15

INTERFACING R TO OTHER LANGUAGES

R is a great language, but it can't do everything well. Thus, it is sometimes desirable to call code written in other languages from R. Conversely, when working in other great languages, you may encounter tasks that could be better done in R.

R interfaces have been developed for a number of other languages, from ubiquitous languages like C to esoteric ones like the Yacas computer algebra system. This chapter will cover two interfaces: one for calling C/C++ from R and the other for calling R from Python.

15.1 Writing C/C++ Functions to Be Called from R

You may wish to write your own C/C++ functions to be called from R. Typically, the goal is performance enhancement, since C/C++ code may run much faster than R, even if you use vectorization and other R optimization techniques to speed things up.

Another possible goal in dropping down to the C/C++ level is specialized I/O. For example, R uses the TCP protocol in layer 3 of the standard Internet communication system, but UDP can be faster in some settings.

To work in UDP, you need C/C++, which requires an interface to R for those languages.

R actually offers two C/C++ interfaces via the functions `.C()` and `.Call()`. The latter is more versatile but requires some knowledge of R's internal structures, so we'll stick with `.C()` here.

15.1.1 Some R-to-C/C++ Preliminaries

In C, two-dimensional arrays are stored in row-major order, in contrast to R's column-major order. For instance, if you have a 3-by-4 array, the element in the second row and second column is element number 5 of the array when viewed linearly, since there are three elements in the first column and this is the second element in the second column. Also keep in mind that C subscripts begin at 0, rather than at 1, as with R.

All the arguments passed from R to C are received by C as pointers. Note that the C function itself must return `void`. Values that you would ordinarily return must be communicated through the function's arguments, such as `result` in the following example.

15.1.2 Example: Extracting Subdiagonals from a Square Matrix

Here, we will write C code to extract subdiagonals from a square matrix. (Thanks to my former graduate assistant, Min-Yu Huang, who wrote an earlier version of this function.) Here's the code for the file *sd.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 variable `stride` alludes to a concept from the parallel-processing community. Say we have a matrix in 1,000 columns and our C code is looping through all the elements in a given column, from top to bottom. Again, since C uses row-major order, consecutive elements in the column are 1,000 elements apart from each other if the matrix is viewed as one long vector.

Here, we would say that we are traversing that long vector with a stride of 1,000—that is, accessing every thousandth element.

15.1.3 Compiling and Running Code

You compile your code using R. For example, in a Linux terminal window, we could compile our file like this:

```
% 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
```

This would produce the dynamic shared library file *sd.so*.

Note that R has reported how it invoked GCC in the output of the example. You can also run these commands by hand if you have special requirements, such as special libraries to be linked in. Also note that the locations of the *include* and *lib* directories may be system-dependent.

NOTE *GCC is easily downloadable for Linux systems. For Windows, it is included in Cygwin, an open source package available from* http://www.cygwin.com/.

We can then load our library into R and call our C function like this:

```
> 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),
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
```

For convenience here, we've given the name `result` to both the formal argument (in the C code) and the actual argument (in the R code). Note that we needed to allocate space for `result` in our R code.

As you can see from the example, the return value takes on the form of a list consisting of the arguments in the R call. In this case, the call had four arguments (in addition to the function name), so the returned list has four components. Typically, some of the arguments will be changed during execution of the C code, as was the case here with `result`.

15.1.4 Debugging R/C Code

Chapter 13 discussed a number of tools and methods for debugging R code. However, the R/C interface presents an extra challenge. The problem in using a debugging tool such as GDB here is that you must first apply it to R itself.

The following is a walk-through of the R/C debugging steps using GDB on our previous *sd.c* code as the example.

```
$ R -d gdb
GNU gdb 6.8-debian
...
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")
>   # hit ctrl-c here
Program received signal SIGINT, Interrupt.
0xb7ffa430 in __kernel_vsyscall ()
(gdb) b subdiag
Breakpoint 1 at 0xb77683f3: file sd.c, line 3.
(gdb) continue
Continuing.

Breakpoint 1, subdiag (m=0x92b9480, n=0x9482328, k=0x9482348, result=0x9817148)
    at sd.c:3
3          int nval = *n, kval = *k;
(gdb)
```

So, what happened in this debugging session?

1. We launched the debugger, GDB, with R loaded into it, from a command line in a terminal window:

   ```
   R -d gdb
   ```

2. We told GDB to run R:

   ```
   (gdb) run
   ```

3. We loaded our compiled C code into R as usual:

   ```
   > dyn.load("sd.so")
   ```

4. We hit the CTRL-C interrupt key pair to pause R and put us back at the GDB prompt.
5. We set a breakpoint at the entry to `subdiag()`:

   ```
   (gdb) b subdiag
   ```

6. We told GDB to resume executing R (we needed to hit the ENTER key a second time in order to get the R prompt):

```
(gdb) continue
```

We then executed our C code:

```
> 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");
```

At this point, we can use GDB to debug as usual. If you're not familiar with GDB, you may want to try one of the many quick tutorials on the Web. Table 15-1 lists some of the most useful commands.

Table 15-1: Common GDB Commands

Command	Description
`l`	List code lines
`b`	Set breakpoint
`r`	Run/rerun
`n`	Step to next statement
`s`	Step into function call
`p`	Print variable or expression
`c`	Continue
`h`	Help
`q`	Quit

15.1.5 Extended Example: Prediction of Discrete-Valued Time Series

Recall our example in Section 2.5.2 where we observed 0- and 1-valued data, one per time period, and attempted to predict the value in any period from the previous k values, using majority rule. We developed two competing functions for the job, `preda()` and `predb()`, as follows:

```
# prediction in discrete time series; 0s and 1s; use k consecutive
# observations to predict the next, using majority rule; calculate the
# error rate
preda <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   # the vector pred will contain our predicted values
   pred <- vector(length=n-k)
```

```
   for (i in 1:(n-k)) {
      if (sum(x[i:(i+(k-1))]) >= k2) pred[i] <- 1 else pred[i] <- 0
   }
   return(mean(abs(pred-x[(k+1):n])))
}

predb <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   pred <- vector(length=n-k)
   sm <- sum(x[1:k])
   if (sm >= k2) pred[1] <- 1 else pred[1] <- 0
   if (n-k >= 2) {
      for (i in 2:(n-k)) {
         sm <- sm + x[i+k-1] - x[i-1]
         if (sm >= k2) pred[i] <- 1 else pred[i] <- 0
      }
   }
   return(mean(abs(pred-x[(k+1):n])))
}
```

Since the latter avoids duplicate computation, we speculated it would be faster. Now is the time to check that.

```
> y <- sample(0:1,100000,replace=T)
> system.time(preda(y,1000))
   user  system elapsed
  3.816   0.016   3.873
> system.time(predb(y,1000))
   user  system elapsed
  1.392   0.008   1.427
```

Hey, not bad! That's quite an improvement.

However, you should always ask whether R already has a fine-tuned function that will suit your needs. Since we're basically computing a moving average, we might try the `filter()` function, with a constant coefficient vector, as follows:

```
predc <- function(x,k) {
   n <- length(x)
   f <- filter(x,rep(1,k),sides=1)[k:(n-1)]
   k2 <- k/2
   pred <- as.integer(f >= k2)
   return(mean(abs(pred-x[(k+1):n])))
}
```

That's even more compact than our first version. But it's a lot harder to read, and for reasons we will explore soon, it may not be so fast. Let's check.

```
> system.time(predc(y,1000))
   user  system elapsed
  3.872   0.016   3.945
```

Well, our second version remains the champion so far. This actually should be expected, as a look at the source code shows. Typing the following shows the source for that function:

```
> filter
```

This reveals (not shown here) that `filter1()` is called. The latter is written in C, which should give us some speedup, but it still suffers from the duplicate computation problem—hence the slowness.

So, let's write our own C code.

```
#include <R.h>

void predd(int *x, int *n, int *k, double *errrate)
{
   int nval = *n, kval = *k, nk = nval - kval, i;
   int sm = 0;  // moving sum
   int errs = 0;  // error count
   int pred;  // predicted value
   double k2 = kval/2.0;
   // initialize by computing the initial window
   for (i = 0; i < kval; i++) sm += x[i];
   if (sm >= k2) pred = 1; else pred = 0;
   errs = abs(pred-x[kval]);
   for (i = 1; i < nk; i++) {
      sm = sm + x[i+kval-1] - x[i-1];
      if (sm >= k2) pred = 1; else pred = 0;
      errs += abs(pred-x[i+kval]);
   }
   *errrate = (double) errs / nk;
}
```

This is basically `predb()` from before, "hand translated" into C. Let's see if it will outdo `predb()`.

```
> system.time(.C("predd",as.integer(y),as.integer(length(y)),as.integer(1000),
+   errrate=double(1)))
   user  system elapsed
  0.004   0.000   0.003
```

The speedup is breathtaking.

You can see that writing certain functions in C can be worth the effort. This is especially true for functions that involve iteration, as R's own iteration constructs, such as `for()`, are slow.

15.2 Using R from Python

Python is an elegant and powerful language, but it lacks built-in facilities for statistical and data manipulation, two areas in which R excels. This section demonstrates how to call R from Python, using RPy, one of the most popular interfaces between the two languages.

15.2.1 Installing RPy

RPy is a Python module that allows access to R from Python. For extra efficiency, it can be used in conjunction with NumPy.

You can build the module from the source, available from *http://rpy.sourceforge.net*, or download a prebuilt version. If you are running Ubuntu, simply type this:

```
sudo apt-get install python-rpy
```

To load RPy from Python (whether in Python interactive mode or from code), execute the following:

```
from rpy import *
```

This will load a variable `r`, which is a Python class instance.

15.2.2 RPy Syntax

Running R from Python is in principle quite simple. Here is an example of a command you might run from the >>> Python prompt:

```
>>> r.hist(r.rnorm(100))
```

This will call the R function `rnorm()` to produce 100 standard normal variates and then input those values into R's histogram function, `hist()`.

As you can see, R names are prefixed by `r.`, reflecting the fact that Python wrappers for R functions are members of the class instance `r`.

