lcl} P(exactly\ one\ event\ occurs) = & \rightarrow & \boldsymbol{p(i) \quad =} \\ P(A\ and\ not\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{p_A(1-p_B)(1-p_C) +} \\ P(not\ A\ and\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{(1-p_A p_B)(1-p_C) +} \\ P(not\ A\ and\ not\ B\ and\ C) & \rightarrow & \boldsymbol{(1-p_A)(1-p_B p_C)} \end{array}$$

For the general observation n, the probability of occurrence is calculated as follows:

$$\sum_{i=1}^{n} p_i(1-p_1)\ldots(1-p_{i-1})(1-p_{i+1})\ldots(1-p_n)$$

*The i^{th} term inside the summation is the probability that event i occurs and all others ***do not***.

The R code to compute the preceding mathematics (with probabilities p_i stored in vector **p**) is as follows:

```
> exactlyone  #the notp <- creates a vector of “not occur” probabilities 1-pj by recycling
function(p) {
    notp <- 1 - p
    tot <- 0.0
    for(i in 1:length(p))
        tot <- tot + p[i] + prod(notp[-i])
    return(tot)
}
```

The expression **prod(notp[-i])** computes the produce of all elements of **notp**—sans the i^{th}, as needed.

cumulative sums · products · minima · maxima ⮝⮟ in R

the **cumsum( )** and **cumprod( )** return cumulative sums and products of their applied arguments.

```
> x <- c(12,5,13)
> cumsum(x)            #computes the cumulative sum of ordered values in vector x
[1] 12 17 30

> cumprod(x)           #computes the cumulative product of ordered values in vector x
[1]  12  60 780
```

The is a notable difference between the **min( )** and **pmin( )** functions. Function **min( )** combines all arguments into a single vector, returning the ***minimum*** value. Function **pmin( )**, if applied to two or more vectors, returns a vector of the ***pair-wise minima***.

```
> z <- matrix(c(1,5,6,2,3,2), ncol = 2)
> z
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2

> min(z[,1],z[,2])              #returns the smallest value of (1,5,6,2,3,2)
[1] 1

> pmin(z[,1],z[,2])             #returns the smaller of (1,2); of (5,3); and of (6,2)
[1] 1 3 2
```

Additionally, more than two arguments can be used in the **pmin( )** function:

```
> pmin(z[1,],z[2,],z[3,])              #returns the minima of (1,5,6); and of (2,3,2)
[1] 1 2
```

The **max( )** and **pmax( )** are exhibit analogous behavior to those of the **min( )** and **pmin( )** functions:

Functions ***minimization/maximization*** can be accomplished through the **nlm( )** and **optim( )** functions. The following example identifies the smallest value of $f(x) = x^2 - \sin(x)$:

```
> nlm(function(x) return(x^2 - sin(x)), 8)
$minimum
[1] -0.2324656

$estimate
[1] 0.4501831

$gradient
[1] 4.024558e-09

$code
[1] 1

$iterations
[1] 5
```

The **minimum value** in the above illustration was identified as approximately -0.23, occurring at $x = 0.45$.

The above technique derives from a Newton-Raphson method of numerical analysis for approximating roots; the functions runs through **5 iterations** in the above example. The second argument in the **nlm( )** function specifies the initial estimation (**8**); Note, that the example above employs **8** arbitrarily. More discipline should be applied in practice to ensure convergence.

calculus $\int_0^1 f(x)$ in R

R has many capable calculus applications, including symbolic differentiation and numerical integration.

$$\frac{d}{dx}e^{x^2} = 2xe^{x^2} \qquad \text{and} \qquad \int_0^1 x^2 dx \approx 0.3333333$$

```
> D(expression(exp(x^2)), "x")
exp(x^2) * (2 * x)

> integrate(function(x) x^2, 0, 1)
0.3333333 with absolute error < 3.7e-15
```

There are many available calculus packages in R to leverage (a small few listed below):

Example calculus packages in R

function	description
odesolve	differential equations
ryacas	interfacing R with the Yacas symbolic mathematics system
Deriv	symbolic differentiation
numDeriv	the standard for numerical differentiation in R
pracma	functions for computing numerical derivatives
gaussquad	a collection of functions to perform Gaussian quadrature

statistical distribution functions in R

To no surprise, R has a core magnitude of statistical distributions covered in the CRAN.

The distribution is typically prefixed with the data scope:

… **d** for the density or probability mass function (**pmf**)
… **p** for the cumulative distribution function (**cdf**)
… **q** for quantiles
… **r** for random number generation

with what follows, after the prefix, indicating the distribution applied;

Common R statistical distribution function examples

distribution	density/pmf	cdf	quantiles	random numbers
normal	dnorm()	pnorm	qnorm()	rnorm()
chi square	dchisq()	pchisq()	qchisq()	rchisq()
binomial	dbinom()	pbinom()	qbinom()	rbinom()

The following simulates 1,000 chi-square variates with 2 degrees of freedom; finding their mean:

```
> mean(rchisq(1000, df = 2))       #"r" specifies the generation of random numbers
[1] 1.994469
```

The above initial argument specifies the 1000 random numbers to be generated in the simulation. Additionally, distribution functions in R also have arguments specific to the distribution families. In the above example, the **df =** argument refers to the degrees of freedom belonging to the **chi-square** family. The following example computes the 95th percentile of the chi-square distribution with 2 degrees of freedom:

```
> qchisq(0.95,2)                       #returns 95% quantile of chi-square distribut
[1] 5.991465

> qchisq(c(0.5,0.95), df = 2)     #returns 50% and 95% quantile of chi-square distribut
[1] 1.386294 5.991465
```

The 1st argument of distribution functions is a vector to evaluate the **d**, **p**, **q**, at multiple points (seen above).

sorting in R

Ordinary numerical sorting of a vector is available through the **sort( )** function:

```
> x <- c(13,5,12,5)
> sort(x)              #returns a numerically sorted vector x
[1]  5  5 12 13

> x                    #vector x remains in the original assigned order
[1] 13  5 12  5
```

The **order( )** function will provide the indices of the sorted values from the original vector:

```
> order(x)
[1] 2 4 3 1
```

The **order( )** function indicates that **x[2]** is the smallest value in vector **x**; **x[1]** being the largest value in **x**.

The **order( )** function can be applied along with **indexing** to sort ***dataframes***:

```
> y
    V1 V2
1  def  2
2   ab  5
3 zzzz  1

> r <- order(y$V2)                        #return the indices of column V2 in dataframe y
> r
[1] 3 1 2

> z <- y[r,]                #assign the sorted index of column V2 from dataframe y to z
> z
    V1 V2
3 zzzz  1
1  def  2
2   ab  5
```

Looking at the **order(y$V2)** call, the resulting **3** identifies **x[3,2]** as the smallest number in **x[,2]**; the **1** identifies **x[1,2]** as the middle number in **x[,2]**; the **2** identifies **x[2,2]** as the largest number in **x[,2]**. The latter call assigned an **index** to be used as an argument in the assignment of **z** for a sorted dataframe.

The **order( )** function can also be applied to ***character variables***:

```
> d                                         #dataframe d
   kids ages
1  Jack   12
2  Jill   10
3 Billy   13

> d[order(d$kids),] d                       #sort dataframe d by kids' names
   kids ages
3 Billy   13
1  Jack   12
2  Jill   10

> d[order(d$ages),]                         #sort dataframe d by kids' ages
   kids ages
2  Jill   10
1  Jack   12
3 Billy   13
```

A related function to sorting in R is the **rank( )** function; reporting the rank of each element in a vector:

```
> x
[1] 13  5 12  5

> rank(x)         #element 13 is ranked 4 (smallest); element 5 appears twice, ranked 1.5
[1] 4.0 1.5 3.0 1.5
```

linear algebra operations on vectors and matrices in R

Multiplying vectors by scalars:

```
> y
[1]  1  3  4 10

> 2*y                                     #element wise multiplication of 2 by vector y
[1]  2  6  8 20
```

Computing the **inner-product** (dot product) of two vectors with **crossprod( )**:

```
> crossprod(1:3, c(5,12,13))         #does not calculate actual vector cross product
     [,1]
[1,]   68
```

The compute → $\mathbf{1 * 5 + 2 * 12 + 3 * 13 = 68}$; note **crossprod( )** *does not* calculate the vector cross product.

Mathematical matrix multiplication is applied through the **%*%** operator, opposed to the ***** operator:

Matrix product notation → $\begin{pmatrix} \mathbf{1} & \mathbf{3} \\ \mathbf{2} & \mathbf{4} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{-1} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{3} \end{pmatrix}$ → The R code as follows:

```
> a <- matrix(1:4, ncol = 2 ,byrow = TRUE)
> a
     [,1] [,2]
[1,]    1    2
[2,]    3    4

> b <- matrix(c(1,0,-1,1), ncol = 2)
> b
     [,1] [,2]
[1,]    1   -1
[2,]    0    1

> a %*% b
     [,1] [,2]
[1,]    1    1
[2,]    3    1
```

The **solve( )** function solves systems of ***linear equations*** and also provide ***matrix inverses***.

Linear System → $\begin{matrix} \mathbf{x_1 + x_2 = 2} \\ \mathbf{-x_1 + x_2 = 4} \end{matrix}$ → Matrix Notation → $\begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{-1} & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{x_1} \\ \mathbf{x_2} \end{pmatrix} = \begin{pmatrix} \mathbf{2} \\ \mathbf{4} \end{pmatrix}$ → code below:

```
> a <- matrix(c(1,-1,1,1), nrow = 2)
> b <- c(2,4)
> solve(a,b)
[1] -1  3

> solve(a)                      #the missing 2nd argument causes the inverse to be compute
     [,1] [,2]
[1,]  0.5 -0.5
[2,]  0.5  0.5
```

Examples of available Linear Algebra functions in R (a few provided below):

Example R linear algebraic functions			
t()	matrix transpose	diag()	extracts the diagonal vector of a square matrix (useful for obtaining variances from a covariance matrix and for constructing a diagonal matrix)
qr()	QR decomposition		
chol()	Cholesky decomposition		
det()	determinant		
eigen()	eigenvalues/eigenvectors	sweep()	numerical analysis sweep operations

The following example notes the flexibility of the **diag( )** function:

```
> m
     [,1] [,2]
[1,]    1    2
[2,]    7    8

> dm  <- diag(m)          #takes the diagonal axis of matrix m and assigns to vector d
> dm
[1] 1 8

> diag(dm)                #creates a matrix using vector dm as the diagonal axis
     [,1] [,2]
[1,]    1    0
[2,]    0    8

> diag(3)                 #creates an identity matrix of size 3x3
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

If the argument is a matrix, **diag( )** returns a vector; If the argument is a vector, **diag( )** returns a matrix. Additionally, if the argument is a scalar, **diag( )** returns the ***identity matrix*** of the specified size.

The **sweep( )** function makes more complex operations available in R. the following illustration takes a 3x3 matrix $\mathbb{R}^{3\times 3}$ and adds **1** to row 1; **4** to row 2; and **7** to row 3.

```
> m <- matrix(1:9, nrow = 3, byrow = TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> sweep(m, 1, c(1,4,7), "+")
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    8    9   10
[3,]   14   15   16
```

The first two arguments of **sweep( )** are similar to **apply( )**: the ***array***; and the ***margin*** (1 for rows, example above). The 4th argument is the ***function*** to apply, with the 3rd argument being the function ***argument***.

set operations ∩∪ in R

Example set operations in R	
Union(x,y)	Union of the sets x and y
Intersect(x,y)	Intersection of the sets x and y
Setdiff(x,y)	Set difference between x and y, consisting of all elements of x not in y
Setequal(x,y)	Test for equality between x and y
X %in% y	Membership, testing whether c is an element of the set y
Choose(n,k)	Number of possible subsets of size k chosen from a set of size n

The following illustrates the use of the above ***set operations*** in R:

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)

> union(x,y)
[1] 1 2 5 8 9

> intersect(x,y)
[1] 1 5

> setdiff(x,y)
[1] 2

> setdiff(y,x)
[1] 8 9
```

```
> setequal(x,y)
[1] FALSE

> setequal(x, c(1,2,5))
[1] TRUE

> 2 %in% x
[1] TRUE

> 2 %in% y
[1] FALSE

> choose(5,2)
[1] 10
```

Considering the symmetric difference between two sets—all elements belong to exactly one of the two operand sets. Because the symmetric difference between sets **x** and **y** consist exactly of those elements in **x** but not in **y** (and vice versa), the code consists of easy calls to **setdiff( )** and **union( )**:

```
> symdiff
function(a,b) {
    sdfxy <- setdiff(x,y)
    sdfyx <- setdiff(y,x)
    return(union(sdfxy,sdfyx))
}
> x
[1] 1 2 5

> y
[1] 5 1 8 9

> symdiff(x,y)
[1] 2 8 9
```

Below offers additional illustration of a binary operand for determining whether one set **u** is a subset of **v**:

```
> "%subsetof%" <- function(
+     return(setequal(inter
+ }
> c(3,8) %subsetof% 1:10
[1] TRUE

> c(3,8) %subsetof% 5:10
[1] FALSE
```

Below applies the **combn( )** function to generate combinations, resulting as follows:

```
> c32 <- combn(1:3,2)
> c32
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3

> class(c32)
[1] "matrix"
```

A function can also be called within **combn( )**:

```
> combn(1:3,2,sum)
[1] 3 4 5
```

simulation programming in R

A common use for R programming is that of running simulations.