The preceding code will, if not refined, produce ugly output, with your (possibly voluminous!) data appearing as the graph title and the *x*-axis label. You can avoid this by supplying a title and label, as in this example:

```
>>> r.hist(r.rnorm(100),main='',xlab='')
```

RPy syntax is sometimes less simple than these examples would lead you to believe. The problem is that R and Python syntax may clash. For instance,

consider a call to the R linear model function `lm()`. In our example, we will predict `b` from `a`.

```
>>> a = [5,12,13]
>>> b = [10,28,30]
>>> lmout = r.lm('v2 ~ v1',data=r.data_frame(v1=a,v2=b))
```

This is somewhat more complex than it would have been if done directly in R. What are the issues here?

First, since Python syntax does not include the tilde character, we needed to specify the model formula via a string. Since this is done in R anyway, this is not a major departure.

Second, we needed a data frame to contain our data. We created one using R's `data.frame()` function. In order to form a period in an R function name, we need to use an underscore on the Python end. Thus we called `r.data_frame()`. Note that in this call, we named the columns of our data frame `v1` and `v2` and then used these in our model formula.

The output object is a Python dictionary (analog of R's `list` type), as you can see here (in part):

```
>>> lmout
{'qr': {'pivot': [1, 2], 'qr': array([[ -1.73205081, -17.32050808],
       [  0.57735027,  -6.164414  ],
       [  0.57735027,   0.78355007]]), 'qraux':
```

You should recognize the various attributes of `lm()` objects here. For example, the coefficients of the fitted regression line, which would be contained in `lmout$coefficients` if this were done in R, are here in Python as `lmout['coefficients']`. So, you can access those coefficients accordingly, for example like this:

```
>>> lmout['coefficients']
{'v1': 2.5263157894736841, '(Intercept)': -2.5964912280701729}
>>> lmout['coefficients']['v1']
2.5263157894736841
```

You can also submit R commands to work on variables in R's namespace, using the function `r()`. This is convenient if there are many syntax clashes. Here is how we could run the `wireframe()` example in Section 12.4 in RPy:

```
>>> r.library('lattice')
>>> r.assign('a',a)
>>> r.assign('b',b)
>>> r('g <- expand.grid(a,b)')
>>> r('g$Var3 <- g$Var1^2 + g$Var1 * g$Var2')
>>> r('wireframe(Var3 ~ Var1+Var2,g)')
>>> r('plot(wireframe(Var3 ~ Var1+Var2,g))')
```

First, we used `r.assign()` to copy a variable from Python's namespace to R's. We then ran `expand.grid()` (with a period in the name instead of an underscore, since we are running in R's namespace), assigning the result to `g`. Again, the latter is in R's namespace. Note that the call to `wireframe()` did not automatically display the plot, so we needed to call `plot()`.

The official documentation for RPy is at *http://rpy.sourceforge.net/rpy/doc/rpy.pdf*. Also, you can find a useful presentation, "RPy—R from Python," at *http://www.daimi.au.dk/~besen/TBiB2007/lecture-notes/rpy.html*.

16

PARALLEL R

Since many R users have very large computational needs, various tools for some kind of parallel operation of R have been devised. This chapter is devoted to parallel R.

Many a novice in parallel processing has, with great anticipation, written parallel code for some application only to find that the parallel version actually ran more slowly than the serial one. For reasons to be discussed in this chapter, this problem is especially acute with R.

Accordingly, understanding the nature of parallel-processing hardware and software is crucial to success in the parallel world. These issues will be discussed here in the context of common platforms for parallel R.

We'll start with a few code examples and then move to general performance issues.

16.1 The Mutual Outlinks Problem

Consider a network graph of some kind, such as web links or links in a social network. Let `A` be the *adjacency matrix* of the graph, meaning that, say, `A[3,8]` is 1 or 0, depending on whether there is a link from node 3 to node 8.

For any two vertices, say any two websites, we might be interested in mutual outlinks—that is, outbound links that are common to two sites. Suppose that we want to find the mean number of mutual outlinks, averaged

over all pairs of websites in our data set. This mean can be found using the following outline, for an *n*-by-*n* matrix:

```
1  sum = 0
2  for i = 0...n-1
3     for j = i+1...n-1
4        for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
5  mean = sum / (n*(n-1)/2)
```

Given that our graph could contain thousands—even millions—of websites, our task could entail quite large amounts of computation. A common approach to dealing with this problem is to divide the computation into smaller chunks and then process each of the chunks simultaneously, say on separate computers.

Let's say that we have two computers at our disposal. We might have one computer handle all the odd values of `i` in the `for i` loop in line 2 and have the second computer handle the even values. Or, since dual-core computers are fairly standard these days, we could take this same approach on a single computer. This may sound simple, but a number of major issues can arise, as you'll learn in this chapter.

16.2 Introducing the snow Package

Luke Tierney's `snow` (Simple Network of Workstations) package, available from the CRAN R code repository, is arguably the simplest, easiest-to-use form of parallel R and one of the most popular.

NOTE *The CRAN Task View page on parallel R,* http://cran.r-project.org/web/views/HighPerformanceComputing.html*, has a fairly up-to-date list of available parallel R packages.*

To see how `snow` works, here's code for the mutual outlinks problem described in the previous section:

```
1  # snow version of mutual links problem
2
3  mtl <- function(ichunk,m) {
4     n <- ncol(m)
5     matches <- 0
6     for (i in ichunk) {
7        if (i < n) {
8           rowi <- m[i,]
9           matches <- matches +
10             sum(m[(i+1):n,] %*% rowi)
11       }
12    }
13    matches
14 }
```

```
15
16 mutlinks <- function(cls,m) {
17    n <- nrow(m)
18    nc <- length(cls)
19    # determine which worker gets which chunk of i
20    options(warn=-1)
21    ichunks <- split(1:n,1:nc)
22    options(warn=0)
23    counts <- clusterApply(cls,ichunks,mtl,m)
24    do.call(sum,counts) / (n*(n-1)/2)
25 }
```

Suppose we have this code in the file *SnowMutLinks.R*. Let's first discuss how to run it.

16.2.1 Running snow Code

Running the above `snow` code involves the following steps:

1. Load the code.
2. Load the `snow` library.
3. Form a `snow` cluster.
4. Set up the adjacency matrix of interest.
5. Run your code on that matrix on the cluster you formed.

Assuming we are running on a dual-core machine, we issue the following commands to R:

```
> 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
```

Here, we are instructing `snow` to start two new R processes on our machine (`localhost` is a standard network name for the local machine), which I will refer to here as *workers*. I'll refer to the original R process—the one in which we type the preceding commands—as the *manager*. So, at this point, three instances of R will be running on the machine (visible by running the `ps` command if you are in a Linux environment, for example).

The workers form a *cluster* in `snow` parlance, which we have named `cl`. The `snow` package uses what is known in the parallel-processing world as a *scatter/gather* paradigm, which works as follows:

1. The manager partitions the data into chunks and parcels them out to the workers (scatter phase).

2. The workers process their chunks.
3. The manager collects the results from the workers (gather phase) and combines them as appropriate to the application.

We have specified that communication between the manager and workers will be via network sockets (covered in Chapter 10).

Here's a test matrix to check the code:

```
> 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
```

Row 1 has zero outlinks in common with row 2, two in common with row 3, and one in common with row 4. Row 2 has zero outlinks in common with the rest, but row 3 has one in common with row 4. That is a total of four mutual outlinks out of $4 \times 3/2 = 6$ pairs—hence, the mean value of $4/6 = 0.6666667$, as you saw earlier.

You can make clusters of any size, as long as you have the machines. In my department, for instance, I have machines whose network names are `pc28`, `pc29`, and `pc30`. Each machine is dual core, so I could create a six-worker cluster as follows:

```
> cl6 <- makeCluster(type="SOCK",c("pc28","pc28","pc29","pc29","pc30","pc30"))
```

16.2.2 Analyzing the snow Code

Now let's see how the `mutlinks()` function works. First, we sense how many rows the matrix `m` has, in line 17, and the number of workers in our cluster, in line 18.

Next, we need to determine which worker will handle which values of `i` in the `for i` loop in our outline code shown earlier in Section 16.1. R's `split()` function is well suited for this. For instance, in the case of a 4-row matrix and a 2-worker cluster, that call produces the following:

```
> split(1:4,1:2)
$`1`
[1] 1 3

$`2`
[1] 2 4
```

An R list is returned whose first element is the vector (1,3) and the second is (2,4). This will set up having one R process work on the odd values of `i` and the other work on the even values, as we discussed earlier. We ward off the

warnings that split() would give us ("data length is not a multiple of split variable") by calling options().

The real work is done in line 23, where we call the snow function clusterApply(). This function initiates a call to the same specified function (mtl() here), with some arguments specific to each worker and some optional arguments common to all. So, here's what the call in line 23 does:

1. Worker 1 will be directed to call the function mtl() with the arguments ichunks[[1]] and m.
2. Worker 2 will call mtl() with the arguments ichunks[[2]] and m, and so on for all workers.
3. Each worker will perform its assigned task and then return the result to the manager.
4. The manager will collect all such results into an R list, which we have assigned here to counts.

At this point, we merely need to sum all the elements of counts. Well, I shouldn't say "merely," because there is a little wrinkle to iron out in line 24.

R's sum() function is capable of acting on several vector arguments, like this:

```
> sum(1:2,c(4,10))
[1] 17
```

But here, counts is an R list, not a (numeric) vector. So we rely on do.call() to extract the vectors from counts, and then we call sum() on them.

Note lines 9 and 10. As you know, in R, we try to vectorize our computation wherever possible for better performance. By casting things in matrix-times-vector terms, we replace the for j and for k loops in the outline in Section 16.1 by a single vector-based expression.

16.2.3 How Much Speedup Can Be Attained?

I tried this code on a 1000-by-1000 matrix m1000. I first ran it on a 4-worker cluster and then on a 12-worker cluster. In principle, I should have had speedups of 4 and 12, respectively. But the actual elapsed times were 6.2 seconds and 5.0 seconds. Compare these figures to the 16.9 seconds runtime in nonparallel form. (The latter consisted of the call mtl(1:1000,m1000).) So, I attained a speedup of about 2.7 instead of a theoretical 4.0 for a 4-worker cluster and 3.4 rather than 12.0 on the 12-node system. (Note that some timing variation occurs from run to run.) What went wrong?

In almost any parallel-processing application, you encounter *overhead*, or "wasted" time spent on noncomputational activity. In our example, there is overhead in the form of the time needed to send our matrix from the manager to the workers. We also encountered a bit of overhead in sending the function mtl() itself to the workers. And when the workers finish their tasks, returning their results to the manager causes some overhead, too. We'll

discuss this in detail when we talk about general performance considerations in in Section 16.4.1.

16.2.4 Extended Example: K-Means Clustering

To learn more about the capabilities of snow, we'll look at another example, this one involving k-means clustering (KMC).

KMC is a technique for exporatory data analysis. In looking at scatter plots of your data, you may have the perception that the observations tend to cluster into groups, and KMC is a method for finding such groups. The output consists of the centroids of the groups.

The following is an outline of the algorithm:

```
for iter = 1,2,...,niters
   set vector and count totals to 0
   for i = 1,...,nrow(m)
      set j = index of the closest group center to m[i,]
      add m[i,] to the vector total for group j, v[j]
      add 1 to the count total for group j, c[j]
   for j = 1,...,ngrps
      set new center of group j = v[j] / c[j]
```

Here, we specify niters iterations, with initcenters as our initial guesses for the centers of the groups. Our data is in the matrix m, and there are ngrps groups.

The following is the snow code to compute KMC in parallel:

```
# snow version of k-means clustering problem

library(snow)

# returns distances from x to each vector in y;
# here x is a single vector and y is a bunch of them;
# define distance between 2 points to be the sum of the absolute values
# of their componentwise differences; e.g., distance between (5,4.2) and
# (3,5.6) is 2 + 1.4 = 3.4
dst <- function(x,y) {
   tmpmat <- matrix(abs(x-y),byrow=T,ncol=length(x)) # note recycling
   rowSums(tmpmat)
}

# will check this worker's mchunk matrix against currctrs, the current
# centers of the groups, returning a matrix; row j of the matrix will
# consist of the vector sum of the points in mchunk closest to jth
# current center, and the count of such points
findnewgrps <- function(currctrs) {
   ngrps <- nrow(currctrs)
   spacedim <- ncol(currctrs) # what dimension space are we in?
```

```
22    # set up the return matrix
23    sumcounts <- matrix(rep(0,ngrps*(spacedim+1)),nrow=ngrps)
24    for (i in 1:nrow(mchunk)) {
25       dsts <- dst(mchunk[i,],t(currctrs))
26       j <- which.min(dsts)
27       sumcounts[j,] <- sumcounts[j,] + c(mchunk[i,],1)
28    }
29    sumcounts
30 }
31
32 parkm <- function(cls,m,niters,initcenters) {
33    n <- nrow(m)
34    spacedim <- ncol(m) # what dimension space are we in?
35    # determine which worker gets which chunk of rows of m
36    options(warn=-1)
37    ichunks <- split(1:n,1:length(cls))
38    options(warn=0)
39    # form row chunks
40    mchunks <- lapply(ichunks,function(ichunk) m[ichunk,])
41    mcf <- function(mchunk) mchunk <<- mchunk
42    # send row chunks to workers; each chunk will be a global variable at
43    # the worker, named mchunk
44    invisible(clusterApply(cls,mchunks,mcf))
45    # send dst() to workers
46    clusterExport(cls,"dst")
47    # start iterations
48    centers <- initcenters
49    for (i in 1:niters) {
50       sumcounts <- clusterCall(cls,findnewgrps,centers)
51       tmp <- Reduce("+",sumcounts)
52       centers <- tmp[,1:spacedim] / tmp[,spacedim+1]
53       # if a group is empty, let's set its center to 0s
54       centers[is.nan(centers)] <- 0
55    }
56    centers
57 }
```

The code here is largely similar to our earlier mutual outlinks example. However, there are a couple of new `snow` calls and a different kind of usage of an old call.

Let's start with lines 39 through 44. Since our matrix `m` does not change from one iteration to the next, we definitely do not want to resend it to the workers repeatedly, exacerbating the overhead problem. Thus, first we need to send each worker its assigned chunk of `m`, just once. This is done in line 44 via `snow`'s `clusterApply()` function, which we used earlier but need to get creative with here. In line 41, we define the function `mcf()`, which will, running

on a worker, accept the worker's chunk from the manager and then keep it as a global variable `mchunk` on the worker.

Line 46 makes use of a new `snow` function, `clusterExport()`, whose job it is to make copies of the manager's global variables at the workers. The variable in question here is actually a function, `dst()`. Here is why we need to send it separately: The call in line 50 will send the function `findnewgrps()` to the workers, but although that function calls `dst()`, `snow` will not know to send the latter as well. Therefore we send it ourselves.

Line 50 itself uses another new `snow` call, `clusterCall()`. This instructs each worker to call `findnewgrps()`, with `centers` as argument.

Recall that each worker has a different matrix chunk, so this call will work on different data for each worker. This once again brings up the controversy regarding the use of global variables, discussed in Section 7.8.4. Some software developers may be troubled by the use of a hidden argument in `findnewgrps()`. On the other hand, as mentioned earlier, using `mchunk` as an argument would mean sending it to the workers repeatedly, compromising performance.

Finally, take a look at line 51. The `snow` function `clusterApply()` always returns an R list. In this case, the return value is in `sumcounts`, each element of which is a matrix. We need to sum the matrices, producing a totals matrix. Using R's `sum()` function wouldn't work, as it would total all the elements of the matrices into a single number. Matrix addition is what we need.

Calling R's `Reduce()` function will do the matrix addition. Recall that any arithmetic operation in R is implemented as a function; in this case, it is implemented as the function `"+"`. The recall to `Reduce()` then successively applies `"+"` to the elements of the list `sumcounts`. Of course, we could just write a loop to do this, but using `Reduce()` may give us a small performance boost.

16.3 Resorting to C

As you've seen, using parallel R may greatly speed up your R code. This allows you to retain the convenience and expressive power of R, while still ameliorating large runtimes in big applications. If the parallelized R gives you sufficiently good performance, then all is well.

Nevertheless, parallel R is still R and thus still subject to the performance issues covered in Chapter 14. Recall that one solution offered in that chapter was to write a performance-critical portion of your code in C and then call that code from your main R program. (The references to C here mean C or C++.) We will explore this from a parallel-processing viewpoint. Here, instead of writing parallel R, we write ordinary R code that calls parallel C. (I assume a knowledge of C.)

16.3.1 Using Multicore Machines

The C code covered here runs only on multicore systems, so we must discuss the nature of such systems.

You are probably familiar with dual-core machines. Any computer includes a CPU, which is the part that actually runs your program. In essence, a dual-core machine has two CPUs, a quad-core system has four, and so on. With multiple cores, you can do parallel computation!

This parallel computation is done with *threads*, which are analogous to `snow`'s workers. In computationally intensive applications, you generally set up as many threads as there are cores, for example two threads in a dual-core machine. Ideally, these threads run simultaneously, though overhead issues do arise, as will be explained when we look at general performance issues in Section 16.4.1.

If your machine has multiple cores, it is structured as a *shared-memory* system. All cores access the same RAM. The shared nature of the memory makes communication between the cores easy to program. If a thread writes to a memory location, the change is visible to the other threads, without the programmer needing to insert code to make that happen.

16.3.2 Extended Example: Mutual Outlinks Problem in OpenMP

OpenMP is a very popular package for programming on multicore machines. To see how it works, here is the mutual outlinks example again, this time in R-callable OpenMP code:

```
#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;
   #pragma omp parallel
   {  int i,mysum=0,
          me = omp_get_thread_num(),
          nth = omp_get_num_threads();
      // in checking all (i,j) pairs, partition the work according to i;
      // this thread me will handle all i that equal me mod nth
      for (i = me; i < nval; i += nth) {
```

```
            mysum += procpairs(i,m,nval);
        }
        #pragma omp atomic
        tot += mysum;
    }
    int divisor = nval * (nval-1) / 2;
    *mlmean = ((float) tot)/divisor;
}
```

16.3.3 Running the OpenMP Code

Again, compilation follows the recipe in Chapter 15. We do need to link in the OpenMP library, though, by using the `-fopenmp` and `-lgomp` options. Suppose our source file is *romp.c*. Then we use the following commands to run the code:

```
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
```

Here's an R test:

```
> 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
```

The typical way to specify the number of threads in OpenMP is through an operating system environment variable, `OMP_NUM_THREADS`. R is capable of setting operating system environment variables with the `Sys.setenv()` function. Here, I set the number of threads to 4, because I was running on a quad-core machine.

Note the runtime—only 0.2 seconds! This compares to the 5.0-second time we saw earlier for a 12-node `snow` system. This might be surprising to some readers, as our code in the `snow` version was vectorized to a fair degree, as mentioned earlier. Vectorizing is good, but again, R has many hidden sources of overhead, so C might do even better.

NOTE *I tried R's new byte-compilation function `cmpfun()`, but `mtl()` actually became slower.*

Thus, if you are willing to write part of your code in parallel C, dramatic speedups may be possible.

16.3.4 OpenMP Code Analysis

OpenMP code is C, with the addition of *pragmas* that instruct the compiler to insert some library code to perform OpenMP operations. Look at line 20, for instance. When execution reaches this point, the threads will be activated. Each thread then executes the block that follows—lines 21 through 31—in parallel.

A key point is variable scope. All the variables within the block starting on line 21 are local to their specific threads. For example, we've named the total variable in line 21 `mysum` because each thread will maintain its own sum. By contrast, the global variable `tot` on line 4 is held in common by all the threads. Each thread makes its contribution to that grand total on line 30.