The **rbinom( )** function random binomial (Bernoulli) variates. Assuming the probability of correctly predicting ≥ (at least) 4 heads out of 5 coin tosses:

```
> x <- rbinom(100000,5,0.5)      #100,000 random variates, 5 trials, 50% success rate
> mean(x >= 4)                   #resulting in a Boolean vector of equal length to x
[1] 0.1869
```

Of many available simulation functions, some of R's core functions are listed as follows:

Example simulation functions in R	
rnorm()	Normal distribution simulations
rexp()	Exponential simulations
runif()	Uniform simulations
rgamma()	Gamma simulations
rpois()	Poisson simulations

Below finds $E[\max(X,Y)]$; expected value of the maximum of independent N(0,1) random variables X and T:

```
> sum <- 0
> nreps <- 100000
> for (i in 1:nreps) {                #generates 100,000 pairs
+     xy <- rnorm(2)                  #generates 2 N(0,1) values
+     sum <- sum + max(xy)            #adds the maximum of each averaged value
+ }
> print(sum/nreps)
[1] 0.5647364
```

The above code uses an **explicit loop** for the convenience of clarity, the above can be achieved more efficiently with the sacrifice of some computational cost and clarity; overall more compact coding:

```
> emax
function(nreps) {
    x <-  rnorm(2*nreps)                                #2x nreps value
    maxxy <- pmax(x[1:nreps], x[(nreps+1):(2*nreps)])   #fist simulates X, second Y
    return(mean(maxxy))                                 #pmax computes pair-wise maxi
}
> emax(nreps)
[1] 0.566471
```

R-documentation states that all random-number generators use 32-bit integers for seed values. R will generate a different stream of random numbers for each run; the stream can be set with **set.seed( )**.

```
set.seed(8888)        #seed is set to 8888, but can be any number)
```

oject-oriented programming in R

R differs from other **object-oriented programming** languages like C++, Java, and Python, but possesses a number of properties unique to those of **OOP**:

… Everything in R is an object—from numbers to character strings, etc.
… R utilizes ***encapsulation***—where separate, but related data are packaged into single class instances. ***Encapsulation*** enhances clarity within code.
… R classes are ***polymorphic***—where the same function call executes different operations on different object classes. ***Polymorphism*** enhances reuse and reproduction.
… R allows ***inheritance***—extending a given class to a more specialized and distinct class.

s3 classes **S3** in R

The original class structures in R are known as **S3** classes. An **S3** class consists of a list, with a class name attribute and ***dispatch*** capability; allowing generic function use. **S4** classes were developed with the intent of added safety and security to the coding environment; preventing access to a nonexistent class component.

The concept of ***polymorphism*** play a key role in allowing ***generic functions*** and code to be written, regardless of object classes. The following example runs R's regression function **lm( )**:

```
> lmout <- lm(y ~ x)
> class(lmout)         #because the object class is lm ...
[1] "lm"

> lmout                #the generic print() function dispatches to print.lm() method

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
       -3.0          3.5
```

The above illustration of ***generic functions*** is can be explained by looking at the **print( )** function:

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x00000000049d2d38>
<environment: namespace:base>
```

The single call to **UseMethod( )** signifies a ***dispatcher*** function to the appropriate class methods. If the class is removed from the variable **lmout** in the above illustration, the output is unaltered and superfluous:

```
> unclass(lmout)
$coefficients
(Intercept)           x
       -3.0         3.5

$residuals
   1    2    3
 0.5 -1.0  0.5

$effects
(Intercept)           x
  -6.928203   -4.949747    1.224745
$rank
[1] 2

$fitted.values
  1   2   3
0.5 4.0 7.5

$assign
[1] 0 1

$qr
$qr
  (Intercept)          x
1  -1.7320508 -3.4641016
2   0.5773503 -1.4142136
3   0.5773503  0.9659258
attr(,"assign")
[1] 0 1
```

implementations of generic methods in R

All implementations of a given generic method can be found using the **methods( )** functions:

```
> methods(print)    # *asterisks* denote nonvisible functions; not in default namespace
  [1] print.acf*
  [2] print.anova*
  ...
```

The ***nonvisible*** functions in the default namespace can be accessed through the **getAnywhere( )** function:

Illustrating the latter by using the **print.aspell( )** method; **aspell( )** performs spellcheck on a file argument.

Example file ***wrd*** contains the following text: `[1] "Which word is mispelled?"` , applied as follows:

```
> aspell("wrds")
mispelled
  wrds:1:15
```

The point of focus is what mechanism was used to print the output. The **aspell( )** function returns an object of class **"aspell"**. R does not have a generic **print.aspell( )** method. Instead, R called **UseMethod( )** on the object of class **"aspell"**. Furthermore, if the print method is called directly, R will **not** recognize it:

```
> print.aspell(wrds)
Error: could not find function "print.aspell"
```

The solution to the above error is to access the method by calling the **getAnywhere( )** function:

```
> getAnywhere(print.aspell)
A single object matching ‘print.aspell’ was found
It was found in the following places
  registered S3 method for print from namespace utils
  namespace:utils
with value

function (x, ...)
{
    if (nrow(x))
        writeLines(paste(format(x, ...), collapse = "\n\n"))
    invisible(x)
}
<bytecode: 0x000000000309e2a0>
<environment: namespace:utils>
```

As seen above, the function belongs to the **utils** namespace, and thus can be accessed as such:

```
> utils:::print.aspell("wrds")
mispelled
  wrds:1:15
```

Additionally, all of the generic methods can be printed through the **methods( )** function:

```
> methods(class = "default")
  ...
```

writing s3 classes in R

A class is created by forming a list, with the list components representing member variables of the class. The "class" attribute is set manually by calling the **attr( )** or **class( )** function; various implementations of the generic functions are then defined. The latter can be seen in the **lm( )** function as applied earlier:

```
> lm
  ...
    z <- list(coefficients = if (is.matrix(y)) matrix(, 0,
          3) else numeric(), residuals = y, fitted.values = 0 *
          y, weights = w, rank = 0L, df.residual = if (!is.null(w)) sum(w !=
          0) else if (is.matrix(y)) nrow(y) else length(y))
  ...
    class(z) <- c(if (is.matrix(y)) "mlm", "lm")
  ...
```

A listed is created, assigned to **z**, serving as the **"lm"** class instance framework (eventually a function return value). Some components of the latter list were previously assigned (**residuals**). Additionally, the class attribute was set to **"lm"** (the **"mlm"** discussed later).

Constructing a class for the employee example used previously is illustrated as follows:

```
> j <- list(name = "Joe", salary = 50000, union = TRUE)
> class(j) <- "employee"
> attributes(j)
$names
[1] "name"   "salary" "union"

$class
[1] "employee"
```

The **"employee"** class is created with variable **j** as its formal argument; printed with default method below:

```
> j
$name
[1] "Joe"

$salary
[1] 50000

$union
[1] TRUE

attr(,"class")
[1] "employee"
```

The call to print **j** resulted in treatment as a list for printing purposes; print method created below:

```
> print.employee <- function(wrkr) {
+     cat(wrkr$name, "\n")
+     cat("salary", wrkr$salary, "\n")
+     cat("union member", wrkr$union, "\n")
+ }

> methods(,"employee")         #any call to print() on a object class "employee" applies
[1] print
```

The above creation of an **"employee"** class is illustrated and proven by printing variable **j** as follows:

```
> j
Joe
salary 50000
union member TRUE
```

using inheritance ⇄ in R

The notion of ***inheritance*** forms new classes as specialized versions of old classes. Following the employee example, a new class can be created for hourly employees **“hrlyemployee”**; a new subclass of **“employee”**.

```
> k <- list(name = "Kate", salary = 68000, union = FALSE, hrsthismonth = 2)
> class(k) <- c("hrlyemployee", "employee")
```

Note that the new class has an additional variable: **hrsthismonth**. The name of the created class consists of two character strings (the new class, the old class). The ***inheritance*** is seen when variable **k** is printed:

```
> k
Kate
salary 68000
union member FALSE
```

Concretely, the execution of variable **k** from the command line called the **print( )** function; which in turn called the **UseMethod( )** function to search for the **first** of **k**’s two class names (**“hrlyemployee”**). Because there was no method for the **“hrylyemployee”** class, the **UseMethod( )** function searched **employee”**.

Recalling the **“lm”** class from earlier, the following line of code makes sense, as **“mlm”** is a subclass of **“lm”**:

```
> lm
  ...
      class(z) <- c(if (is.matrix(y)) "mlm", "lm")
  ...
```

s4 classes **S4** in R

Some programming schools of thought find S3 classes leave more exposure to the code in R, opposed the expected safety that exists in other **OOP** languages. For example, consider the **employee** example from earlier; three fields names – **name salary union**. The following is a few examples of S3 class exposure:

… Union status is left NULL upon entry
… *Union* is misspelled as *onion*
… A object is created of some class other than the **"employee"** class, but mistakenly assigns the object's class attribute to **"employee"**.

S3 classes will produce no warnings to the above instances; in theory, **S4** classes will warn of the mishaps. **S4** structures are known to be more robust than **S3** classes, some of the basic differences are as follows:

Operation	S3 class	S4 class
Define class	Implicit in constructor code	setClass()
Create object	Build list, set class attribute	New()
Reference member variable	$	@
Implement generic f()	Define f.classname()	setMethod()
Declare generic	UseMethod()	setGeneric()

writing s4 classes in R

S4 classes are defined by calling the **setClass( )** function. The following continues the **employee** problem:

```
> setClass("employee",
+          representation(
+              name = "character",
+              salary = "numeric",
+              union = "logical")
+          )
> employee
```

the S4 class **"employee"** is created. An instance of the class is created by the S4 **new( )** *constructor* function.

```
> joe <- new("employee", name = "Joe", salary = 55000, union = TRUE)
> joe
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 55000

Slot "union":
[1] TRUE
```

Member variables now exist in **"slots"** and are referenced with the **@** symbol; opposed to **$** with S3 classes:

```
> joe@salary
[1] 55000

> slot(joe,"salary")
[1] 55000

> joe@salary <- 88000
> joe@salary
[1] 88000
```

Equally, new assignments can be made to the slots through the **@** symbol. Lastly, to note the improved control around S4 classes, consider the following assignment to a misspelled slot:

```
> joe@salry <- 88000
Error in (function (cl, name, valueClass)  :
  'salry' is not a slot in class "employee"
```

Conversely, an S3 class would not restrict the latter example.

implementing a generic function on an S4 class in R

The **setMethod( )** function is used to define an implementation of a generic function on an S4 class. The following illustration will perform the latter with the **show( )** function; the S4 *analog* of S3's generic **print( )**.

```
> joe
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 88000

Slot "union":
[1] TRUE
```

```
> show(joe)
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 88000

Slot "union":
[1] TRUE
```

Note that the above default print of **joe** actually calls the **show( )** function for **S4 class objects**. The following code redefines the **show( )** function for the **"employee"** class created previously:

```
> setMethod("show", "employee",
+           function(object) {
+               inorout <- ifelse(object@union, "is", "is not")
+               cat(object@name, "has a salary of", object@salary,
+                   "and", inorout, "in the union", "\n")
+           }
+           )
[1] "show"
```

The initial argument in the **setMethod( )** function gives the name of the generic function for class-specific definition; the second argument specifies the class name; the third argument defines the new function:

```
> joe
Joe has a salary of 88000 and is in the union
```

class comparisons **S3** *vs* **S4** in R

The general tradeoff between classes—the convenience of S3 classes, or the security of S4 classes.

A concrete comparison between the S3 and S4 classes is given by **various contributors**.

managing objects in R

As a programmer increases the amount of objects accumulated over time, there are various tools to help:

… The **ls( )** function
… The **rm( )** function
… The **save( )** function
… The **class( )** and **mode( )** function, among others that provide information on object structure
… The **exists( )** function

The **ls( )** command lists all current objects in the workspace. A useful named argument in the **ls( )** functions is **pattern**; enabling *wildcards*:

```
> ls()
 [1] "%subsetof%" "a"          "b"          "c32"
 [5] "d"          "dm"         "exactlyone" "f"
 [9] "inc"        "m"          "r"          "symdiff"
[13] "t"          "two"        "u"          "w"
[17] "x"          "y"          "z"

> ls(pattern = "a")
[1] "a"          "exactlyone"
```

The **rm( )** function is used to remove objects that are no longer needed from the workspace.

```
> rm("%subsetof%")
> ls()
 [1] "a"          "b"          "c32"        "d"
 [5] "dm"         "exactlyone" "f"          "inc"
 [9] "m"          "r"          "symdiff"    "t"
[13] "two"        "u"          "w"          "x"
[17] "y"          "z"
```

A call to **rm(list = ls( ))** will clear the entire workspace of all objects.

```
> rm(list = ls())
> ls()
character(0)
```

Additionally, the above arguments **list** and **pattern** can be combined to effectively manage an environment:

```
> rm(list=ls(pattern = "a"))
> ls()
 [1] "b"          "c32"        "d"         "dm"
 [5] "f"          "inc"        "m"         "r"
 [9] "symdiff"    "t"          "two"        "u"
[13] "w"          "x"          "y"          "z"
```

The **browseEnv( )** is also helpful; it opens a web browser to display the global objects and various details.

A collection of objects can be written to disk with the **save( )** function; retrieved with **load( )** or **attach( )**:

```
save(..., list = character(),
     file = stop("'file' must be specified"),
     ascii = FALSE, version = NULL, envir = parent.frame(),
     compress = isTRUE(!ascii), compression_level,
     eval.promises = TRUE, precheck = TRUE)

load(file, envir = parent.frame(), verbose = FALSE)

attach(what, pos = 2L, name = deparse(substitute(what)),
       warn.conflicts = TRUE)
```

When the exact **structure** of an object returns from a library functions is needed, beyond documentation:

```
The following functions can be utilized to examine object structures in R:
class()     mode()     names()     attributes()     unclass()     str()     edit()
```

R contains facilities for constructing *contingency tables*, discussed previously:

```
> ct <- table(ct)
> ct
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

```
> ctu <- unclass(ct)
> ctu
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1

> class(ctu)
[1] "matrix"
```

The object **cttab** is returned by the function **table( )**, of class **"table"**. The documentation can be referenced via **?table**, or the class can be explored through the available functions in R

Illustrated on the left, the counts portion of the object is a matrix. Note the names of the object are displayed as belonging to the matrix. If the **unclass( )** function as not applied in this context, the output would be dependent on the default **print( )** method for the assigned class of the object (no difference in this case). The **str( )** functions operates similarly in a more compact form. Note that **unclass( )** still applies a class.

Expanding on the above table to examine the underlying code of the **table( )** function used above:

```
> edit(table)   #opens a window display of code, page() does the same through export
```

The **edit( )** function opens the function passed as an argument into another widow for adjustment. The code can then be browsed through a text editor, finding the following code at the end of the **table( )** function:

```
  y <- array(tabulate(bin, pd), dims, dimnames = dn)              #line 77 of table()
  class(y) <- "table"                                             #line 78 of table()
  y                                                               #line 79 of table()
```

The above reveals that the **table( )** function is a wrapper for another; the **tabulate( )** function. More importantly, the code reveals the structure of a **"table"** object is simply an ***array*** created from the ***counts***.

The **names( )** functions prints the components in an object; the **attributes( )** function provides more detail.

The **exists( )** function returns a logical value TRUE or FALSE, indicating the argument object's existence.

```
> exists("ct")
[1] TRUE

> exists("ctu")
[1] TRUE
```

This function is particularly useful when developing functions, loading packages, and running object-dependent logical statements. For example, if general-purpose code is being written, the code may need to determine the existence of certain objects necessary to proper execution; if the object does not exist, then the code will have to create it.

Additionally, objects can be saved to disk through the **save( )** function and reloaded through the **load( )** function; the latter general-code might load previously saved code if the specified object does not exist.

input/output (i/o) ⇄ in R

Input/Output (I/O) serves as a key component in most real-world applications of programming. R programming is not designed to be the outlet most suitable for implementing I/O applications in enterprises or large network structures. However, R employs a versatile set of tools for reading and writing files, navigating directories, and accessing the keyboard and monitor, the Internet, etc.

accessing the keyboad and monitor in R

R offers several functions for accessing the keyboard and monitor:

Access the keyboard and monitor in R:			
scan()	readline()	print()	cat()

The **scan( )** function can be used to read into a vector (numeric or character), from a file or keyboard:

```
scan(file = "", what = double(), nmax = -1, n = -1, sep = "",
     quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",)
```

The **scan( )** function reads files all at once, assuming vector elements are separated by ***whitespace***; including blanks, carriage return/line feeds, and horizontal tabs. The **sep =** argument can be used to specify the latter.

Additionally, the **scan( )** function defaults to reading data as mode (class) **“double”**, controlled by **what =**.

The **scan( )** function is also available for reading from the keyboard for individual entry (enter twice to esc.):

```
> scan()                    #use argument quiet = TRUE to prevent printing read items
1: 1
2: 2 3
4: 4 5 6 7 8 9 10
11:
Read 10 items
 [1]  1  2  3  4  5  6  7  8  9 10
```

The **readline( )** function can be used to read in a single line from the keyboard; often used for prompts:

```
> inits <- readline("type your initials: ")
type your initials: MR
> inits
[1] "MR"
```

At the top-level environment in R, objects can be printed by inputting the variable name and executing the code. Conversely, the latter does not work within the body of a function; the **print( )** function works instead.

```
> x <- 1:3
> print(x^2)
[1] 1 4 9
```

print( ) is a **generic function**; the call is dependent upon what class object the argument is. For example, if the object class passed through the **print( )** function is of the class “table”, then **print.table( )** is called.

Because the **print( )** function can only output a single expression that is numbered, **cat( )** can be better:

```
> print("abc")
[1] "abc"

> cat("abc\n")          #need to specify an end-of-line character to signal break
abc
```

By default, the **cat( )** function prints with intervening spaces:

```
> x
[1] 1 2 3

> cat(x, "abc", "de\n")
1 2 3 abc de
```

The intermittent spaces can be prevented by the use of **sep = ""** argument:

```
> cat(x, "abc", "de\n", sep = "")
123abcde
```

Any string can be used as a value to the **sep =** argument as illustrated below:

```
> cat(x, "abc", "de\n", sep = "\n")         #separated by new line character
1
2
3
abc
de

> x <- c(5,12,13,8,88)
> cat(x, sep = c(".",".",".","\n","\n"))    #separated by a vector of characters
5.12.13.8
88
```

reading and writing files in R

The following material discusses:

… reading dataframes and matrices from files
… working with text files
… accessing files on remote machines
… obtaining file directory information

reading a dataframe or matrix from a file in R

Given file **z** with contents as follows, the file can be read including header information.

```
> z <- read.table("z.txt", header = TRUE)
> z
  Name Age
1 John  25
2 Mary  28
3  Jim  19
```

Note that the **scan( )** function will not work here due to the file having a mixture of ***numeric***, ***character***, and ***header*** information. A simple way to use the **scan( )** function would to read a file containing a matrix with the **byrow** argument.

```
> x <- matrix(scan("x.txt"),
nrow = 5, byrow = TRUE)
Read 15 items
> x
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    1    1    1
[3,]    1    1    0
[4,]    1    1    0
[5,]    0    0    1
```

```
> read.matrix <- function(filename) {
+     as.matrix(read.table(filename))
+ }
> read.matrix("x.txt")
     V1  V2  V3
[1,]  1   0   1
[2,]  1   1   1
[3,]  1   1   0
[4,]  1   1   0
[5,]  0   0   1
```

Note in the above illustration that reading a matrix is also available with **read.table( )**, converted to a matrix.

reading a text file in R

When referring to ***text files***, some computer scientists distinguish between the latter and ***binary files***—every file is technically ***binary*** consisting of 0s and 1s. Therefore, this context refers to ***text files*** as mainly consisting of ASCII characters with the use of newline characters for human perception. Nontext files, like JPEG or .exe files are generally referred to as ***binary files***.

The **readlines( )** functions can be used to read in a ***text file***; one line at a time or in a single operation.

Given the following file **z** from before, without the header information, the files can be read at once:

```
> readLines("z.txt")           #each line is treated as a string, returning a vector
[1] "John 25" "Mary 28" "Jim 19"
```

In order to read the file one line at a time, a **connection** needs to be created first. A **connection** in R is a fundamental mechanism used in various I/O operations; used for file access in the following context. The **file( )**, **url( )**, or other R functions will create a connection.

```
> c <- file("z.txt", "r")                          #open the connection to z.txt
> c
description       class        mode        text      opened    can read    can write
    "z.txt"      "file"         "r"      "text"    "opened"       "yes"         "no"

> readLines(c, n=1)                                #read the first line of z.txt
[1] "John 25"

> readLines(c, n=1)                                #read the second line of z.txt
[1] "Mary 28"

> readLines(c, n=1)                                #read the third line of z.txt
[1] "Jim 19"

> readLines(c, n=1)                       #returned the end of file (EOF) of z.txt
character(0)
```

Note that the connection had to be established for R to keep track of the index as it was read through. Furthermore, the **End of File (EOF)** can be detected in code:

```
> c <- file("z.txt", "r")
> while(TRUE) {
+     rl <- readLines(c, n=1)
+     if (length(rl) == 0) {
+         print("end of file")
+         break
+     } else print(rl)
+ }
[1] "John 25"
[1] "Mary 28"
[1] "Jim 19"
[1] "end of file"
```

```
> c <- file("z.txt", "r")

> readLines(c, n=2)
[1] "John 25" "Mary 28"


> seek(con = c, where = 0)
[1] 18


> readLines(c, n=1)
[1] "John 25"
```

Note that the read can be backtracked by using the **seek( )** function. The **where=0** argument in the **seek( )** function resets the position of the file pointer ***zero*** characters from the start of the file (at the beginning). The return of **16** represents the cursor position prior to the **seek( )** function was called. The **close( )** function will be used to inform the system that the file writing is complete and to write the results to disk.

```
> close(c)
Warning messages:
1: closing unused connection (z.txt)
```

accessing files on remote machines with urls in R

With R being a statistically based programming language, ***file reads*** are often more common than ***writes***. When ***writes*** are needed, the **write.table( )** works almost synonymously to the **read.table( )** function; the difference is *writing* to a dataframe, opposed to *reading* a dataframe.

```
> kids <- c("Jack", "Jill")
> ages <- c(12, 10)
> d <- data.frame(kids, ages, stringsAsFactors = FALSE)
> d
  kids ages
1 Jack   12
2 Jill   10

> write.table(d, "kids")          #writes dataframe d to file kids in working directory
```

The preceding illustration writes the dataframe **d** to a file named **"kids"** located in the working directory. The following illustration writes a **matrix** to a file by using arguments **row.names** and **col.names**.

```
> write.table(xc, "xnew", row.names = FALSE, col.names = FALSE)
```

Additionally, the **cat( )** function can be used in writing to files, appending to files, and writing multiple fields.

```
> cat("abc\n", file = "u")                  #using the cat() function to write to files
> cat("de\n", file = "u", append = TRUE)    #using the cat() function to append files

> cat(file = "v", 1, 2, "xyz\n")      #using the cat() function to write multiple fields
```

Lastly, the **writeLines( )** function can be used in writing to files; if a ***connection*** is to be used, the **"w"** is required to be specified, indicating the function is ***writing*** to a file, opposed to ***reading*** a file.

```
> c <- file("www", "w")                       #creates files www for writing (“w”)
> writeLines(c("abc", "de", "f"), c)          #writes contents to the connection www
> close(c)                                    #closes open connection
```

obtaining file and directory information in R

Among the various function in R to obtain information about files and directories, setting permissions, etc:

Access directory and file information in R:	
file.info()	provides file size, creation time, directory-vs-ordinary file status, and various other set arguments in a character vector
dir()	returns a character vector listing the names of all files in the directory (first argument). if the option **recursive = true** argument is specified, the entire directory tree will be displayed from the root argument.
file.exists()	returns a boolean vector indicating whether the given files exists for each name in the first argument, a character vector
getwd() and setwd()	function to determine the current working directory; function to set/change the current working directory.
?files	returns all the file- and directory-related functions in R

accessing the internet in R

R's **socket facilities** give programmers access to the Internet's TCP/IP protocol. For the context of the material discussed, the term ***network*** refers to a set of locally connected machines, without accessing the internet. The medium between machines in these circumstances is often an Ethernet connection. The ***Internet*** essentially connects ***networks*** together through ***routers***; special purpose computers for connecting two or more machines. All machines connected to the Internet obtain an **Internet Protocol (IP)** address. These numerically rooted numbers can be stored as characters that are translated by a **Domain Name Service (DNS)** provider. Beyond the **IP** address specifying the base location, a **port number** is necessary to allocate the information transmittal; for example, how program A or B on one machine connected to the internet specifies the destination on another machine's program A or B, also connected to the internet.