But even the variable `nval` on line 18 is held in common with all the threads (during the execution of `mutlinks()`), as it is declared outside the block beginning on line 21. So, even though it is a local variable in terms of C scope, it is global to all the threads. Indeed, we could have declared `tot` on that line, too. It needs to be shared by all the threads, but since it's not used outside `mutlinks()`, it could have been declared on line 18.

Line 29 contains another pragma, `atomic`. This one applies only to the single line following it—line 30, in this case—rather than to a whole block. The purpose of the `atomic` pragma is to avoid what is called a *race condition* in parallel-processing circles. This term describes a situation in which two threads are updating a variable at the same time, which may produce incorrect results. The `atomic` pragma ensures that line 30 will be executed by only one thread at a time. Note that this implies that in this section of the code, our parallel program becomes temporarily serial, which is a potential source of slowdown.

Where is the manager's role in all of this? Actually, the manager is the original thread, and it executes lines 18 and 19, as well as `.C()`, the R function that makes the call to `mutlinks()`. When the worker threads are activated in line 21, the manager goes dormant. The worker threads become dormant once they finish line 31. At that point, the manager resumes execution. Due to the dormancy of the manager while the workers are executing, we do want to have as many workers as our machine has cores.

The function `procpairs()` is straightforward, but note the manner in which the matrix `m` is being accessed. Recall from the discussion in Chapter 15 on interfacing R to C that the two languages store matrices differently: column by column in R and row-wise in C. We need to be aware of that difference here. In addition, we have treated the matrix `m` as a one-dimensional array, as is common in parallel C code. In other words, if `n` is, say, 4, then we treat `m` as a vector of 16 elements. Due to the column-major nature of R matrix storage, the vector will consist first of the four elements of column 1, then the four of column 2, and so on. To further complicate matters, we must keep in mind that array indices in C start at 0, instead of starting at 1 as in R.

Putting all of this together yields the multiplication in line 12. The factors here are the (k,i) and (k,j) elements of the version of `m` in the C code, which are the (i+1,k+1) and (j+1,k+1) elements back in the R code.

16.3.5 Other OpenMP Pragmas

OpenMP includes a wide variety of possible operations—far too many to list here. This section provides an overview of some OpenMP pragmas that I consider especially useful.

16.3.5.1 The omp barrier Pragma

The parallel-processing term *barrier* refers to a line of code at which the threads rendezvous. The syntax for the `omp barrier` pragma is simple:

```
#pragma omp barrier
```

When a thread reaches a barrier, its execution is suspended until all other threads have reached that line. This is very useful for iterative algorithms; threads wait at a barrier at the end of every iteration.

Note that in addition to this explicit barrier invocation, some other pragmas place an implicit barrier following their blocks. These include `single` and `parallel`. There is an implied barrier immediately following line 31 in the previous listing, for example, which is why the manager stays dormant until all worker threads finish.

16.3.5.2 The omp critical Pragma

The block that follows this pragma is a *critical section*, meaning one in which only one thread is allowed to execute at a time. The `omp critical` pragma essentially serves the same purpose as the `atomic` pragma discussed earlier, except that the latter is limited to a single statement.

NOTE *The OpenMP designers defined a special pragma for this single-statement situation in the hope that the compiler can translate this to an especially fast machine instruction.*

Here is the `omp critical` syntax:

```
1 #pragma omp critical
2 {
3     // place one or more statements here
4 }
```

16.3.5.3 The omp single Pragma

The block that follows this pragma is to be executed by only one of the threads. Here is the syntax for the `omp single` pragma:

```
1 #pragma omp single
2 {
3     // place one or more statements here
4 }
```

This is useful for initializing sum variables that are shared by the threads, for instance. As noted earlier, an automatic barrier is placed after the block. This should make sense to you. If one thread is initializing a sum, you wouldn't want other threads that make use of this variable to continue execution until the sum has been properly set.

You can learn more about OpenMP in my open source textbook on parallel processing at *http://heather.cs.ucdavis.edu/parprocbook*.

16.3.6 GPU Programming

Another type of shared-memory parallel hardware consists of graphics processing units (GPUs). If you have a sophisticated graphics card in your machine, say for playing games, you may not realize that it is also a very powerful computational device—so powerful that the slogan "A supercomputer on your desk!" is often used to refer to PCs equipped with high-end GPUs.

As with OpenMP, the idea here is that instead of writing parallel R, you write R code interfaced to parallel C. (Similar to the OpenMP case, *C* here means a slightly augmented version of the C language.) The technical details become rather complex, so I won't show any code examples, but an overview of the platform is worthwhile.

As mentioned, GPUs do follow the shared-memory/threads model, but on a much larger scale. They have dozens, or even hundreds, of cores (depending on how you define *core*). One major difference is that several threads can be run together in a block, which can produce certain efficiencies.

Programs that access GPUs begin their run on your machine's CPU, referred to as the *host*. They then start code running on the GPU, or *device*. This means that your data must be transferred from the host to the device, and after the device finishes its computation, the results must be transferred back to the host.

As of this writing, GPU has not yet become common among R users. The most common usage is probably through the CRAN package `gputools`, which consists of some matrix algebra and statistical routines callable from R. For instance, consider matrix inversion. R provides the function `solve()` for this, but a parallel alternative is available in `gputools` with the name `gpuSolve()`.

For more about GPU programming, again see my book on parallel processing at *http://heather.cs.ucdavis.edu/parprocbook*.

16.4 General Performance Considerations

This section discusses some issues that you may find generally useful in parallelizing R applications. I'll present some material on the main sources of overhead and then discuss a couple of algorithmic issues.

16.4.1 Sources of Overhead

Having at least a rough idea of the physical causes of overhead is essential to successful parallel programming. Let's take a look at these in the contexts of the two main platforms, shared-memory and networked computers.

16.4.1.1 Shared-Memory Machines

As noted earlier, the memory sharing in multicore machines makes for easier programming. However, the sharing also produces overhead, since the two cores will bump into each other if they both try to access memory at the same time. This means that one of them will need to wait, causing overhead. That overhead is typically in the range of hundreds of nanoseconds (billionths of seconds). This sounds really small, but keep in mind that the CPU is working at a subnanosecond speed, so memory access often becomes a bottleneck.

Each core may also have a *cache*, in which it keeps a local copy of some of the shared memory. It's intended to reduce contention for memory among the cores, but it produces its own overhead, involving time spent in keeping the caches consistent with each other.

Recall that GPUs are special types of multicore machines. As such, they suffer from the problems I've described, and more. First, the *latency*, which is the time delay before the first bit arrives at the GPU from its memory after a memory read request, is quite long in GPUs.

There is also the overhead incurred in transferring data between the host and the device. The latency here is on the order of microseconds (millionths of seconds), an eternity compared to the nanosecond scale of the CPU and GPU.

GPUs have great performance potential for certain classes of applications, but overhead can be a major issue. The authors of `gputools` note that their matrix operations start achieving a speedup only at matrix sizes of 1000 by 1000. I wrote a GPU version of our mutual outlinks application, which turned out to have a runtime of 3.0 seconds—about half of the `snow` version but still far slower than the OpenMP implementation.

Again, there are ways of ameliorating these problems, but they require very careful, creative programming and a sophisticated knowledge of the physical GPU structure.

16.4.1.2 Networked Systems of Computers

As you saw earlier, another way to achieve parallel computation is through networked systems of computers. You still have multiple CPUs, but in this case, they are in entirely separate computers, each with its own memory.

As pointed out earlier, network data transfer causes overhead. Its latency is again on the order of microseconds. Thus, even accessing a small amount of data across the network incurs a major delay.

Also note that `snow` has additional overhead, as it changes numeric objects such as vectors and matrices to character form before sending them, say from the manager to the workers. Not only does this entail time for the conversion (both in changing from numeric to character form and

in charging back to numeric at the receiver), but the character form tends to make for much longer messages, thus longer network transfer time.

Shared-memory systems can be networked together, which, in fact, we did in the previous example. We had a hybrid situation in which we formed `snow` clusters from several networked dual-core computers.

16.4.2 Embarrassingly Parallel Applications and Those That Aren't

> It's no shame to be poor, but it's no great honor either.
>
> —Tevye, *Fiddler on the Roof*

> Man is the only animal that blushes, or needs to.
>
> —Mark Twain

The term *embarrassingly parallel* is heard often in talk about parallel R (and in the parallel processing field in general). The word *embarrassing* alludes to the fact that the problems are so easy to parallelize that there is no intellectual challenge involved; they are embarrassingly easy.

Both of the example applications we've looked at here would be considered embarrassingly parallel. Parallelizing the `for i` loop for the mutual outlinks problem in Section 16.1 was pretty obvious. Partitioning the work in the KMC example in Section 16.2.4 was also natural and easy.

By contrast, most parallel sorting algorithms require a great deal of interaction. For instance, consider merge sort, a common method of sorting numbers. It breaks the vector to be sorted into two (or more) independent parts, say the left half and right half, which are then sorted in parallel by two processes. So far, this is embarrassingly parallel, at least after the vector is divided in half. But then the two sorted halves must be merged to produce the sorted version of the original vector, and that process is *not* embarrassingly parallel. It can be parallelized but in a more complex manner.

Of course, to paraphrase Tevye, it's no shame to have an embarrassingly parallel problem! It may not exactly be an honor, but it is a cause for celebration, as it is easy to program. More important, embarrassingly parallel problems tend to have low communication overhead, which is crucial to performance, as discussed earlier. In fact, when most people refer to embarrassingly parallel applications, they have this low overhead in mind.

But what about nonembarrassingly parallel applications? Unfortunately, parallel R code is simply not suitable for many of them for a very basic reason: the functional programming nature of R. As discussed in Section 14.3, a statement like this:

```
x[3] <- 8
```

is deceptively simple, because it can cause the entire vector `x` to be rewritten. This really compounds communication traffic problems. Accordingly, if your application is not embarrassingly parallel, your best strategy is probably to write the computationally intensive parts of the code in C, say using OpenMP or GPU programming.

Also, note carefully that even being embarrassingly parallel does not make an algorithm efficient. Some such algorithms can still have significant communication traffic, thus compromising performance.