When A intends to deliver information to B, software is written to a ***socket***; a system call syntactically similar to a call that writes to a file. Within the latter call, A specifies the **IP** address and **port number** of B; In turn, B also writes responses to A in the latter socket. The ***connection*** between A and B at that point is somewhat of an "agreement" to exchange information, opposed to an actual physical connection.

Applications correspond to a ***client/server*** model. Programmers in R that write server programs must assign a port number above 1024 (Matloff, 247). When a server program crashes, a delay occurs before the port is reusable.

sockets in R

Note that all bytes sent by A to B amidst an existing connection are synonymous to a ***single large message***. Therefore, if A relays one line of 10 characters and one line of 20 characters to B, A will consider the two separate lines, but the **TCP/IP** read 30 characters of an incomplete message. Parsing the single large message into lines can be achieved in R through some of the following functions:

Parsing character strings received through sockets in R:

readLines() and writeLines	Allows programming as if **TCP/IP** is sending message line by line. An application that is naturally viewed in line find the functions useful.
serialize() and unserialize()	R objects, like matrices and function calls can be sent by converting to a character string by the sender; returned to object by the receiver.
readBin() and writeBin()	Used for sending data in ***binary form***. Noting the distinction between ***binary*** files and ***text*** files discussed earlier.

Each of the above functions operate within R ***connections***. The choice of function remains critical. A long vector will be more conveniently sent via the **serialize( )** and/or **unserialize( )** functions, but can sacrifice time consumption. The latter is caused by the initial conversion necessary to and from the character representations of the vector, but also because the character representation is typically much longer. The following lists two additional R ***socket*** functions:

Parsing character strings received through sockets in R:

socketConnection()	Establishes an R connection through ***sockets***. The ***port number*** is specified through the **port =** argument; additionally stating whether a server or client is to be created. The latter is achieved through setting the **server =** argument to TRUE or FALSE, respectively. Client cases must also provide the servers **IP** address through the **host =** argument.
socketSelect()	Useful what a server is connected to multiple clients. The main **socklist =** argument consists of a list of connections; the return value is the sublist of connections with data primed for the server to read.

string manipulation in R

There is a substantial amount of **string manipulation** functions in R that are at the disposal of a programmer. The following discusses only a few of these functions and will additionally omit many optional arguments.

The **grep( )** function searches for a specified substring pattern in a vector of **x** strings. If **x** has **n** elements—contains **n** strings—the **grep( )** function will return a vector of length up to **n**. Each vector element is an index of **x** where the substring pattern of **x[i]** is found.

```
> grep(pattern, x, ...)                               #function syntax
> grep("Well", c("Well Bore", "Well Head", "BOP", "Valve"))
[1] 1 2

> grep("well", c("Well Bore", "Well Head", "BOP", "Valve"))
integer(0)
```

Note the **pattern** argument in the above example is case-sensitive.

The **nchar( )** function returns the length of string **x** as follows:

```
nchar(x, type = "chars", allowNA = FALSE, keepNA = NA)    #function syntax
> nchar("Well Head")
[1] 9
```

The **nchar( )** function exhibited difficulties dealing with **NA** values and non-character classes; note that the above function accommodates **NA** values and passing a noncharacter class results in a return error.

The **paste( )** function concatenates multiple strings, returning a single string:

```
> paste("Well", "Head")
[1] "Well Head"

> paste("Well", "Head", sep = "")
[1] "WellHead"

> paste("Well", "Head", sep = ".")
[1] "Well.Head"

> paste("Production", "Well", "Head")
[1] "Production Well Head"
```

The **sep =** argument in the **paste( )** can be used to specify a character between the strings.

The **sprintf( )** function assembles a string from parts in a formatted manner.

```
> i <- 8
> s <- sprintf("the square of %d is %d", i, i^2)
> s
[1] "the square of 8 is 64"
```

Concretely, the function name is intended to indicate the ***string print*** to a **string**, opposed to the screen. The function states to print the string **"the square of"** follows by the decimal value (base 10) of **i**.

The **substr( )** function returns the substring in a given character position range **stop:stop** as passed through.

```
> substring("Well Head", 6, 9)
[1] "Head"
```

The **strsplit( )** function splits a string **x** into an R list of substrings based on *another string split* in **x**.

```
> strsplit("11-01-2016", split = "-")
[[1]]
[1] "11"   "01"   "2016"
```

The **regexpr( )** finds the character position of the first instance of the specified **pattern** within the **text**.

```
> regexpr("Head", "Well Head")
[1] 6
attr(,"match.length")
[1] 4
attr(,"useBytes")
[1] TRUE
```

The above illustration reports that **"Head"** appeared in **"Well Head"**, beginning at character position **6**.

The **gregexpr( )** function is synonymous to the **regexpr( )** with the addition of finding ***all*** string instances.

```
> gregexpr("ing","Cementing and Casing")
[[1]]
[1]  7 18
attr(,"match.length")
[1] 3 3
attr(,"useBytes")
[1] TRUE
```

The above illustration finds **"ing"** twice in **"Cementing and Casing"**, starting at character position 7 and 18.

regular expressions [T] in R

programming languages occasionally reference ***regular expression*** in the context of **string manipulation**. The latter becomes relevant when using any of the below string functions in R:

The following functions should be carefully used with regular expressions in R:						
grep()	grepl()	regexpr()	gregexpr()	sub()	gsub()	strsplit()

A ***regular expression*** is shorthand to specify broad classes of strings. For example, the expression **"[We]"** refers to any string the contains ***either*** of the letters **W** or **e**; illustrated as follows:

```
> grep("[We]", c("Production", "Well", "Head"))
[1] 2 3
```

The above illustration indicates the vector elements **2** and **3** contain either a **"W"** or a **"e"** (case sensitive).

```
> grep("W.l", c("Production", "Well", "Head"))
[1] 2

> grep("w.l", c("Production", "Well", "Head"))               #case sensitive!
integer(0)
```

The above illustration indicates the vector element **2** contains a **"W"** followed by any character before an **"l"**.

```
> grep("P..d", c("Production", "Well", "Head"))
[1] 1
```

The above illustration indicates the vector element **1** contains a **"P"** followed by any 2-characters before **"d"**.

In the case where an actual period **"."** is to be searched, the following displays the importance of syntax:

```
> grep(".", c("Prod.", "Well", "Hd."))
[1] 1 2 3

> grep("\\.", c("Prod.", "Well", "Hd."))
[1] 1 3
```

Note that specifying a period **"."** Chooses elements **1**, **2**, and **3**; periods are ***metacharacters***. Consequentially, the ***metacharacter*** nature of the period must be ***escaped*** prior to calling the function. The latter is achieved by using a backslash **"\"**; note, **two** are required due to the first backslash needing escape.

string utilities in the edtdbg debugging tool in R

The code within the **edtdbg** debugging tool (discussed later) utilizes string utilities. The **dgbsendeditcmd( )** functions is an illustration of the latter:

```
> dgbsendeditcmd                                         #send command to editor
function(cmd) {
    syscmd <- paste("vim --remote-send ", cmd, " iiservername ", vimserver, sep = "")
}
```

The main concept of the above illustration is that the **edtdbg** sends remote commands to the **Vim** text editor. For example, the following instructs **Vim** on server **168** to move the cursor to line **12** (typed into a terminal *shell* window):

```
vim --remote-send 12G --servername 168
```

Understanding the above illustration, it is shown how the **dgbsendeditcmd( )** function incorporates R programming to deliver the same results:

```
> paste("vim --remote-send ",cmd," --servername ",vimserver,sep="")
vim --remote-send 12G --servername 168
```

An additional core element in the operation of **edtdbg** is an arrangement to record the file ***dbgsink*** containing most R debugger data through a call to the **sink( )** function. The information captured includes the line numbers and cursor position in the source file as it is stepped through with R's debugger:

```
debug at cities.r#16: {
```

The following illustration continues to discuss other core functions of the **editdbg** tool in R that extensively demonstrates the usefulness of many R ***string manipulation*** functions. Details on the below code snippets can be found on pages 258-259 of ***The Art of R Programming*** by Norman Matloff:

```
> linenumstart <- regexpr("#",debugline) + 1
> buffname <- substr(debugline,10,linenumstart-2)
> colon <- regexpr(":",debugline)
> linenum <- substr(debugline,linenumstart,colon-1)
```

```
> substr(debugline,10,linenumstart-2)
```

```
> kbdin <- readline(prompt="enter number(s) of fns you wish to toggle dbg: ")
enter number(s) of fns you wish to toggle dbg:
```

```
1 4 5
```

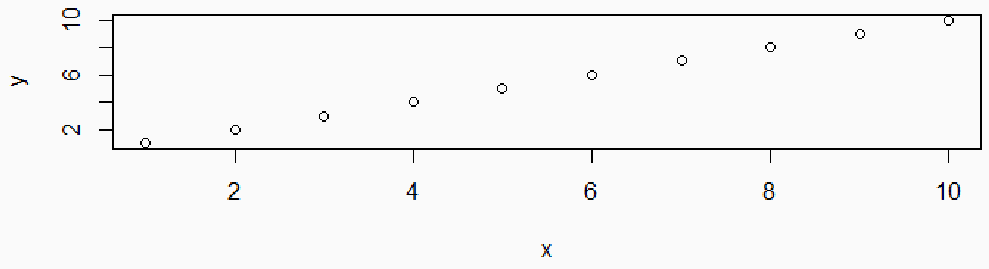
```
> tognums <- as.integer(strsplit(kbdin,split=" ")[[1]])
```

graphics in R

The following content focuses on the graphical functions in the ***base R graphics package***.

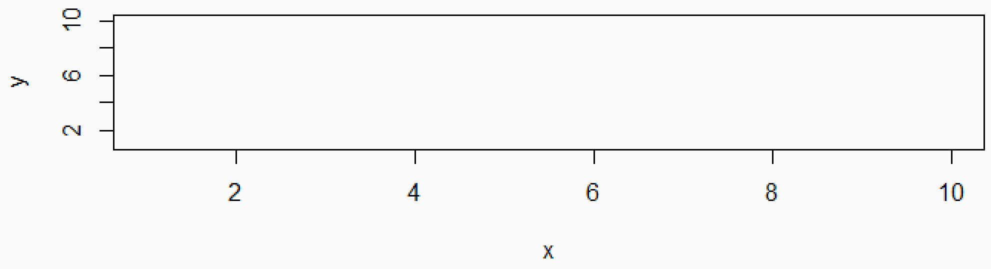
The **plot( )** function serves as the foundation for many of the R graphics capabilities. The **plot( )** function is a *generic* R function; a placeholder for a family of different functions called based on the object class.

```
> x <- 1:10
> y <- 1:10
> plot(x,y)
```



Note that the **plot( )** functions operates in stages; a graph can be built up in iterative stages by issuing a series of commands. For example, an empty base graph can be drawn that only contains the axis':

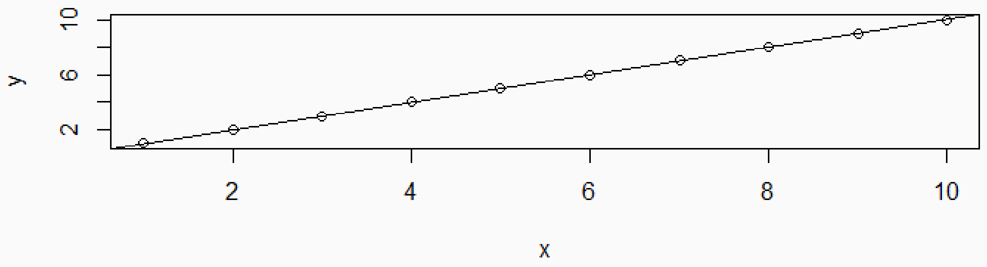
```
> plot(x,y, type = "n",
+      xlab = "x",
+      ylab = "y")
```



adding lines to plots in R

The **abline( )** function is used to add a line to the existing graph in the preceding illustration:

```
> plot(x,y)
> lmout <- lm(y ~ x)
> abline(lmout)
```

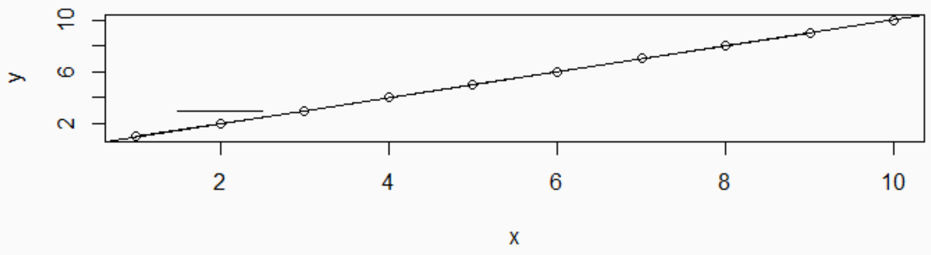


The above line is fitted to the graph based on the call to **lm( )** as a class instance containing the slope and intercept of the line. The regression line was assigned to the **lmout** variable and passed through **lm( )**.