Consider the KMC problem, run under `snow`. Suppose we were to set up a large enough number of workers so that each worker had relatively little work to do. In that case, the communication with the manager after each iteration would become a signficant portion of run time. In this situation, we would say that the *granularity* is too fine, and then probably switch to using fewer workers. We would then have larger tasks for each worker, thus a *coarser* granularity.

16.4.3 Static Versus Dynamic Task Assignment

Look again at the loop beginning on line 26 of our OpenMP example, reproduced here for convenience:

```
for (i = me; i < nval; i += nth) {
   mysum += procpairs(i,m,nval);
}
```

The variable `me` here was the thread number, so the effect of this code was that the various threads would work on nonoverlapping sets of values of `i`. We do want the values to be nonoverlapping, to avoid duplicate work and an incorrect count of total number of links, so the code was fine. But the point now is that we were, in effect, preassigning the tasks that each thread would handle. This is called *static* assignment.

An alternative approach is to revise the `for` loop to look something like this:

```
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);
      ...
   }
}
...
```

This is *dynamic* task assignment, in which it is not determined ahead of time which threads handle which values of `i`. Task assignment is done during execution. At first glance, dynamic assignment seems to have the potential for better performance. Suppose, for instance, that in a static assignment

setting, one thread finishes its last value of `i` early, while another thread still has two values of `i` left to do. This would mean our program would finish somewhat later than it could. In parallel-processing parlance, we would have a *load balance* problem. With dynamic assignment, the thread that finished when there were two values of `i` left to handle could have taken up one of those values itself. We would have better balance and theoretically less overall runtime.

But don't jump to conclusions. As always, we have the overhead issue to reckon with. Recall that a `critical` pragma, used in the dynamic version of the code above, has the effect of temporarily rendering the program serial rather than parallel, thus causing a slowdown. In addition, for reasons too technical to discuss here, these pragmas may cause considerable cache activity overhead. So in the end, the dynamic code could actually be substantially slower than the static version.

Various solutions to this problem have been developed, such as an OpenMP construct named `guided`. But rather than present these, the point I wish to make is that they are unnecessary. In most situations, static assignment is just fine. Why is this the case?

You may recall that 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 our 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. Thus, they will all finish at pretty close to the same time, and we do not need to worry about load imbalance. Dynamic scheduling will not be necessary.

This reasoning does depend on a statistical assumption, but in practice, the assumption will typically be met sufficiently well for the outcome: Static scheduling does as well as dynamic in terms of uniformity of total work times across threads. And since static scheduling doesn't have the overhead problems of the dynamic kind, in most cases the static approach will give better performance.

There is one more aspect of this to discuss. To illustrate the issue, consider again the mutual outlinks example. Let's review the outline of the algorithm:

```
1  sum = 0
2  for i = 0...n-1
3     for j = i+1...n-1
4        for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
5  mean = sum / (n*(n-1)/2)
```

Say `n` is 10000 and we have four threads, and consider ways to partition the `for i` loop. Naively, we might at first decide to have thread 0 handle the `i` values 0 through 2499, thread 1 handle 2500 through 4999, and so on. However, this would produce a severe load imbalance, since the thread that

handles a given value of i does an amount of work proportional to n-i. That, in fact, is why we staggered the values of i in our actual code: Thread 0 handled the i values 0, 4, 8 ..., thread 1 worked on 1, 5, 9, ..., and so on, yielding good load balance.

The point then is that static assignment might require a bit more planning. One general approach to this is to randomly assign tasks (i values, in our case here) to threads (still doing so at the outset, before work begins). With a bit of forethought such as this, static assignment should work well in most applications.

16.4.4 Software Alchemy: Turning General Problems into Embarrassingly Parallel Ones

As discussed earlier, it's difficult to attain good performance from nonembarrassingly parallel algorithms. Fortunately, for statistical applications, there is a way to turn nonembarrassingly parallel problems into embarrassingly parallel ones. The key is to exploit some statistical properties.

To demonstrate the method, let's once again turn to our mutual outlinks problem. The method, applied with w workers on a links matrix m, consists of the following:

1. Break the rows of m into w chunks.
2. Have each worker find the mean number of mutual outlinks for pairs of vertices in its chunk.
3. Average the results returned by the workers.

It can be shown mathematically that for large problems (the only ones you would need parallel computing for anyway), this chunked approach gives the estimators of the same statistical accuracy as in the nonchunked method. But meanwhile, we've turned a nonparallel problem into not just a parallel one but an embarrassingly parallel one! The workers in the preceding outline compute entirely independently of each other.

This method should not be confused with the usual chunk-based approaches in parallel processing. In those, such as the merge-sort example discussed on page 347, the chunking is embarrassingly parallel, but the combining of results is not. By contrast, here the combining of results consists of simple averaging, thanks to the mathematical theory.

I tried this approach on the mutual outlinks problem in a 4-worker snow cluster. This reduced the runtime to 1.5 seconds. This is far better than the serial time of about 16 seconds, double the speedup obtained by the GPU and approaching comparability to the OpenMP time. And the theory showing that the two methods give the same statistical accuracy was confirmed as well. The chunked method found the mean number of mutual outlinks to be 249.2881, compared to 249.2993 for the original estimator.

16.5 Debugging Parallel R Code

Parallel R packages such as `Rmpi`, `snow`, `foreach`, and so on do not set up a terminal window for each process, thus making it impossible to use R's debugger on the workers. (My `Rdsm` package, which adds a threads capability to R, is an exception to this.)

What then can you do to debug apps for those packages? Let's consider `snow` for a concrete example.

First, you should debug the underlying single-worker function, such as `mtl()` in Section 16.2. Here, we would set up some artificial values of the arguments and then use R's ordinary debugging facilities.

Debugging the underlying function may be sufficient. However, the bug may be in the arguments themselves or in the way we set them up. Then things get more difficult.

It's even hard to print out trace information, such as values of variables, since `print()` won't work in the worker processes. The `message()` function may work for some of these packages; if not, you may need to resort to using `cat()` to write to a file.

Regression

Linear Regression

Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called predictor variable whose value is gathered through experiments. The other variable is called response variable whose value is derived from the predictor variable.

In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

The general mathematical equation for a linear regression is −

```
y = ax + b
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are constants which are called the coefficients.

StepstoEstablishaRegression

A simple example of regression is predicting weight of a person when his height is known. Todo this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height andcorresponding weight.

- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation usingthese
- Get a summary of the relationship model to know the average error in prediction. Alsocalled **residuals**.
- To predict the weight of new persons, use the **predict()** function in R.

Input Data

Below is the sample data representing the observations −

```
# Values of height
151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight.
63, 81, 56, 91, 47, 57, 76, 72, 62, 48
```

lm()Function

This function creates the relationship model between the predictor and the

response variable.Syntax

The basic syntax for **lm()** function in linear regression is −

```
lm(formula,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create Relationship Model & get the Coefficients

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

Apply the lm() function

.relation <- lm(y~x) print(relation)

When we execute the above code, it produces the following result −

```
Call:
lm(formula = y

Coefficien
ts:              x
(Intercept    0.67
```

Get the Summary of the Relationship

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

Apply the lm() function.

relation <- lm(y~x)

print(summary(relation))

When we execute the above code, it produces the following result −

Output:

```
Call:
lm(formula = y ~ x)

Residuals:
  Min    1Q   Median   3Q   Max
-6.3002-1.6629  0.0412     1.8944  3.9775

Coefficients:
       Estimate Std. Error t value
Pr(>|t|) (Intercept) -38.45509
                    8.04901  -4.778
0.00139 **
```

```
Residual standard error: 3.253 on 8 degrees of
freedom Multiple R-squared:  0.9548,
                              Adjusted R-squared:
```

predict() FunctionSyntax

The basic syntax for predict() in linear regression is –

predict(object, newdata)

Following is the description of the parameters used −

- **object** is the formula which is already created using the lm() function.
- **newdata** is the vector containing the new value for predictor variable.

Predict the weight of new persons

```
# The predictor vector.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

# The resposne vector.

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

# Find weight of a person with height 170

a <- data.frame(x = 170)

result <-  predict(relation,a)

print(result)
```

When we execute the above code, it produces the following result −

```
      1
76.22869
```

Visualize the Regression Graphically

Create the predictor and response variable.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

relation <- lm(y~x)

Give the chart file a name.

png(file = "linearregression.png")

Plot the chart.

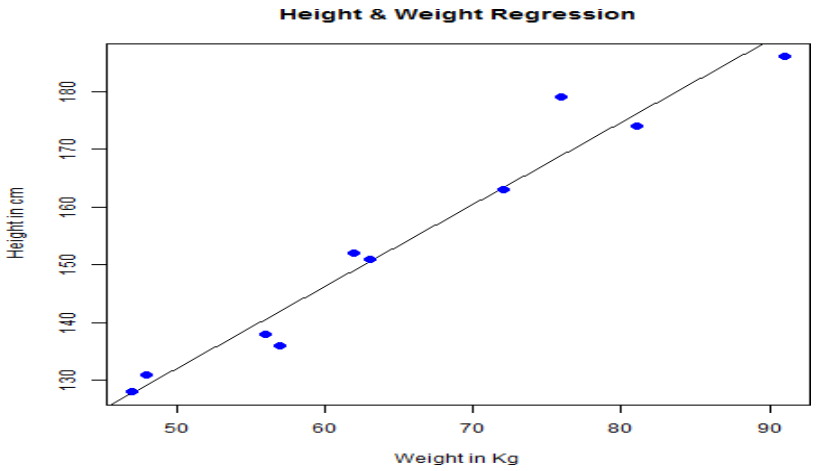
plot(y,x,col = "blue",main = "Height & Weight Regression",

abline(lm(x~y)), cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")

Save the file.

dev.off()

When we execute the above code, it produces the following result −



R-MultipleRegression

Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is −

```
y = a + b1x1 + b2x2 +...bnxn
```

Following is the description of the parameters used −

- **y** is the response variable.
- **a, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

We create the regression model using the **lm()** function in R. The model determines the value ofthe coefficients using the input data. Next we can predict the value of the response variable for agiven set of predictor variables using these coefficients.

lm()Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in multiple regression is −

```
lm(y ~ x1+x2+x3...,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between the response variable and predictorvariables.
- **data** is the vector on which the formula will be applied.

Example:

Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

```
input <- cars[,c("mpg","disp","hp","wt")]
```

Create Relationship Model & get the Coefficients

```
input <- mtcars[,c("mpg","disp","hp","wt")]

# Create the relationship model.
model <- lm(mpg~disp+hp+wt, data = input)

# Show the model.
print(model)

# Get the Intercept and coefficients as vector elements.
```

When we execute the above code, it produces the following result −

```
Call:
lm(formula = mpg ~ disp + hp + wt,

Coefficien
ts:          dis       h       w
(Intercept  -0.000937  -0.031157  -
```

Create Equation for Regression Model

Based on the above intercept and coefficient values, we create the mathematical equation.

```
Y = a+Xdisp.x1+Xhp.x2+Xwt.x3or
Y = 37.15+(-0.000937)*x1+(-0.0311)*x2+(-3.8008)*x3
```

Apply Equation for predicting New Values

We can use the regression equation created above to predict the mileage when a new set of values for displacement, horse power and weight is provided.

For a car with disp = 221, hp = 102 and wt = 2.91 the predicted mileage is

```
Y = 37.15+(-0.000937)*221+(-0.0311)*102+(-3.8008)*2.91 = 22.7104
```

R-LogisticRegression

The Logistic Regression is a regression model in which the response variable (dependentvariable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is −

```
y = 1/(1+e^-(a+b1x1+b2x2+b3x3+...))
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are the coefficients which are numeric constants.

The function used to create the regression model is the **glm()** function.

The basic syntax for **glm()** function in logistic regression is −

Generalized Linear Models Using R

GLMs (Generalized linear models) are a type of statistical model that is extensively used in the analysis of non-normal data, such as count data or binary data. They enable us to describe the connection between one or more predictor variables and a response variable in a flexible manner.

Major components of GLMs

- A probability distribution for the response variable
- A linear predictor function of the predictor variables
- A link function that connects the linear predictor to the response variable's mean.

The probability distribution and link function used is determined by the type of response variable and the research topic at hand. R includes methods for fitting GLMs, such as the glm() function. The user can specify the formula for the model, which contains the response variable and one or more predictor variables, as well as the probability distribution and link function to be used, using this function.

Mathematical Formulation of GLM

In Generalized Linear Models (GLMs), the response variable Y is assumed to follow a distribution from the exponential family. The model relates the expected value of Y, denoted μ, to the predictors X via a link function:

$$g(\mu)=X\beta$$

Here, β is the vector of model coefficients and $g(\cdot)$ is a specified link function. The variance of Y is given by:

$$Var(Y)=\phi V(\mu)$$

where $V(\mu)$ is the variance function and ϕ is a dispersion parameter.

Classical Linear Regression as a Special Case

In linear regression, $Y=X\beta+\epsilon$, with $\epsilon\sim N(0,\sigma 2)$, is a special case where:

- $g(\mu)=\mu$ (identity link)
- $V(\mu)=1$
- $\phi=\sigma^2$

Estimation

Model parametersβ are estimated via maximum likelihood. For observations (xi,yi), the likelihood is:

$$L(\beta)=\prod^{n}_{i=1} f(y_i|\mu_i)$$

where $f(\cdot)$ is the density function of the assumed distribution and μ_i is the expected value of Y_i given x_i.

GLM model families

There are several GLM model families depending on the make-up of the response variable. These includes three well-known GLM model families:

- **Binomial:** The binomial family is used for binary response variables (i.e., two categories) and assumes a binomial distribution.

```
model <- glm(binary_response_variable ~ predictor_variable1 + predictor_variable2,family =
binomial(link = "logit"), data = data)
```

- **Gaussian:** This family is used for continuous response variables and assumes a normal distribution. The link function for this family is typically the identity function.

```
model <- glm(response_variable ~ predictor_variable1 + predictor_variable2,
        family = gaussian(link = "identity"), data = data)
```

- **Gamma:** The gamma family is used for continuous response variables that are strictly positive and have a skewed distribution.

```
model <- glm(positive_response_variable ~ predictor_variable1 + predictor_variable2,family =
gamma(link = "inverse"), data = data)
```

- **Quasibinomial:** When a response variable is binary but has a higher variance than would be predicted by a binomial distribution, the quasibinomial model is utilized. This could happen if the response variable has excessive dispersion or additional variation that the model is not taking into account.

```
model <- glm(response_variable ~ predictor_variable1 + predictor_variable2,
        family = quasibinomial(), data = data)
```

Building a Generalized Linear Model

1. Loading the Dataset

We will use the "mtcars" dataset in R to illustrate the use of generalized linear models. This dataset includes data on different car models, including mpg, horsepower (hp) and weight. (wt). The response variable will be "mpg," and the predictor factors will be "hp" and "wt."

```
data(mtcars)
head(mtcars)
```

To create a generalized linear model in R, we must first select a suitable probability distribution for the answer variable.

- If the answer variable is binary (e.g., 0 or 1), we could use the Bernoulli distribution.
- If the response variable is a count (for example, the number of vehicles sold), the Poisson distribution may be used.

2. Building the model

To create a generalized linear model in R, use the glm() tool. We must describe the model formula (the response variable and the predictor variables) as well as the probability distribution family.

- data(mtcars)

- model <- glm(mpg ~ hp + wt, data = mtcars, family = gaussian)

The Gaussian family is used in this example, which implies that the response variable has a normal distribution.

3. Calculate summary of the model

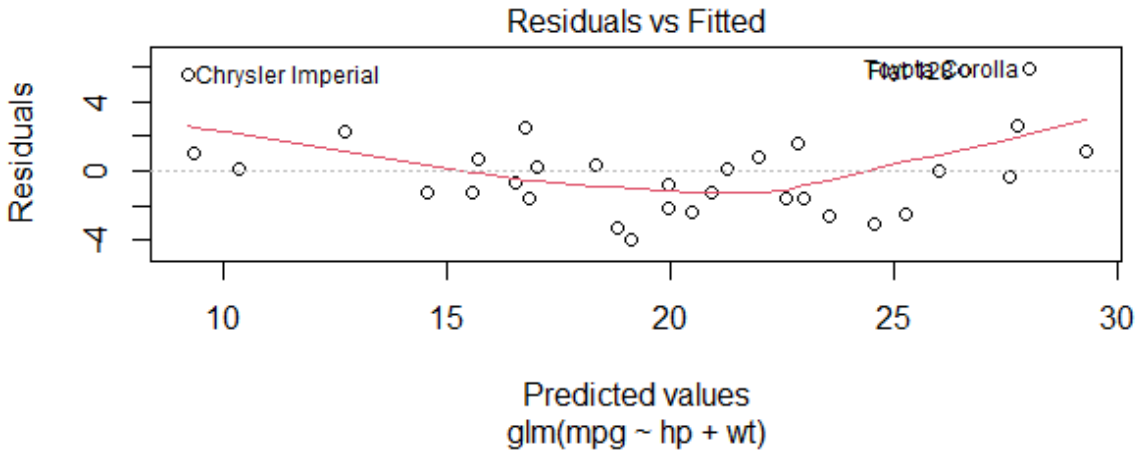
```
summary(model)
```

A one unit hp increase predicts a 0.03177 mpg decrease, while one unit wt increase predicts a 3.87783 mpg decrease.

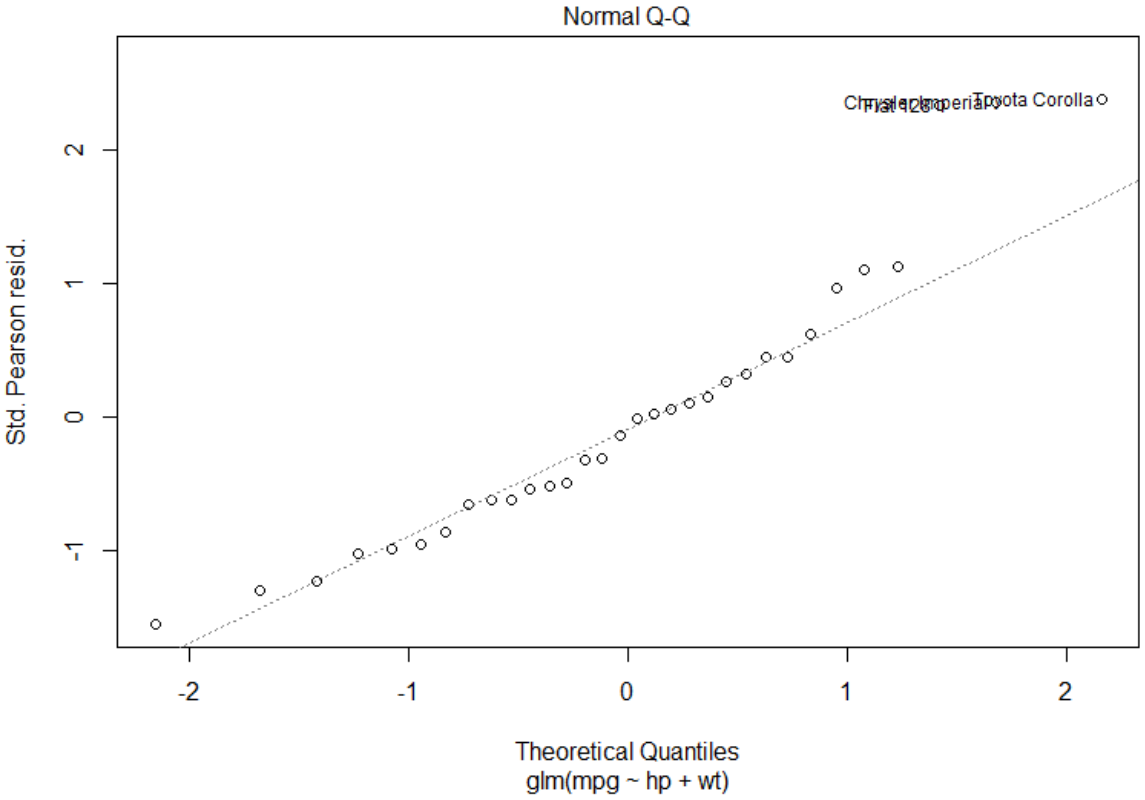
4. Visualize the model

After creating an extended linear model, we must evaluate its fit to the data. This can be accomplished with the help of diagnostic graphs such as the residual plot and the Q-Q plot.

```
plot(model, which = 1)

plot(model, which = 2)
```



The residual plot displays the residuals (differences between measured and predicted values) plotted against the fitted values. (i.e. the predicted values). We want to see a random scatter of residuals around zero, which indicates that the model is capturing the data trends.