Concretely, the **abline( )** functions takes the slope and intercepts computed by **lm( )** and specifically the **lmout$coefficients** feature from the prior regression function; superimposing the line onto the graph.

Additional lines can be included in the graph through the **lines( )** function; with the arguments of the **lines( )** function being a vectors of *x-values* and a vector of *y-values*:

```
>  lines(c(1.5,2.5), c(3,3))
```

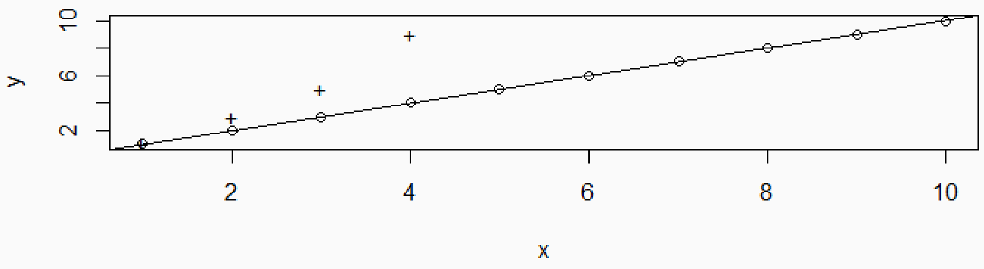


The lines can "connect the dots" through the use of the **type = 1** argument in the **plot( )** or **line( )** functions. Additionally, the **lty** parameter in **plot( )** can be used to specify the line type. The available types of lines in R's **plot( )** function can be examined by executing the **help(par)** function in the console.

adding points to plots in R

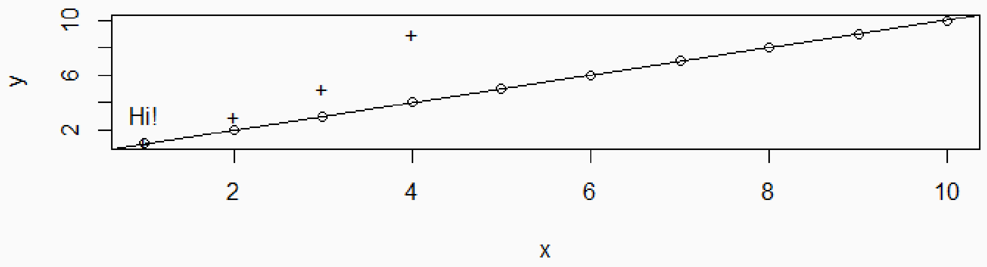
The **points( )** function adds a set of **(x,y)** and respective labels to the currently displayed graph.

```
> plot(x,y)
> abline(lmout)
> points(y1, pch = "+")
```



The **text( )** function is used to add text to a specified coordinate in the plot:

```
> text(3, "Hi!")
```



Similarly, the **legend( )** function can be called to add a legend; specified by the function arguments.

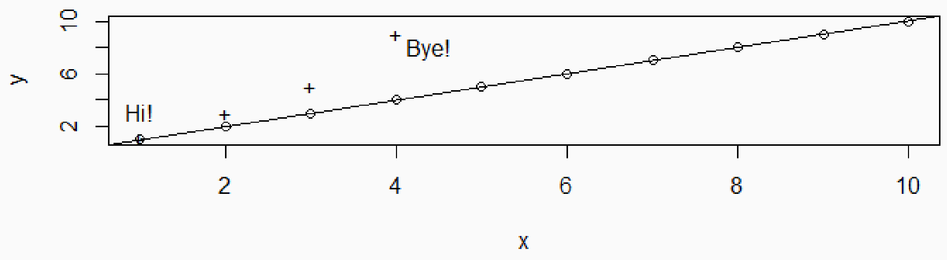
As illustrated by running the **text( )** function above, the exact location of text placement can sometimes be difficult. The solution to the latter is to utilize the **locator( )** function. The function works be converting the cursor to a crosshair, allowing the user to select any point on the graph and return the **(x,y)** coordinates:

```
> locator(1)  #the 1 specifies a single point to be selected and returned from the plot
$x
[1] 3.994866

$y
[1] 8.823904
```

Equally, the **locator( )** function can be used to place text as follows:

```
> text(locator(1), "Bye!")
```



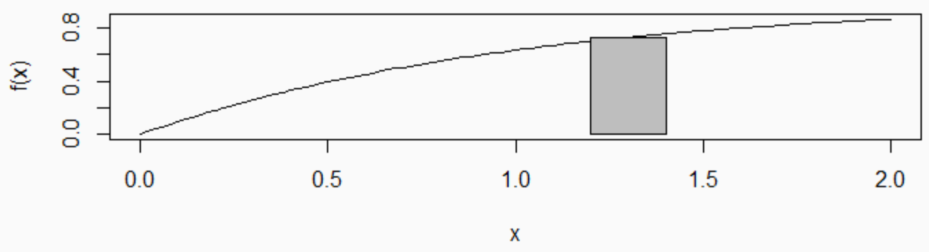
customizing graphs in R

The size of text printed to the graph can be specified by passing the **cex =** argument through **text( )**.

The **xlim( )** and **ylim( )** arguments within the **plot( )** or **points( )** function can be called to specify the axis'. Note that if the largest/longest graph/line is initially plotted first, the graph will properly display all others.

The **polygon( )** function draws arbitrary polygonal objects; below draws $f(x) = 1 - e^{-x}$ and adds a shape:

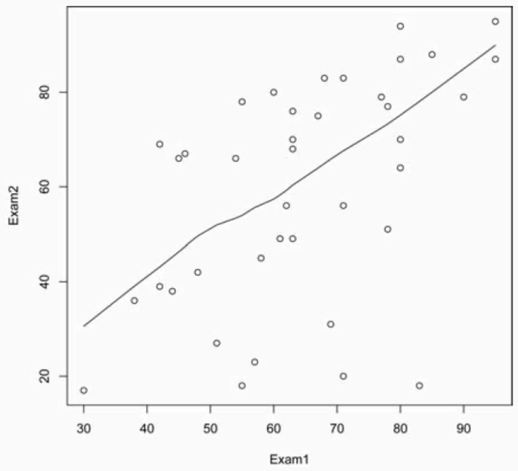
```
> f <- function(x)
      return(1-exp(-x))
curve(f,0,2)
polygon(c(1.2,1.4,1.4,1.2),
        c(0,0,f(1.3),
          f(1.3)),
        col="gray")
```



smoothing points in R

datasets plotted on graph are typically smoothed during the statistical analysis process. This is typically achieved through fitting a ***nonparametric regression estimator*** to the data. In R, the **lowess( )** and more recent **loess( )** functions apply regression smoothing to the data passed through their arguments.

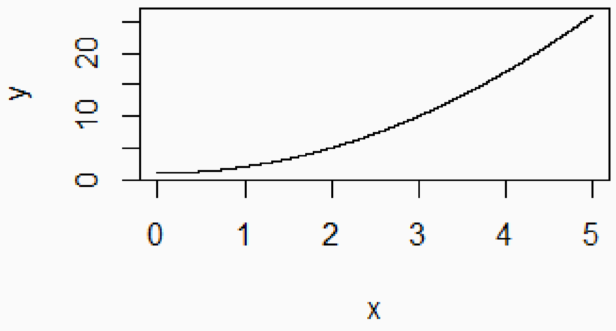
```
> plot(testscores)
> lines(lowess(testscores))
```



graphing explicit functions in R

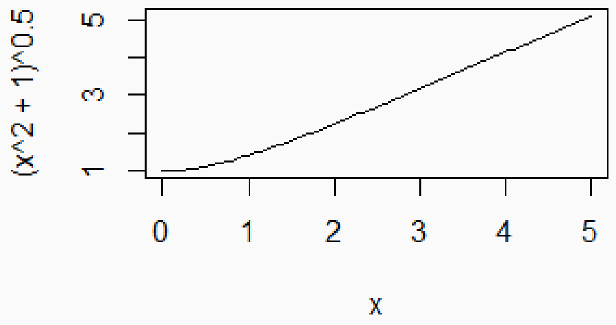
Given the function $g(t) = (t^2 + 1)^{-0.5}$ for t between **0** and **5**:

```
> g <- function(t) {
   return (t^2+1)^0.5 } # define g()
x <- seq(0,5,length=10000)
# x = [0.0004, 0.0008, 0.0012,..., 5]
y <- g(x)
# y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]
plot(x,y,type="l")
```



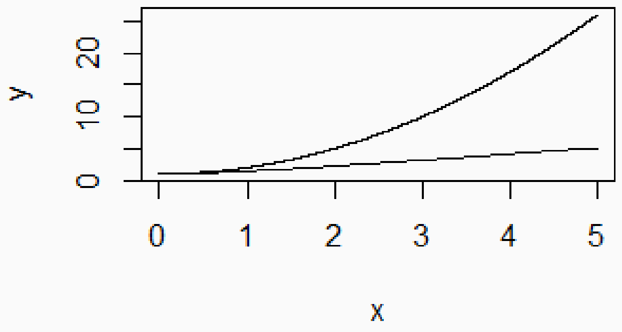
Note the **curve( )** function can be used in substitution of the verbose syntax above:

```
> curve((x^2+1)^0.5,0,5)
```



If the **curve( )** is to be added to an existing plot, the following code will do so along with visually smoothing:

```
> curve((x^2+1)^0.5,0,5, add = TRUE)
```



graphics devices in R

R graphic displays consist of various **graphic devices**. The screen is the default device, RStudio will be the graphic device if that IDE is being used. In order to **save** a graph to a file, an additional device is set up.

To open a file

```
> pdf("file")
```

To view open devices in R

```
> dev.list()
RStudioGD       png       pdf
        2         3         4
```

To view the currently active device in R

```
> dev.cur()
pdf
  4
```

To save displayed graphs in R

One method of saving the currently displayed graph in R is to copy the image to the open **pdf** device:

```
> dev.set(2)
RStudioGD
        2

> dev.copy(which = 4)
pdf
  4
```

The above is executed by reestablishing the screen as the current device and copying it to the **pdf** device.

However, it is best to set up the **pdf** device prior to running the graphics code, or rerunning prior to saving.