The residuals Q-Q plot displays the residuals plotted against the anticipated values if they were normally distributed. The points should follow a straight line, showing that the residuals are normally distributed.

Time series Analysis

Time series is a series of data points in which each data point is associated with a timestamp. A simple example is the price of a stock in the stock market at different points of time on a given day. Another example is the amount of rainfall in a region at different months of the year. R language uses many functions to create, manipulate and plot the time series data. The data for the time series is stored in an R object called time-series object. It is also a R data object like a vector or data frame.The time series object is created by using the ts() function.

Syntax
The basic syntax for ts() function in time series analysis is –

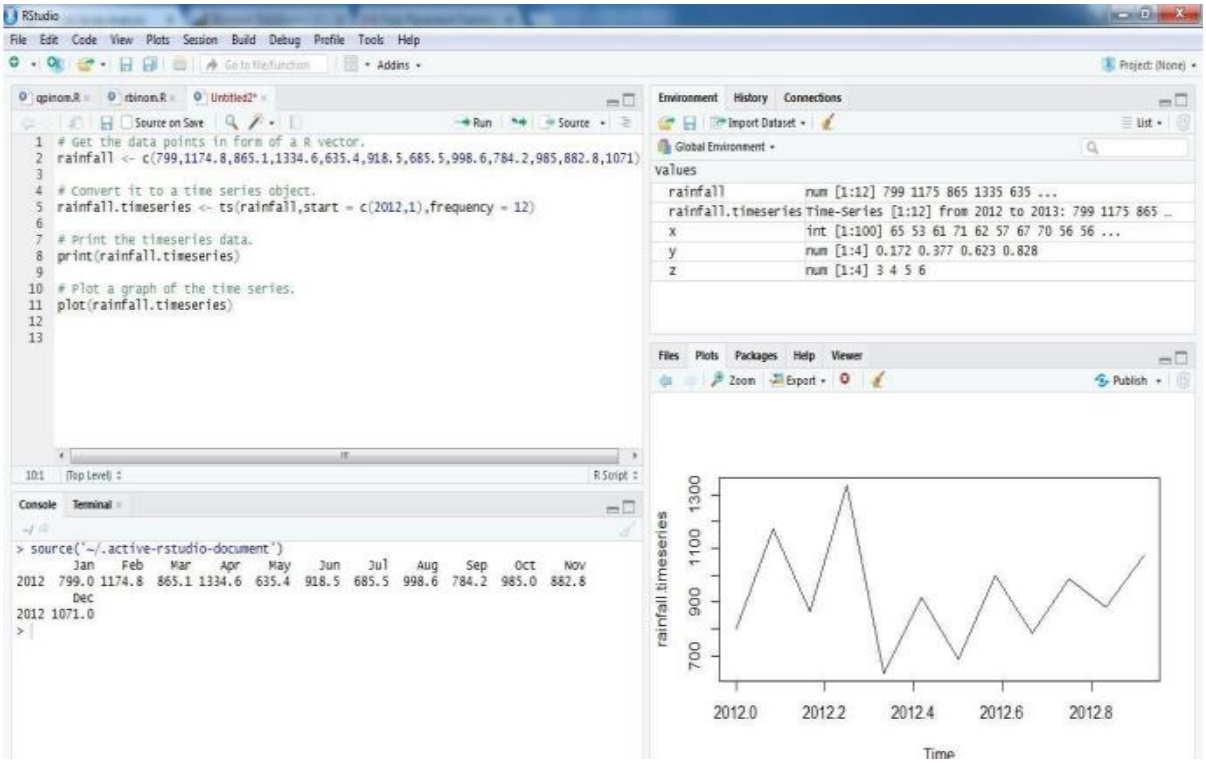
timeseries.object.name <- ts(data, start, end, frequency)

Following is the description of the parameters used –
- data is a vector or matrix containing the values used in the time series.
- start specifies the start time for the first observation in time series.
- end specifies the end time for the last observation in time series.
- frequency specifies the number of observations per unit time.

Except the parameter "data" all other parameters are optional.

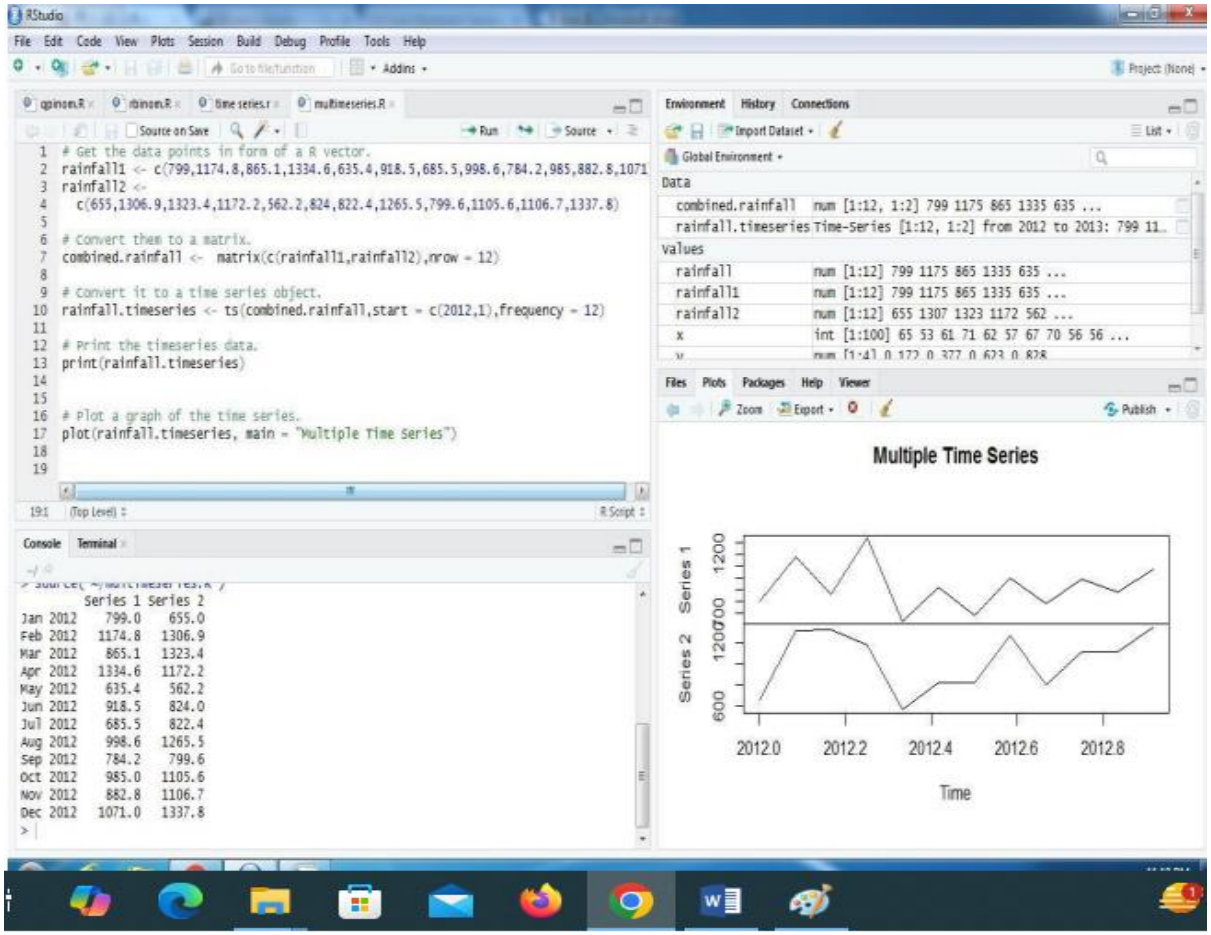
Example:

Consider the annual rainfall details at a place starting from January 2012. We create an R time series object for a period of 12 months and plot it.



MultipleTimeSeries

We can plot multiple time series in one chart by combining both the series into a matrix.



Autocorrelation

Autocorrelation is used to measure the degree of similarity between a time series and a lagged version of itself over the given range of time intervals. We can also call autocorrelation as "serial correlation" or "lagged correlation". It is mainly used to measure the relationship between the actual values and the previous values.

In R, we can calculate the autocorrelation in a vector by using the module tseries. Within this module, we have to use acf() method to calculate autocorrelation.