To close a graphics device in R

The **pdf** device must be closed prior to interacting with it outside of R.

```
> dev.set(4)
pdf
  4

> dev.off()
RStudioGD
        2
```

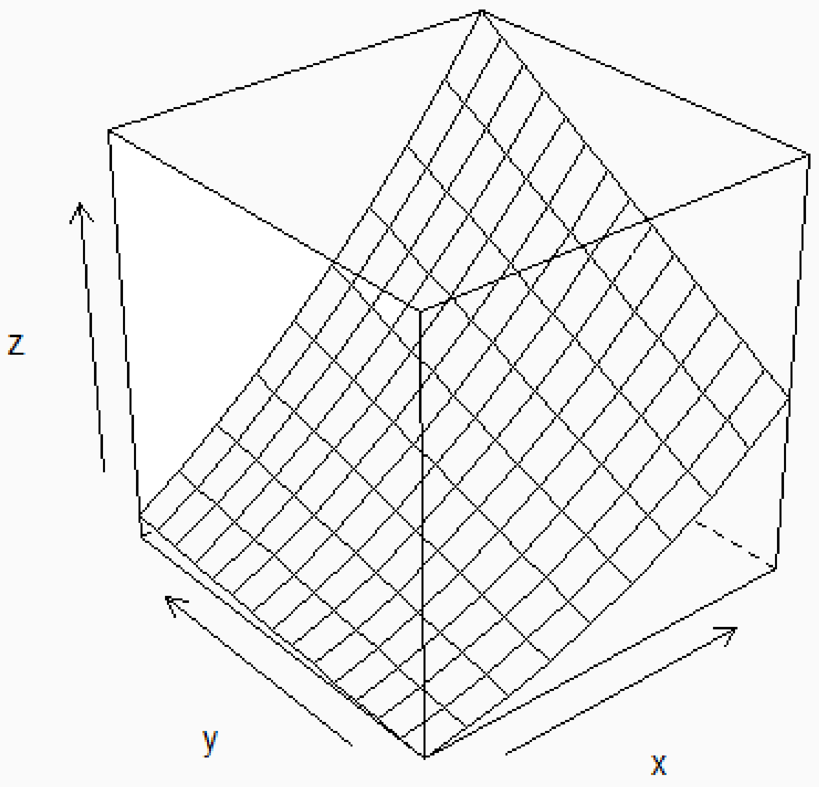
The device an equally be closed through exiting the R session; it is best practice to manage devices through commands rather than assuming a closed session of R will reset to default devices.

creating three-dimensional plots in R

The are many function within R to plot 3-deminsional plots; the **persp( )** and **wireframe( )** functions draw surfaces; and the **cloud( )** function draws three-dimensional scatter plots. The following illustration plots a 3D surface plot using the **wireframe( )** function as a part of the **lattice** library.

```
> library(lattice)
> a <- 1:10
> b <- 1:15
> eg <- expand.grid(x=a,y=b)
> eg$z <- eg$x^2 + eg$x * eg$y
> wireframe(z ~ x+y, eg
```

In the illustration, the **expand.grid( )** function creates a 2-column dataframe **(x,y)**; will all possible combinations of the two input values. With **a = 10** and **b = 15**, the dataframe will be a total of **150 rows**. Column **z** is added as a function of columns **a** and **b**. The **wireframe( )** function is called to create the graph in regression model form; stating that **z** is to be graphed against **x** and **y**. The **wireframe( )** function connects all of the points, like a *surface***.** In contrast, the **cloud( )** function plots isolated points. Note that the **(x,y)** pairs passed through **wireframe( )** must form a rectangular grid.



debugging in R

In the words of Pete Salzman and Norman Matloff, *the principle of confirmation is the essence of debugging*:

```
#Fixing a buggy program is a process of confirming, one by one,
#that the many things you believe to be true about the code actually
#are true. When you find that one of your assumptions is not true,
#you have found a clue to the location (if not the exact nature) of a bug.
```

The idea is to begin small and confirm the behavior of the coding within scripts and objects. Programmers generally agree that code should be written in a **modular** manner. For example, the first-level code should consist of around a dozen lines, containing of mostly calls to functions. The called functions should also be concise and call other functions if necessary. The debugging should be in a **top-down** manner equally. Prior to calling the **debug( )** function in R on an object, run the object to examine the output; whether expected values are returned or not.

Additionally, **Antibugging** approaches can be applied within the syntax. For example, the return of variable **x** should be position, the following code could be inserted:

```
stopifnot(x > 0)
```

Failure to comply with the code will call the **stopifnot( )** function is called:

```
Error: x > 0 is not TRUE
```

Traditional methods of debugging involved insertions of temporary **print( )** statements:

```
x <- y^2 + 3*g(z,2)
cat("x =",x,"\n")
w <- 28
if (w+q > 0) {
   u <- 1
   print("the 'if' was done")
} else {
   v <- 10
   print("the 'else' was done")
}
```

However, inserting temporary **print( )** functions becomes slow and distracting through many iterations.

using debugging facilities in R

The core R debugging facilities consist of the ***Browser***; allowing single steps through the code. The ***Browser*** is called through the **debug( )** function or the **browser( )** function. The debugging facility is specific to functions; for example, function **f( )** is set to debug through **debug(f)**. The ***Browser*** will automatically enter into the beginning of the function until the argument is dropped via **undebug(f)**. Conversely, with long function, the process can be called upon certain lines within a function with the **browser( )** function. To enter a function and remove the set after exiting, the **debugonce( )** function is used effectively.

When the ***browser*** is entered, the command line changes from **>** to **Browse[d] >** with options:

… **n** (for next): executes the next line and then pause. Hitting **ENTER** causes the latter action equally.

… **c** (for continue): like **n**, except that several lines of code may be executed before the next pause. In a loop, this command will result in the remainder of the loop being executed and then pausing upon exit from the loop. In a function but not in a loop, the remainder of the function will be executed before the next pause

… Any R command: still in R's interactive mode and thus can query the value of **x** by simply typing **x**. a variable with the same name as a browser command must explicitly be call something **print( )**

… **where**: prints a stack trace. Displays what sequence of function calls led execution to current location
… **Q**: This quits the browser, returning to R's main interactive mode

The **setBreakpoint( )** function can be called to set certain key locations to bug within the code. Additionally, the **browser** can be called directly in the code by inserting calls to **browser( )** directly within the code. The call to **browser** can be contingent through use of the **expr** argument:

```
> browser(s > 1)         #enters the browser when variable x is greater than 1
> if (s > 1) browser()   #produces the same effect as above
```

A practical example of the above illustration is given a function that has **50** iterations. In order to save time and focus the debugging process, the following code could be used assuming the loop index is **i**:

```
> browser(s > 49)         #enters the browser when variable x is greater than 49
> if (s > 49) browser()   #produces the same effect as above
```

The **setBreakpoint( )** function can be used in the format as follows:

```
> setBreakpoint(filename, linenumber)
```

The above function calls the ***browser*** at the ***linenumber*** passed through as an argument. This is useful through the following example. Assuming the current position in the browser is at **line 12,** within the **x.R** file, and the breakpoint is best set at line **28**, the following function is applied:

```
> setBreakpoint(x.R, 28)
```

The above would require exiting the **browser**, adding a call to **browser( )** at line **28**, and reentering the function if the **setBreakpoint( )** function was not called instead. The **setBreakpoint( )** function works by calling the **trace( )** function. If the **setBreakpoint( )** function had been called on function **g( )**, the breakpoint could be cancelled as follows:

```
> untrace(g)
```

Equally, the **setBreakpoint( )** function can be called whether or not currently in the **debugger**.

tracking in R

The **trace( )** function is flexible with a steep initial learning curve for operating.

```
> trace(f, t)
```

The above instructs R to call function **t( )** each time the function **f( )** is entered. For example, if a breakpoint is to be set at the beginning of function **gy( )**, the following command is applied:

```
> trace(gy, browser)
```

The effect of the above is the same as if the **browser( )** command had been placed in the source code of **gy( )** except considerably more convenient. It is noted that calling the **trace( )** function does **not** alter the source code; instead, the latter alters a *temporary* version of the file maintained by R. The above would subsequently be undone through the following code:

```
> untrace(gy)
```

performaing checks after a crash in R

When R crashes without using the debugger, there is still a debugging tool available after the crash. A ***post-mortem*** can be performed calling the **traceback( )** function. The information regarding the function of the crash will be returned in addition to the call chain of the latter function. More information is obtained if R is set up to **dump frames** in the event of a crash:

```
> options(error = dump.frames)
```

If R is set up as above, the **debugger( )** function can be called after the event of a crash

```
> debugger()
```

The user will then select the level of function calls to inspect regarding the crash in R. For each chosen level, the values of the variables can be examined. After viewing a given level, the **debugger( )** can be returned by selecting the **N** key on the keyboard.

Additionally, R can be set to enter the debugger automatically:

```
> options(error = recover)
```

Note the debugger will be entered even the event of simple syntax errors (not preferable), the above is reversed through the following code:

```
> options(error = NULL)
```

ensuring consistency while debugging in R

In the case of random numbers, it is important to reproduce the same values in order to effectively debug while producing the same errors as the initial cause. The **set.seed( )** function controls the latter by reinitializing the random number sequence of a given value:

```
> runif(3)
[1] 0.1842674 0.7220917 0.4818511

> runif(3)
[1] 0.1563653 0.4570898 0.6093413

> runif(3)
[1] 0.55114431 0.05548648 0.67934008

> set.seed(8888)
> runif(3)
[1] 0.5775979 0.4588383 0.8354707

> set.seed(8888)
> runif(3)
```

performance enhancement in R

The is a constant trade-off between time and space in computer programming. Concretely, the notion of a fast-running program often requires more space in memory. Conversely, a conservative program using less memory will likely sacrifice in performance speed. The trade-off between time and space is relevant in the context of R for the following reasons:

… R is an interpreted language. Many of the commands are written in C and thus do run in fast machine code. But other commands, and your own R code, are pure R and thus interpreted. So, there is a risk that your R application may run more slowly than you would like.

… All objects in an R session are stored in memory. More precisely, all objects are stored in R's memory address space. R places a limit of 231 − 1 bytes on the size of any object, even on 64-bit machines and even if you have a lot of RAM. Yet some applications do encounter larger objects.

writing fast code in R

The main tools available to make code faster in R are as follows:

… Optimize your R code through vectorization, use of byte-code compilation, and other approaches

… Write the key, CPU-intensive parts of code in a compiled language such as C/C++

… Write code in some form of parallel R

To optimize code effectively, an understanding is necessary of R's functional programming nature and the way R allocated memory.

the for loop in R

An alternate to using **for loops** in R is to leverage various ***vectorization*** techniques.

For the use of ***vectorization*** to speed up code, the following vector **x** and **y** are of equal lengths:

```
> z <- x + y        #vectorized summation of all elements in x and y, respectively
```

The above is considerably more computationally cheap than the following **for loop** option:

```
> for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

The proof is seen in runtime comparisons of the above methods of element-wise vector summation:

```
> x <- runif(1000000)
> y <- runif(1000000)
> z <- vector(length=1000000)
> system.time(z <- x + y)
   user system elapsed
   0.052 0.016 0.068

> system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
   user system elapsed
   8.088 0.044 8.175
```

One type of vectorization is ***vector filtering***. Using the **oddcount( )** function from prior examples:

```
oddcount <- function(x) return(sum(x%%2==1))
```

Though there is no explicit **loop** in the above example, R will internally loop through the array in native machine code; therefore, the anticipated speedup does occur.