Syntax:
acf(vector, lag, pl)

Parameter:

- *vector is the input vector*
- *lag represents the number of lags*
- *pl is to plot the auto correlation*

Example: R program to calculate auto correlation in a vector with different lags

```
# load tseries module
library(tseries)

```

```
# create vector1 with 8 time periods
vector1=c(34,56,23,45,21,64,78,90)

# calculate auto correlation with no lag
print(acf(vector1,pl=FALSE))

# calculate auto correlation with lag 0
print(acf(vector1,lag=0,pl=FALSE))

# calculate auto correlation with lag 2
print(acf(vector1,lag=2,pl=FALSE))

# calculate auto correlation with lag 6
print(acf(vector1,lag=6,pl=FALSE))
```

Output:

```
Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6      7
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064 -0.151

Autocorrelations of series 'vector1', by lag

0
1

Autocorrelations of series 'vector1', by lag

    0     1     2
1.000 0.257 0.208

Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064
```

The same function can be used to visualize the output produced for that we simply have to set pl to TRUE

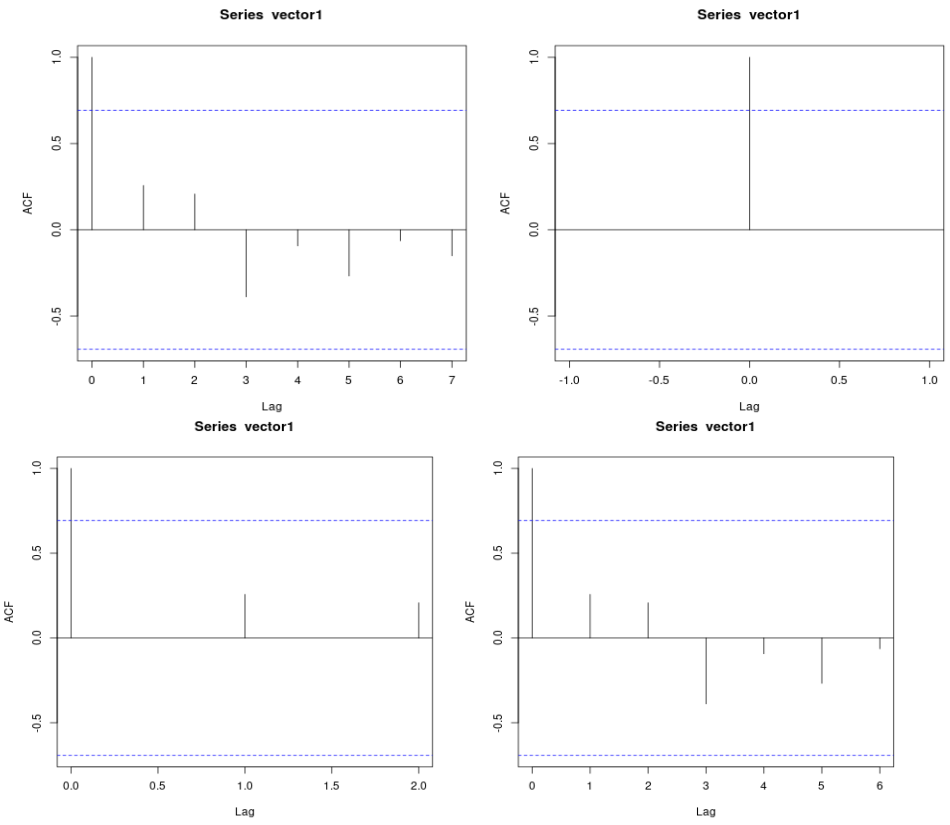
Example: Data visualization

```
# load tseries module
library(tseries)

# create vector1 with 8 time periods
vector1=c(34,56,23,45,21,64,78,90)

# calculate auto correlation with no lag
print(acf(vector1,pl=TRUE))

# calculate auto correlation with lag 0
print(acf(vector1,lag=0,pl=TRUE))
```

-
- *# calculate auto correlation with lag 2*
- print(acf(vector1,lag=2,pl=**TRUE**))
-
- *# calculate auto correlation with lag 6*
- print(acf(vector1,lag=6,pl=**TRUE**))

Output:

```
Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6      7
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064 -0.151

Autocorrelations of series 'vector1', by lag

0
1

Autocorrelations of series 'vector1', by lag

    0     1     2
1.000 0.257 0.208

Autocorrelations of series 'vector1', by lag

     0      1      2      3      4      5      6
 1.000  0.257  0.208 -0.389 -0.093 -0.268 -0.064
```

- 

Clustering in R Programming

Clustering is an unsupervised learning technique where a dataset is divided into groups, or clusters, based on similarities among data points. It helps identify natural groupings within the data without prior labeling. Each cluster has data points that are closer to one another than to other clusters. This approach is commonly employed in data mining and pattern recognition to reveal latent structures, group behavior, or trends in the data. Clustering is particularly helpful when we need to explore data, reduce dimensionality, or pre-process data for additional supervised learning activities.

Types of Clustering in R Programming
In **R**, there are several clustering techniques, each suited for different data types and clustering challenges. Each method has its own advantages and is designed to handle specific data characteristics such as the number of clusters, their shapes, and whether or not noise is present in the data.

1. K-means clustering
The most common method is K-means, where the number of clusters (k) is set beforehand. It is computationally efficient for large datasets but can struggle with irregularly shaped clusters or varying densities. it is a data-partitioning technique that seeks to assign each observation to the cluster with the closest mean after dividing the data into k clusters.

2. Hierarchical Clustering
Hierarchical Clustering, on the other hand, creates a hierarchy of clusters by either merging smaller clusters (agglomerative) or splitting larger ones (divisive). This approach builds a tree like structure known as a dendrogram, which allows you to see the relationships between clusters at various levels of similarity. Hierarchical clustering is great for small to medium datasets where understanding the relationships between clusters is important, but it can be computationally expensive for larger datasets.

3. Spectral Clustering
Spectral Clustering transforms the clustering problem into a graph partitioning problem. By constructing a similarity graph from the data and performing clustering based on the eigenvalues of the graph's Laplacian matrix, it is able to capture complex, non convex clusters. This method works particularly well for datasets where the clusters are not linearly separable, but it can be computationally intensive and requires careful tuning of the similarity matrix.

4. Fuzzy Clustering
Fuzzy Clustering (or Fuzzy C-Means) is a soft clustering technique where data points are assigned membership scores for each cluster, rather than being definitively assigned to one cluster. This means a data point can belong to multiple clusters with varying degrees of membership. Fuzzy clustering is useful when the boundaries between clusters are not well defined, and it allows for more nuanced grouping, but interpreting the membership scores can be more complex.

5. Density Based Clustering
Density Based Clustering is a broader category that includes methods like DBSCAN. These methods focus on finding clusters based on regions of high data density, rather than relying on a distance metric. Density based methods are robust to noise and can find clusters of

arbitrary shapes. However, they can be sensitive to the parameters used, such as the minimum number of points needed to form a cluster.

6. Ensemble Clustering

Ensemble Clustering takes a different approach by combining the results of multiple clustering algorithms or multiple runs of the same algorithm to create a more reliable clustering solution. By aggregating the results of different methods, ensemble clustering aims to improve performance and reduce the risk of overfitting. This method is particularly useful when there is uncertainty about which clustering technique is the most appropriate, and it can provide more robust and stable results.

Each of these clustering techniques has its strengths and weaknesses, making it important to choose the right one based on the specific characteristics of your data and the goals of your analysis. Whether you're working with large datasets, noisy data, or data that requires soft assignments, there's a clustering method in R that can be tailored to your needs.

Implementation of K-Means Clustering in R Programming

We will implement K-Means Clustering algorithm here since it is simple and easy to understand. K-Means is an iterative hard clustering technique that uses an unsupervised learning algorithm. In this, total numbers of clusters are pre defined by the user and based on the similarity of each data point, the data points are clustered. This algorithm also finds out the centroid of the cluster.

Algorithm

1. **Specify number of clusters (K):** Let us take an example of k = 2 and 5 data points.
2. **Randomly assign each data point to a cluster:** In the below example, the red and green color shows 2 clusters with their respective random data points assigned to them.
3. **Calculate cluster centroids:** The cross mark represents the centroid of the corresponding cluster.
4. **Reallocate each data point to their nearest cluster centroid:** Green data point is assigned to the red cluster as it is near to the centroid of red cluster.
5. **Reconfigure cluster centroid.**

Syntax:

kmeans(x, centers, nstart)

where,

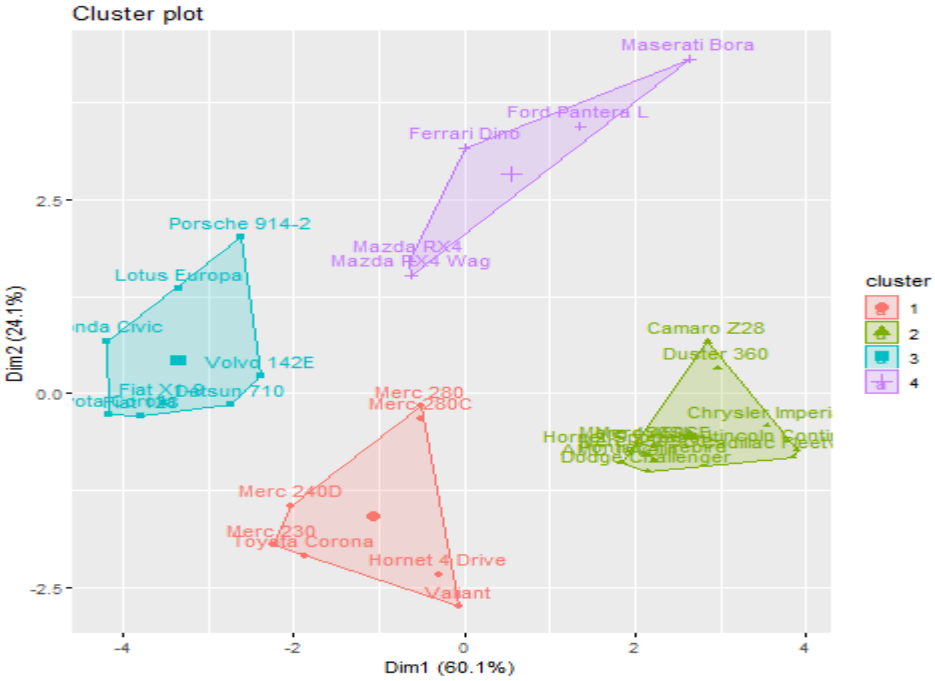
- **x :** represents numeric matrix or data frame object.
- **centers** : represents the **K** value or distinct cluster centers.
- **nstart** : represents number of random sets to be chosen.

Example

```
install.packages("factoextra")
library(factoextra)

df <- mtcars
df <- na.omit(df)
df <- scale(df)

km <- kmeans(df, centers = 4, nstart = 25)
fviz_cluster(km, data = df)

km <- kmeans(df, centers = 5, nstart = 25)
fviz_cluster(km, data = df)
```

Output:

When k = 4



When k = 5