```
> x <- sample(1:1000000,100000,replace=T)
> system.time(oddcount(x))
   user system elapsed
   0.012 0.000 0.015
```

The following performs the same **oddcount( )** function in the form of a **loop**:

```
> system.time(
+ {
+ c <- 0
+ for (i in 1:length(x))
+ if (x[i] %% 2 == 1) c <- c+1
+ return(c)
+ }
+ )
   user system elapsed
   0.308 0.000 0.310
```

Although both cases ran in less than a second, the timing extrapolated with function complexity and data.

Other forms of vectorized function in R that will potentially increase code performance:

```
The following functions are vectorized in R:
ifelse(), which(), where(), any(), all(), cumsum(), and cumprod()
In the case of matrices: rowSums(), colSums(), etc.
In “all possible combination” types: combin(), outer(), lower.tri(), upper.tri(), expand.grid()
```

Although the **apply( )** function eliminates an explicit loop, it is implemented in R rather than C and will not typically cause an increase in code performance. In contrast, other apply functions such as **lapply( )**, can be useful to help code performance.

functional programming and memory issues in R

Most object in R are ***immutable*** (unchangeable); thus, R operations are implemented as functions that reassign to the given object, a trait that can have performance implications.

Once source of performance issues in R is with certain uses of ***vector assignments***:

```
> x[3] <- 8
> z <- "[<-"(z,3,value = 8)         #the above actually calls replacemet function "[<-"
```

Concretely, the above example reassigns the entire vector **z** even though the semantics instruct changing a single element. The effect leads to mass computational cost in long vectors, or small vectors within loops.

Another source for performance issues in R is with ***copy-change-policy***:

```
> y <- z
```

Assuming the above assignment, variable **y** shares the same memory as variable **z**. If either changes, a copy is made in a different area of memory. However, only the ***first*** change is affected; the relocating of the moved variable removes any sharing issues. The **tracemem( )** function reports such memory locations.

R adheres to ***copy-on-change*** semantics; a situation where R does not exhibit location-change behavior:

```
> z <- runif(10)
> tracemem(z)
[1] "<0x88c3258>"

> z[3] <- 8
> tracemem(z)
[1] "<0x88c3258>"
```

```
> z <- 1:10000000
> system.time(z[3] <- 8)
   user system elapsed
   0.180 0.084   0.265
> system.time(z[33] <- 88)
user system elapsed
   0      0       0
```

The above illustrates the location of **z** never changing between assignments. In an event where the copying is performed, it is copied through the **duplicate( )** function within R’s internal code.

finding slow code spots in R

The **Rprof( )** function provides z report of (approximately) how much time code is spending on each function being called. The usefulness arises due to not every section of code always needing optimizing. Sacrifices often related to optimization can often be code writing time and code complexity.

How Rprof( ) works:

R inspects the call stack to determine which function calls are in effect evert 2 seconds (default value). Each inspection result is written to a file (Rprof.out by default).

The **summaryRprof( )** function will summarize all the latter **Rprof( )** function results written to the file. The **Rprof( )** results can often by cryptic in scenarios where the profiling code produces many functions (indirect calls) and might often not be the best route for understanding the inner workings of an R program.

byte code compilations in R

R includes a ***byte code compiler*** that can be used to potential improve code performance. Returning to the example from before:

```
> z <- x + y                                  #faster than the for loop below
> for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

Applying the ***byte code compiler*** to the above illustration as follows:

```
> library(compiler)
> f <- function() for (i in 1:length(x)) z[i] <<- x[i] + y[i]
> cf <- cmpfun(f)
> system.time(cf())
   user system elapsed
  0.845  0.003   0.848
```

Noted how significantly faster the compiled **for loop** function above runs compared to the original below:

```
> system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
   user system elapsed
   8.088 0.044 8.175
```

However, overall performance remains best with the vectorized implementation as **z <- x + y**.

conquering max memory in R

Remembering that all R objects are stored in-memory, R has a limit of 2^{31}—1 bytes on the size of any given object, regardless of word size (32 vs 64-bit) and RAM.

One way to overcome these limitations is through the use of what is known as **chunking**. The concept is to read data from a disk file in portions. This is achieved through using the **skip** argument within **read.table( )**.

```
For example, a dataset with 1,000,000 records can be divided into 10 chunks. The skip = 0
argument will be set at the initial call. The second call will set the skip = 100000
argument; the process repeats until all of the data is read and computed on as needed.
```

An additional method to deal with R memory limits is through the use of specialized **R packages**. The **RMySQL** package provides a more convenient way to handle large databases through connecting to external SQL databases (database knowledge required). The package essentially performs the large memory-required transactions back at the database and reads the results to R. Another option is the **biglm** package that performs regression and general linear-model analysis on large datasets. To more commonly used packages are **ff** and **bigmemory**. The **ff** package sidesteps memory limitations by storing data on-disk, opposed to in-memory. The **bigmemory** packages does the same, while also storing on a machine's main memory (ideal for multicore machines).

interfacing to other languages in R

The following discusses the basic concepts of interfacing between R and other programming languages. Specifically, the examples below are in the context of calling C/C++ from R and calling R from Python.

writing C/C++ fucntions to be called from R

A general reason to write certain code in C to then be run within the R environment is often to increase performance. Another reason to drop down to C would be to utilize a specialized C I/O. For example, R used **TCP** (discussed earlier) protocols the 3rd layer of the standard internet communication system; C uses **UDP** which is fast in certain circumstances. R has wo base C/C++ interfaces with the **.C( )** and **.Call( )** functions.

The **.Call( )** function is often more versatile, equally requiring knowledge of R's internal structures.

R to C/C++ interfacing preliminaries for the illustrations used:
C stores 2D arrays in **row-major** Order; **R** stores 2D arrays in **column-major** order
C uses **0-based** indexing subscripts; **R** uses **1-based** indexing subscripts
Arguments passed from **R** to **C** are received as pointers
C functions must return **void**; values returned must be communicated through function arguments

The following example uses C code to extract subdiagonals from a square matrix.

(*attributed code written by Min-Yu Huang and published by Norman Matloff*).

```c
#include <R.h> // required

// arguments:
//    m: a square matrix
//    n: number of rows/columns of m
//    k: the subdiagonal index--0 for main diagonal, 1 for first
//       subdiagonal, 2 for the second, etc.
//    result: space for the requested subdiagonal, returned here

void subdiag(double *m, int *n, int *k, double *result)
{
  int nval = *n, kval = *k;
  int stride = nval + 1;
  for (int i = 0, j = kval; i < nval-kval; ++i, j+= stride)
      result[i] = m[j];
}
```

The **stride** variable is a reference to parallel processing methods. Assuming a **1000**-colum matrix $\mathbb{R}^{n\times 1000}$ with the C code looping through each element in a given column; top to bottom. Remembering that C uses **row-major** order, consecutive elements in a column are **1,000** elements apart as a single, long vector. In effect, the long vector is being **traversed** with a **stride** of **1,000** (accessing every 1000th element).

The code is compiled, for this example, in a Linux terminal window (Cygwin package for Windows):

```
% R CMD SHLIB sd.c
gcc -std=gnu99 -I/usr/share/R/include     -fpic -g -O2 -c sd.c -o sd.o
gcc -std=gnu99 -shared -o sd.so sd.o    -L/usr/lib/R/lib -lR
```

The above ultimately creates a dynamic file library named ***sd.so***. The library is subsequently loaded into R and the C function is called as follows:

```r
> dyn.load("sd.so")
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
```

The output is returned in R as follows:

```
result=double(dim(m)[1]-k))
[[1]]
 [1] 1 6 11 16 21 2 7 12 17 22 3 8 13 18 23 4 9 14 19 24 5 10 15 20 25

[[2]]
[1] 5

[[3]]
[1] 2

$result
[1] 11 17 23
```

Note the **result** name is assigned to both the formal argument (in C) and the actual argument (in R). Also note that space was allocated for the **result** in R code. The return values are displayed as a list of arguments.

debugging R/C code in R

Extra challenges arise when debugging R/C code in the context of using **GDB** (discussed previously). The following is the former example of R/C debugging using **GDB** on the ***sd.c*** code in the illustration.

```
$ R -d gdb                          #launch gdb from a terminal window command line
GNU gdb 6.8-debian
...
(gdb) run                           #call gdb to run R
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")                 #load compiled C code into R
>   # hit ctrl-c here               #interrupt to pause R and return to gdb prompt
Program received signal SIGINT, Interrupt.
0xb7ffa430 in __kernel_vsyscall ()
(gdb) b subdiag                     #breakpoint set at entry to subdiag()
Breakpoint 1 at 0xb77683f3: file sd.c, line 3.
(gdb) continue                      #resume execution R (hitting enter-key twice)
Continuing.

Breakpoint 1, subdiag (m=0x92b9480, n=0x9482328, k=0x9482348, result=0x9817148)
    at sd.c:3
3          int nval = *n, kval = *k;
(gdb)
```

The C code is then executed as follows:

```
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
+ result=double(dim(m)[1]-k))

Breakpoint 1, subdiag (m=0x942f270, n=0x96c3328, k=0x96c3348, result=0x9a58148)
    at subdiag.c:46
46 if (*n < 1) error("n < 1\n");
```

The **GDB** session for debugging then continues as per usual. For a proper reference to understanding **GDB**:

Common GDB commands:

l	List code lines	n	Step to next statement	c	Continue
b	Set breakpoint	s	Step into function call	h	Help
r	Run/rerun	p	Print variable or expression	q	Quit

using R from Python

The **RPy** module in Python allows access to R from Python. It is often used in tandem with **NumPy** for the best experience. Loading **Rpy** from Python is achieved through executing the following code:

```
from rpy import *
```

The variable **r** is then loaded as a Python class instance.

The following illustrates the basic principles of running R from Python:

```
>>> r.hist(r.rnorm(100))
```

The subsequent action calls the **rnorm( )** function in R to product 100 standard normal variates in a histogram using the **hist( )** function in R. Note that the R names are prefixed with a **r.**; the Python wrappers for R functions are members of the class instance **r** in Python. However, the above call can produce noisy results within the *title of the graph* and the *x-axis label*. The latter is avoided as follows:

```
>>> r.hist(r.rnorm(100),main='',xlab='')
```

Problems will arise for R users when R and Python syntax experience collision. For example, calling the linear model function **lm( )** in R to predict variable **b** from variable **a**:

```
>>> a = [5,12,13]
>>> b = [10,28,30]
>>> lmout = r.lm('v2 ~ v1',data=r.data_frame(v1=a,v2=b))
```

The above execution in Python requires more syntax than if performed in an R environment. Notably, Python syntax does not make use of the **tilde ~** character to execute as a function of; the model was specified as a string. Additionally, a dataframe was needed to contain the data; created through the **data.frame( )** function. Note that the **period "."** within the function is applied as an **underscore "_"** in Python. Lastly, the columns in the dataframe were named **V1** and **V2** prior to using them in the model formula.

The output object is a Python dictionary (analog of R's list type):

```
>>> lmout
{'qr': {'pivot': [1, 2], 'qr': array([[ -1.73205081, -17.32050808],
[ 0.57735027, -6.164414 ],
[ 0.57735027, 0.78355007]]), 'qraux':
```

The attributes of the **lm( )** function differ in Python; the coefficients of the fitted regression line (**lmout$coefficients** in R) are presented as **lmout['coefficients']** in Python and accessed as follows:

```
>>> lmout['coefficients']
{'v1': 2.5263157894736841, '(Intercept)': -2.5964912280701729}
>>> lmout['coefficients']['v1']
2.5263157894736841
```

Additionally, R commands can be executed to work on variables in R's namespace through the **r( )** function. The following executes the **wireframe( )** function illustration presented previously in R; below in **RPy**:

```
>>> r.library('lattice')
>>> r.assign('a',a)                  #assign a copy variable from Python's namespace to R
>>> r.assign('b',b)                  #assign a copy variable from Python's namespace to R
>>> r('g <- expand.grid(a,b)')      #note the '.' in expand.grid; due to R namespace
>>> r('g$Var3 <- g$Var1^2 + g$Var1 * g$Var2')
>>> r('wireframe(Var3 ~ Var1+Var2,g)')
>>> r('plot(wireframe(Var3 ~ Var1+Var2,g))')  #plot() called; not automatically displayed
```

The above is useful in the context of multiple syntax collisions between Python and R.

parallel computing in R

With computational requirements and datasets growing as they have in the present, **parallel operation** in R has be developed amongst several available tools. In R, failed **parallel processing** mechanisms are sometimes discovered when certain programs actually run faster without **parallel processing**.

Given the example of a network graph (weblinks and social networks, etc.), **A** represents the ***adjacency matrix*** of the graph (e.g. **A[3,8]** is **1** or **0**, dependent on whether there is a link from **node 3** to **node 8**). For a given two vertices (two websites), the **manual outlinks** (outbound links common to two websites) become the problem's focus. The mean number of manual outlinks, averaged over all pairs of websites in the data is computed using the below for a **n-by-n** $\mathbb{R}^{n \times n}$ matrix:

```
sum = 0
for i = 0...n-1
  for j = i+1...n-1
    for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
mean = sum / (n*(n-1)/2)
```

The above code could be computationally expensive given the potential size of the dataset. The latter context is often dealt with by **chunking** the data into separate parts and processing in **multiple locations**.

The **snow** (Simple Network of Workstations) package in R is a simple and convenient form of **parallel processing** in R. The code for the manual outlinks problem above is presented as follows with **snow**:

```
# snow version of mutual links problem

mtl <- function(ichunk,m) {
  n <- ncol(m)
  matches <- 0
  for (i in ichunk) {
    if (i < n) {
      rowi <- m[i,]
      matches <- matches +
        sum(m[(i+1):n,] %*% rowi)
      }
   }
  matches
}

mutlinks <- function(cls,m) {
  n <- nrow(m)
  nc <- length(cls)
  # determine which worker gets which chunk of i
  options(warn=-1)
  ichunks <- split(1:n,1:nc)
  options(warn=0)
  counts <- clusterApply(cls,ichunks,mtl,m)
  do.call(sum,counts) / (n*(n-1)/2)
}
```

The code from the illustration above in applying the **snow** package runs as follows:

- … load code
- … load the **snow** library
- … form a **snow** cluster
- … set up ***adjacency matrix*** of focus
- … run code on the latter matrix on the formed **snow** cluster

Given the above code stored in ***SnowMutLinks.R***, the following commands execute for dual-core machines:

```
> source("SnowMutLinks.R")
> library(snow)
> cl <- makeCluster(type="SOCK",c("localhost","localhost"))
> testm <- matrix(sample(0:1,16,replace=T),nrow=4)
> mutlinks(cl,testm)
[1] 0.6666667
```

The above code starts two new R processes locally. Given the two new processes and the existing process where the code was previously entered, there are three total R processes. The origin R process forms a ***cluster*** in **snow** parlance (named **cl** above). The **snow** package uses parallel processing in the context of a ***scatter/gather*** paradigm:

… The origin process partitions the data into ***chunks*** and sends them to the newly created parallel process (***scatter phase****)*

… The newly created processes process their received ***chunks***

… The origin process collects the results from the created processes (***gather phase***) and combines them appropriately

The communication between the processes is established through ***network sockets*** (discussed previously):

```
> testm
   [,1] [,2] [,3] [,4]
[1,]  1   0    0   1
[2,]  0   0    0   0
[3,]  1   0    1   1
[4,]  0   1    0   1
```

The **test matrix** above shows row **1** has zero common outlinks with row **2**; row **1** has two outlinks in common with row **3**; Row **1** has one outlink in common with Row **4**; Row **2** has zero outlinks in common with any; Row **3** has one outlink in common with Row **4**. There are four total manual outlinks out of $4 \times \frac{3}{2} = 6$ pairs—the mean value of $\frac{4}{6} = 0.6666667$ as shown in the previous output.

Clusters can be made of any size depending on the available resources. Given **3 dual-core** machines named ***p28***, ***p29***, ***p30***, the following creates a ***six-process cluster*** using the **snow** package.

```
> cl6 <- makeCluster(type="SOCK",c("pc28","pc28","pc29","pc29","pc30","pc30"))
```

analyzing the snow code in R

To examine the **mutlinks( )** function, the number of **rows in the matrix** and the **number of processes** (aside from the origin process) within the **cluster** (line 17 and 18 above):

The created process is then determined as to which value of **i** in the **for i** loop will be handled. The **split( )** function is the appropriate use for the latter task; in a 4-row matric and a 2-process cluster:

```
> split(1:4,1:2)
$`1`
[1] 1 3

$`2`
[1] 2 4
```

The above set up has one R process work on the odd values of **i** and the other work on the even values of **i**. Potential warnings from the **split( )** function ("data length is not a multiple of split variable") are avoided through the **options( )** function.

The heavy lifting in the **mutlinks( )** function is performed where the **snow clusterApply( )** function is called. The latter calls the same specified function (**mtl( )** in this case) with specific arguments for each process and optional arguments that apply to all processes. The following is performed in line 23 of **mutlinks( )**:

… New Process 1 directed to call the **mtl( )** function with arguments **ichunks[[1]]** and **m**
… New Process 2 calls the **mtl( )** function with arguments **ichunks[[2]]** and **m**; and so one for all processes
… Each process performs their task and returns results to the Origin Process
… The Origin Process collects all results into an R list, assigned above to **counts**

All of the elements of the **count** variable are summed:

```
> sum(1:2,c(4,10))
[1] 17
```

Note the above call only applies to numeric vectors, therefore the **do.call( )** function is used for the **R list**. The vectors are extracted from **counts** and then passed through the **sum( )** function appropriately. Note the vectorized implementation in lines 9 and 10; opposed to using **for j** and **for k loops**.

obtainable speedup in R

In many parallel-processing applications, ***overhead*** exists (wasted time on noncomputational activity). There exists overhead in the form of the time needed to send the matrix from the origin program to the newly created processed. Additional overhead is encountered when sending the function **mtl( )** to the workers. Lastly, after the created processes complete their roles, overhead exists in the return to the origin process.

resorting to C in R

Another option is to write R code that parallels C/C++. In the context of multi-core machines, parallel computation is performed with ***threads***; analogous to the created processes through the **snow** package. The latter is constructed as a ***shared-memory*** system. All cores access the same RAM, changes made across threads are visible to the other threads.

The below example applies the **mutual outlinks** problem in R-callable code with the **OpenMP** package in R:

```
#include <omp.h>
#include <R.h>

int tot; // grand total of matches, over all threads

// processes row pairs (i,i+1), (i,i+2), ...
int procpairs(int i, int *m, int n)
{ int j,k,sum=0;
  for (j = i+1; j < n; j++) {
    for (k = 0; k < n; k++)
      // find m[i][k]*m[j][k] but remember R uses col-major order
      sum += m[n*k+i] * m[n*k+j];
  }
  return sum;
}
void mutlinks(int *m, int *n, double *mlmean)
{ int nval = *n;
  tot = 0;
```

(continued on the proceeding page)

```
19   #pragma omp parallel
20   { int i,mysum=0,
21     me = omp_get_thread_num(),
22     nth = omp_get_num_threads();
23     // in checking all (i,j) pairs, partition the work according to i;
24     // this thread me will handle all i that equal me mod nth
25     for (i = me; i < nval; i += nth) {
26       mysum += procpairs(i,m,nval);
27     }
28     #pragma omp atomic
29     tot += mysum;
30   }
31   int divisor = nval * (nval-1) / 2;
32   *mlmean = ((float) tot)/divisor;
33 }
```

The **OpenMC** library is linked through the **–fopenmp** and **–lgomp** option below (source file ***romp.c***):

```
gcc -std=gnu99 -fopenmp -I/usr/share/R/include -fpic -g -O2 -c romp.c -o romp.o
gcc -std=gnu99 -shared -o romp.so romp.o -L/usr/lib/R/lib -lR -lgomp
```

The code is tested in R as follows:

```
> dyn.load("romp.so")
> Sys.setenv(OMP_NUM_THREADS=4)
> n <- 1000
> m <- matrix(sample(0:1,n^2,replace=T),nrow=n)
> system.time(z <- .C("mutlinks",as.integer(m),as.integer(n),result=double(1)))
  user system elapsed
  0.830 0.000 0.218

> z$result
[1] 249.9471
```

Threads can be specified in **Open MP** through the operating system variable **OMP_NUM_THREADS**. Operating system variables can be set in R with the **Sys.setenv( )** function (set to 4 threads above). Additionally, R has other ***byte-compilation*** functions such as **cmpfun( )**.

The **OpenMC** package code is compiled in C, with the addition of ***pragmas***, instructing the compiler to insert library code to perform **OpenMP** operations (threads activated in line 21). Note the importance of variable scope in the above code. The **mysum** variable in line 21 is for each thread to maintain its own sum. Line 4 contains the global variable **tot** which is shared amongst all threads. Each thread contributes to the total on line 30.

The **nval** variable on line 18 is also a global variable (although a local variable in terms of C) to all threads. An additional ***pragma*** (**atomic**) is located on line 29 and is applied to line 30 (opposed to the entire block). The **atomic** ***pragma*** serves to avoid a ***race condition***; where two threads are updating a variable synonymously.

The origin thread executes in lines 18, 19, and within the **.C( )** function in R that calls the **mutlinks( )** function. The origin thread becomes idle in line 21 when the new processes are activated; until they finish in line 31 (the origin process then reactivates).

The **procpairs()** function accesses matrix **m** as a one-dimensional array (common in C; recalling the difference of reading matrices column by column in R and row-wise in C) indexed at 0 (R indexes at 1).

The result is the multiplication on line 12. The factors are the **(k,i)** and **(k,j)** elements of **m** in the C code, which are factors **(i+1,k+1)** and **(j+1,k+1)** elements in the R code.

other OpenMP programs in R

A ***barrier*** in parallel processing refers to the line of code where threads interact:

```
#pragma omp barrier
```

Execution is suspended upon reaching a ***barrier*** until all threads catch up (useful in iterative algorithms). The above is an ***explicit barrier***, while other pragmas have ***implicit barriers*** within the code blocks (***single*** and ***parallel***; an implied barrier is located on line 31, causing the origin process to remain dormant).

A block that follows the ***omp critical pragma*** is considered a ***critical section***, where threads are executed individually. The ***omp critical pragma*** functions as the ***atomic pragma*** discussed above with the exception of the latter being limited to a single statement:

```
#pragma omp critical
{
  // place one or more statements here
}
```

The block of code following an ***omp single pragma*** is executed by a single thread:

```
#pragma omp single
{
  // place one or more statements here
}
```

The above is useful for initializing sum variables amongst threads; preventing simultaneous thread activity.

gpu programming in R

An additional shared-memory parallel hardware includes ***graphics processing units*** (**GPU**). The brief context in application is to write the R code interfaced to parallel C; opposed to writing parallel R code. **GPU**s follow the ***threads*** model at a greater magnitude (as much as 100 cores). Consequentially, multiple threads are capable of running within the same block. Access to **GPU**s begin on the **CPU** serving as the ***host***. The code is executed on the **GPU** (***device***) by transferring the data to the ***device***.

general performance considerations in R

It is important to understand the context of ***overhead*** inducing aspects of the code being executed in R. An example is the used of **shared-memory machines** as illustrated previously. Because the latter process requires waiting on each thread at a time, ***overhead*** inevitably occurs. Additionally, cores might contain ***caches*** that require ***overhead*** to keep the ***caches*** consistent amongst cores. **GPU**s can suffer from ***latency***, as the delay in time when the initial information bits are received from memory. ***Overhead*** also exists between transfers from the ***host*** and ***devices***.

To illustrate the topic of **static** and **dynamic task assignment**, the following code is from **OpenMP** above:

```
for (i = me; i < nval; i += nth) {
  mysum += procpairs(i,m,nval);
}
```

The variable **me** represents the thread number, keeping threads from overlapping on values of variable **i**. The preassignment of tasks for each thread is what is referred to as **static task assignment**.

An alternative approach to the above illustration is presented on the proceeding page (alliteration intended) by revising the original **for loop**:

```
int nexti = 0; // global variable
...
for ( ; myi < n; ) { // revised "for" loop
  #pragma omp critical
  {
    nexti += 1;
    myi = nexti;
}
if (myi < n) {
  mysum += procpairs(myi,m,nval);
  ...
  }
}
...
```

The above approach is what is referred to as a **dynamic task assignment**, where the values of **i** which are handled by the threads is not predetermined as before. It can be assumed that using the **dynamic task assigning** approach will lead to efficiency gains. An example would be if the completion times of predetermined threads do not finish as expected, causing a ***load balance*** problem. Conversely, the above could cause ***caching*** issues that render **dynamic task assignment** much slower in certain context.

Conclusively, it is appropriate to use **static task assignments** in most circumstances cited below:

> The standard deviation of the sum of independent, identically distributed random variables, divided by the mean of that sum, goes to zero as the number of terms goes to infinity. In other words, sums are approximately constant. This has a direct implication for load-balancing concerns: Since the total work time for a thread in static assignment is the sum of its individual task times, that total work time will be approximately constant; there will be very little variation from thread to thread, [finishing together]

Another issue applies to the following algorithm from the **mutual outlinks** illustration previously discussed:

```
sum = 0
for i = 0...n-1
  for j = i+1...n-1
    for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
mean = sum / (n*(n-1)/2)
```

Given that **n = 10000** with **4-threads**, there are multiple approaches to partitioning the **for i loop**. A massive inefficient would result from assigning the first thread to have the **i** values from **0-2499** and the second thread to handle **2500-4999**, etc; the result being a large ***load imbalance*** (a thread processing a given value of **i** performs a proportional amount of work to **n-i**). The latter is reasoning behind the staggered assignment of threads in the **mutual outlinks** example. **Static assignment** might ultimately require more preparation and analysis, but can be avoided through best practices or methods such as random assignment to threads.

debugging parllel code in R

Some parallel R packages, such as **Rmpi**, **snow**, **foreach**, etc. (at the publishing of Norman Matloff's book), do not create terminal windows for individual processes; removing the option to leverage R's debugger. An available solution would require debugging an underlying function; by setting up artificial values and applying R's ordinary debugging mechanisms. Assuming a bug exists in an actual argument itself, the process becomes more difficult considering trace information is not readily returned as the **print( )** and **message( )** functions might not apply to all parallel R packages. The **cat( )** function should remain available for these particular circumstances in using parallel R packages; writing the contents to an actual file.